UNIVERSIDADE FEDERAL DE SÃO CARLOS
CAMPUS SOROCABA
Bacharelado em Ciência da Computação

SISTEMA DE BANCO DE DADOS

Profª. Dra. Sahudy Montenegro González

PROJETO PRÁTICO

Grupo 14 - Tema: Rede de Hotel

Felipe Ottoni Pereira
Letícia Almeida Paulino de Alencar Ferreira
Ricardo Yugo Suzuki

20 de Agosto de 2023

Fase Final: Correções Fase Intermediária + Controle de Acesso

ÍNDICE

LISTA DE TABELAS

LISTA DE FIGURAS

1. DESCRIÇÃO DO PROBLEMA

Uma rede de hotéis chamada Hamptons utiliza um sistema para organização e gerenciamento de seus hotéis, hóspedes, quartos e reservas. Desse modo, com hotéis espalhados pelo Brasil, é preciso registrar seus números identificadores, nomes, endereços e estrelas (1 a 5). A estrutura de rede é projetada de modo que todo hotel possui quartos e todo quarto pertence a um hotel.

Por conseguinte, para gerenciar os quartos eles são catalogados pelo número identificador, relacionado ao andar a que pertencem, junto ao identificador do hotel a que pertencem, sua capacidade máxima de hóspedes (2 a 6 pessoas), categoria (suíte simples, suite duplo, cobertura, presidencial) e seu preço por noite (R$ 150,00 - R$ 5.000,00).

Além disso, a rede realiza o cadastro dos hóspedes para que as reservas sejam feitas, uma vez que um cliente pode realizar variadas reservas, que são relacionadas a um titular, então, são necessárias as seguintes informações sobre um hóspede: CPF, nome completo e endereço. Dessa forma, uma reserva é caracterizada pelo seu identificador, o CPF do titular da reserva, o número do quarto, o hotel, o modo de pagamento (cartão de crédito, cartão de débito, dinheiro), a data de entrada e a data de saída, assim, guardando um histórico de reservas. Portanto, o sistema da rede Hamptons busca estruturar e administrar seu funcionamento.

Esquema Relacional

Hotel (id_hotel, nome_hotel, estrelas, endereco, estado)
Quarto (numero,id_hotel, capacidade, categoria, preco)
    id_hotel referencia Hotel
Hospede (cpf_hospede, nome, endereco, estado)
Reserva (cpf_hospede,numero_quarto, id_hotel, data_entrada, data_saida,dia_entrada, mes_entrada, ano_entrada, dia_saida, mes_saida, ano_saida, modo_pagamento)
    cpf_hospede referencia Hospede
    numero_quarto, id_hotel referencia Quarto

O modelo acima atende a 3° forma normal, porque as tabelas estão na 1° forma normal, uma vez que não têm atributos multivalorados, também está na 2° Forma Normal, pois não há dependência funcional parcial, e não possui dependência funcional transitiva.

2. ELABORAÇÃO DAS CONSULTAS

Foram elaboradas três consultas que trazem informações referentes à busca por quartos disponíveis, por informações de hóspedes e as receitas dos hotéis. O objetivo dessas consultas é permitir que as equipes dos hotéis gerenciem a disponibilidade dos quartos em relação às reservas, mas também que possam analisar dados de hóspedes e as receitas dos hotéis. Portanto, são elas:

C1 | Felipe

Quais quartos na localização X , da categoria Y, entre a data A e a data B, com capacidade Z, estão disponíveis?

- **Campos de visualização do resultado:** `numero, id_hotel, nome_hotel, estrelas, endereco, preco;`
- **Campos de busca (ou das condições):** `endereco_hotel(relativa), categoria(absoluta), data_entrada(absoluta), data_saida(absoluta) e capacidade(absoluta);`
- **Operadores das condições:** `endereco_hotel(LIKE), categoria(=), data_entrada(>=), data_saida(<=) e capacidade(=);`

SQL

```
SELECT  numero, id_hotel, nome_hotel, estrelas, endereco,preco
FROM quarto NATURAL JOIN hotel NATURAL JOIN (
    SELECT id_hotel, numero FROM (
        SELECT id_hotel, nome_hotel, estrelas, endereco,   numero, preco
        FROM hotel ho NATURAL JOIN quarto q
        WHERE
        estado LIKE '<X>' AND
        categoria = '<Y>' AND
        capacidade = <Z>
    )t1
    INTERSECT
    SELECT id_hotel, numero  FROM (
        SELECT id_hotel, numero
        FROM quarto
        EXCEPT
        SELECT id_hotel, numero_quarto
```

```
        FROM reserva
        WHERE ('<A>', '<B>') OVERLAPS
          (data_entrada,data_saida)
    )t2
)t3;
```

C2 | Letícia

Liste os dados dos hóspedes de sobrenome 'X' e suas reservas que foram pagas pelo modo de pagamento 'Y', a partir do ano 'Z'.

- **Campos de visualização do resultado:** `nome, endereco, id_hotel, numero_quarto, data_entrada, data_saida;`
- **Campos de busca (ou das condições):** `nome(relativo), modo_pagamento(absoluto), ano_entrada(absoluto);`
- **Operadores das condições:** `modo_pagamento(=), nome(ILIKE), ano_entrada(>=);`

SQL

```
SELECT ho.nome, ho.endereco, r.id_hotel, r.numero_quarto, r.data_entrada,
r.data_saida
FROM hospede ho, reserva r
WHERE ho.cpf_hospede = r.cpf_hospede
AND ho.nome ILIKE '<X>'
AND r.modo_pagamento = '<Y>'
AND r.ano_entrada >= <Z>;
```

C3 | Ricardo

Mostre os hotéis e suas receitas no mês X e ano Y. Considerando que a receita mensal dos hotéis é calculada considerando que os hóspedes pagam suas reservas ao saírem do hotel.

- **Campos de visualização do resultado:** `id_hotel, nome_hotel, estrelas, endereco, receita_mensal;`
- **Campos de busca (ou das condições):** `mes(absoluto), ano(absoluto);`
- **Operadores das condições:** `mes(=), ano(=);`

SQL

```
SELECT h.id_hotel, h.nome_hotel, h.estrelas, h.endereco,
  SUM(q.preco * (DATE_PART('day', AGE(DATE_TRUNC('day', r.data_saída),
DATE_TRUNC('day', r.data_entrada))))) AS receita_mensal
FROM hotel h natural join quarto q , reserva r
WHERE
  q.numero = r.numero_quarto
  AND q.id_hotel = r.id_hotel
  AND EXTRACT(MONTH FROM r.data_saida) = <mes>
  AND EXTRACT(YEAR FROM r.data_saida) = <ano>
GROUP BY h.id_hotel;
```

3. POPULANDO O BANCO DE DADOS

Com a criação do projeto físico do banco de dados do *database* "Hamptons" e das criações das tabelas `hotel`, `quarto`, `hospede` e `reserva`, respectivamente, houve a população por meio de *scripts* em python de cada tabela. Nesse sentido, a tabela de metadados especifica quantos registros têm cada tabela do banco de dados e o seus tamanhos, em bytes, após a população.

Nome da tabela	Número de registros	Tamanho (bytes)
`Hotel`	100	80 kB | 81.920 bytes
`Quarto`	10.100	904 kB | 925.696 bytes
`Hospede`	500.000	71 MB | 74.522.624 bytes
`Reserva`	1.000.000	119 MB | 124.715.008 bytes
`Banco de Dados`	1.510.200	198 MB | 208.032.559 bytes

Tabela 1: Metadados das tabelas do banco de dados.

Para popular o banco de dados foram gerados arquivos *python* para cada tabela a fim de gerar e inserir dados no banco de dados, dessa maneira, utilizamos bibliotecas do próprio *python* para gerar os dados necessários e criamos funções para garantir a geração correta durante as inserções. Além disso, também criamos *triggers* para impedir que erros de inserção ocorressem.

As bibliotecas do python utilizadas foram a *psycopg2* a fim de criar a conexão com o banco de dados em postgresql , a *faker* para gerarmos os nomes dos hóspedes e os endereços para os hotéis e hóspedes, , a *random* para trabalharmos com a criação de atributos do tipo inteiros, para gerar os CPFs fizemos uma função que também utiliza a *random,* e *datetime* com o intuito de gerar as datas de entrada e saída para inserções em reserva.

Para restaurar o backup, gerado após a população do banco de dados, precisa-se do arquivo "Rede_hoteis.backup" e criar um banco de dados. Assim, no pgAdmin ao clicar com o botão direito no banco de dados criado e escolher a opção "Restore...", com isso aparecerá uma janela com o campo "Filename" em que deve-se escolher o arquivo "Rede_hoteis.backup", clicar em "Restore" e aguardar "process completed" ser exibido. Portanto, após esses passos o backup foi restaurado e o banco de dados estará preenchido com as tabelas e seus dados.

Então, para consultar os tamanhos do banco de dados e das tabelas foi utilizado as seguintes consultas com funções do Postgresql, respectivamente:

- `SELECT pg_size_pretty(pg_database_size('<nome_bd>'))`
- `SELECT pg_size_pretty(pg_total_relation_size('<tabela>'))`

4. TÉCNICAS DE ACESSO EFICIENTE AO BANCO DE DADOS

Devido ao volume de dados no banco, as consultas podem usar uma grande quantidade de memória e ter tempo de resposta elevado, logo, com o objetivo de evitar isto, é necessário otimizá-las de modo que o custo das consultas seja o menor possível.

Para saber a versão do PostgreSQL a instrução ‘`SELECT version()`’ foi executada e retornou "PostgreSQL 15.2, compiled by Visual C++ build 1914, 64-bit", junto com a versão 6.1 do PgAdmin. Além disso, realizamos a limpeza do cache do PostgresSQL a fim de obter tempos de execução de consultas corretos.

CONSULTA 1

Reescrita e criação de índices

O uso do `EXISTS` ao invés do `EXCEPT` é responsável por otimizar o processamento, uma vez que ele apenas verifica se uma subconsulta retorna resultado, parando de avaliar assim que encontra o primeiro retorno, sendo mais direto.

A igualdade no lugar do `LIKE` não faz diferença na realidade, pois o `LIKE` não está sendo utilizado com expressões regulares nesse caso. Entretanto inicialmente foi considerado obter o estado a partir do campo `endereco`, sendo que seria necessário o uso de expressões regulares e caracteres curingas para fazer a busca do estado dentro da string, portanto, mesmo na consulta inicial, já foi logicamente decidido utilizar o campo `estado` para recuperá-lo. Além disso, para essa consulta o uso dos campos do tipo `date` são mais eficientes do que o uso dos campos com dia, mês e ano separados, pois as comparações e as filtragens são realizadas em cima da data como um todo, não sendo necessário gastar mais processamento analisando campo a campo.

O uso do `OVERLAPS` foi substituído por operadores de comparação porque, utilizando o `OVERLAPS`, ele acabava não aceitando o índice composto criado para as datas. Além disso foi criado um índice composto para as datas:

```
CREATE INDEX index_datas on reserva (data_entrada, data_saida);
```

Pois frequentemente os campos `data_entrada, data_saida` serão utilizados em conjuntos em diversas outras consultas prováveis do minimundo, porém esse índice não era utilizado, pois o programa preferia utilizar o índice da chave primária de reserva, pois ele possuia mais campos em comuns (numero_quarto, id_hotel, data_entrada), porém foi percebido na correção para a fase final que a criação de um índice compostos com esses campos da chave primária acrescentado do campo data_saida seria vantajoso para o otimizador, pois possuiria mais campos em comum para auxiliar nas filtragens, sendo a criação desse índice dada por:

```
CREATE INDEX index_reserva on reserva (numero_quarto, id_hotel, data_entrada,
data_saida);
```

Além disso, na correção para a fase final também foi criado um índice composto (pelos campos serem usados em conjunto na consulta e serem atributos candidatos a serem

usados em índices por conta de serem utilizados na cláusula `WHERE` para filtragem) para os campos categoria e capacidade em quarto por conta desses campos serem usados em conjunto na consulta e serem atributos candidatos à índices por serem utilizados na cláusula `WHERE` para filtragem). Por isso esse índice passou a ser utilizado no plano três na filtragem dessas características.

Também foi criado um índice em cima do campo estado na tabela hotel para verificar se auxiliaria a busca por hotéis localizados em determinado estado, entretanto esse índice não é utilizado pelo otimizador provavelmente porque o otimizador considerou o gasto para acessar o índice e buscar na tabela mais caro do que a varredura sequencial, já que a tabela é relativamente pequena (cem hotéis).

Segue a criação dos dois índices explicados acima:

```
CREATE INDEX index_estado on hotel(estado);

CREATE INDEX index_caracteristicas_quarto on quarto (categoria, capacidade);
```

Consulta modificada

```
SELECT numero, t1.id_hotel, nome_hotel, estrelas, endereco, preco

FROM (
    SELECT ho.id_hotel, numero, nome_hotel, estrelas, endereco, preco
    FROM hotel ho
    NATURAL JOIN quarto q
    WHERE estado = 'BA'
    AND categoria = 'Suite dupla'
    AND capacidade = 4
) t1
WHERE NOT EXISTS (
    SELECT 1
    FROM reserva r
    WHERE r.id_hotel = t1.id_hotel
    AND r.numero_quarto = t1.numero
    AND r.data_entrada <= '2023-02-02' AND r.data_saida > '2023-01-05'
);
```

Figura 1: Plano de execução da consulta 1 inicial

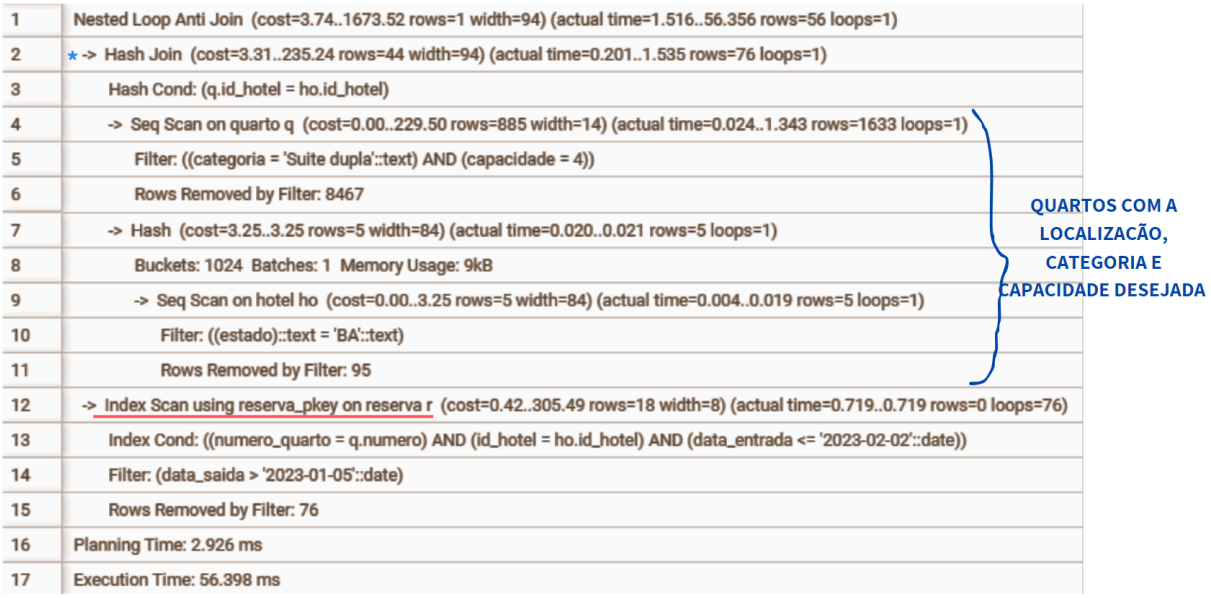

Figura 2: Plano de execução da consulta 1 otimizada até a fase intermediária

1	Nested Loop Anti Join (cost=17.09..154.94 rows=1 width=94) (actual time=2.142..5.527 rows=56 loops=1)
2	-> Hash Join (cost=16.67..110.37 rows=44 width=94) (actual time=0.542..1.722 rows=76 loops=1)
3	Hash Cond: (q.id_hotel = ho.id_hotel)
4	-> Bitmap Heap Scan on quarto q (cost=13.36..104.63 rows=885 width=14) (actual time=0.204..1.404 rows=1633 loops=1)
5	Recheck Cond: ((categoria = 'Suite dupla'::text) AND (capacidade = 4))
6	Heap Blocks: exact=78
7	-> Bitmap Index Scan on index_caracteristicas_quarto (cost=0.00..13.13 rows=885 width=0) (actual time=0.152..0.152 rows=1633 loops=1)
8	Index Cond: ((categoria = 'Suite dupla'::text) AND (capacidade = 4))
9	-> Hash (cost=3.25..3.25 rows=5 width=84) (actual time=0.063..0.063 rows=5 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on hotel ho (cost=0.00..3.25 rows=5 width=84) (actual time=0.016..0.055 rows=5 loops=1)
12	Filter: ((estado)::text = 'BA'::text)
13	Rows Removed by Filter: 95
14	-> Index Only Scan using index_reserva on reserva r (cost=0.42..5.74 rows=18 width=8) (actual time=0.049..0.049 rows=0 loops=76)
15	Index Cond: ((numero_quarto = q.numero) AND (id_hotel = ho.id_hotel) AND (data_entrada <= '2023-02-02'::date) AND (data_saida > '2023-01-05'::date))
16	Heap Fetches: 0
17	Planning Time: 10.989 ms
18	Execution Time: 5.667 ms

Figura 3: Plano de execução da consulta 1 otimizada na fase final

Analisando os planos de consultas

É utilizada uma varredura sequencial em reserva para filtrar de acordo com o `OVERLAPS` no plano um, enquanto nos outros dois planos é utilizando operadores de comparação ao invés do `OVERLAPS`, fazendo com que índices passem a ser usados, no plano dois o índice de chave primária de reserva e no plano três o índice composto `index_reserva` que foi criado.

Tanto no plano um, figura 1, quanto no plano dois, figura 2, é utilizada uma varredura sequencial para filtrar o estado da tabela hotel (linha 14 na figura 1, linha 9 na figura 2), a capacidade e a categoria da tabela quarto (linha 17 na figura 1, linha 4 na figura 2). Já no plano três, figura 3, os campos categoria e capacidade são filtrados através do índice composto `index_caracteristicas_quarto` (linha 7). No plano um a junção das tabelas `quarto` e `hotel` é feita por `Nested Loop` (linha 1), enquanto no plano dois e três é usado `Hash Join` (linha 2 das figuras 2 e 3), sendo mais otimizado porque enquanto no `Nested Join` se compara uma linha de uma tabela com a outra tabela toda, por meio do `Hash Join` você compara ela apenas com uma partição da tabela, por conta da ordenação por meio do `Hash` que é realizada nesse processo.

No plano três, figura 3, o `Bitmap Index Scan on index_caracteristicas_quartos` (linha 7) é usado para achar os endereços das tuplas que satisfazem o filtro sobre os campos categoria e capacidade. Após isso, o `Bitmap Heap Scan on quarto q` (linha 4) que efetivamente seleciona as tuplas pelo endereço achado pelo passo anterior.

É perceptível, tanto pelo código quanto pelo plano de consulta, a complexidade do primeiro plano, figura 1, em relação aos outros, principalmente ao analisarmos a quantidade de junções (linhas 1, 4 e 11), subconsultas (linhas 10, 20, 21, 24 e 26) e varreduras sequenciais que a consulta um exige (linhas 14, 17, 25, 27, 30 e 31), além da junção por intersecção feita pelo `HashSetOp Intersect` (operações custosas, linha 8). Enquanto na versão otimizada, figura 3, há bem menos aninhamentos e junções, além de aproveitar índices para não fazer varreduras sequenciais (linhas 7, 14).

	Consulta inicial	Consulta otimizada	Diferença (%)
Tempo de Execução	347.976ms	5.667ms	98,3%

Tabela 2: Comparação entre a consulta 1 e sua otimização

Processador	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
RAM	8,00 GB (utilizável: 7,73 GB)
Memória Persistente	HDD

Tabela 3: Especificações Máquina utilizada na consulta 1.

CONSULTA 2

A consulta dois realiza a projeção de dados derivados do produto cartesiano entre as as tabelas hospede e reserva com as condições de seleções no atributo nome do hóspede, modo de pagamento da reserva e a partir de que ano vai-se analisar os dados. Desse modo, os recursos utilizados para sua otimização foram, a princípio, realizar uma junção natural em vez do produto, porque o produto entre tabelas é custoso e como as as tabelas possuem o atributo com mesmo nome '`cpf_hospede`' que podem ser usados numa junção, a natural é adequada.

Além disso, trocar a cláusula `ILIKE` para `LIKE` também foi uma escolha em busca da melhor eficiência, uma vez que o `ILIKE` é case-insensitive exige-se mais computação, o que aumenta o tempo de execução da consulta, por isso, a instrução SQL da consulta foi modificada. Como o nome dos hóspedes segue o padrão nome e um sobrenome, ambos começando com letra maiúscula, pode-se utilizar o `LIKE` na consulta, considerando que o

usuário conhece o padrão. Desse modo, a consulta tem seu tempo de execução diminuído, se beneficiando da eficiência do `LIKE`.

Nesse sentido, com essa modificação na consulta um outro recurso utilizado para sua otimização foi a criação de índices. Assim, foram criados três índices simples, um sobre o atributo `modo_pagamento` de `reserva` do tipo *BTree* para melhorar a seleção do modo de pagamento escolhido pelo usuário, outro para o `ano_entrada` da mesma tabela também do tipo *BTree* a fim de aprimorar a busca sobre a condição. Nesse caso do índice `ano_entrada_ind` ele é utilizado apenas quando o número de tuplas é reduzido, o que depende do valor passado como parâmetro em `ano_entrada`.

Ademais, um sobre o `nome` da tabela `hospede` do tipo *GIN* pois o usuário passa como parâmetro uma substring com o sobrenome que procura. Para criar o índice `gnome_hospede_ind` foi preciso instalar a extensão *pg_trgm* , pois ela fornece recursos para a comparação de strings com base em trigramas, que são grupos de três caracteres consecutivos.

Portanto, com esses recursos em conjunto houve a otimização de 70,14% do tempo de execução do plano de consulta e também houve modificações sobre quais estratégias o otimizador decidiu seguir.

Para casos de teste com valores de entrada diferentes o plano de consulta otimizada não se altera quando os valores de nome e modo pagamento são modificados, entretanto, ao mudar o ano de entrada pode ocorrer alterações. Isso acontece pois no banco de dados há menos reservas entre os anos 2029 e 2027 do que entre 2000 e 2029, assim, o otimizador faz escolhas diferentes sobre qual índice usar. Então, para consultas com ano entrada até 2026 ele opta pelo índice `btpagamento`, já para entradas com ano a partir de 2027 o otimizador utiliza o índice `ano_entrada_ind`.

Consulta modificada

```
select ho.nome, ho.endereco, r.id_hotel, r.data_entrada, r.numero_quarto,
r.data_saida
from hospede ho natural join reserva r
where ho.nome LIKE 'X'
and r.modo_pagamento = 'Y'
and r.ano_entrada >= Z;
```

Índice criado para consulta

```
create index btpagamento on reserva(modo_pagamento)
create index ano_entrada_ind on reserva using btree (ano_entrada)
CREATE INDEX gnome_hospede_ind ON hospede USING gin (nome gin_trgm_ops)
```

	QUERY PLAN text
1	Gather (cost=10767.42..28580.20 rows=18 width=80) (actual time=484.200..577.018 rows=2445 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=9767.42..27578.40 rows=8 width=80) (actual time=437.534..519.512 rows=815 loops=3)
5	Hash Cond: ((r.cpf_hospede)::text = (ho.cpf_hospede)::text)
6	-> Parallel Seq Scan on reserva r (cost=0.00..17614.00 rows=75042 width=24) (actual time=0.010..70.568 rows=59029 loops=3)
7	Filter: ((ano_entrada >= 2014) AND (modo_pagamento = 'Debito'::text))
8	Rows Removed by Filter: 274304
9	-> Parallel Hash (cost=9767.17..9767.17 rows=20 width=80) (actual time=437.175..437.176 rows=2338 loops=3)
10	Buckets: 8192 (originally 1024) Batches: 1 (originally 1) Memory Usage: 984kB
11	-> Parallel Seq Scan on hospede ho (cost=0.00..9767.17 rows=20 width=80) (actual time=0.503..370.320 rows=2338 loops=3)
12	Filter: (nome ~~* '%Almeida'::text)
13	Rows Removed by Filter: 164328
14	Planning Time: 1.069 ms
15	Execution Time: 577.271 ms

Figura 4 - Plano de execução da consulta 2 inicial

	QUERY PLAN text
1	Gather (cost=12474.47..26453.00 rows=3547 width=80) (actual time=81.103..172.090 rows=2445 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=11474.47..25098.30 rows=1478 width=80) (actual time=34.853..116.288 rows=815 loops=3)
5	Hash Cond: ((r.cpf_hospede)::text = (ho.cpf_hospede)::text)
6	-> Parallel Bitmap Heap Scan on reserva r (cost=3655.97..17086.01 rows=73822 width=24) (actual time=13.336..83.835 rows=59029 loops=3)
7	Recheck Cond: (modo_pagamento = 'Debito'::text)
8	Filter: (ano_entrada >= 2014)
9	Rows Removed by Filter: 51682
10	Heap Blocks: exact=3543
11	-> Bitmap Index Scan on btpagamento (cost=0.00..3611.68 rows=330567 width=0) (actual time=14.868..14.877 rows=332133 loops=1)
12	Index Cond: (modo_pagamento = 'Debito'::text)
13	-> Parallel Hash (cost=7766.36..7766.36 rows=4171 width=80) (actual time=21.253..21.254 rows=2338 loops=3)
14	Buckets: 16384 Batches: 1 Memory Usage: 960kB
15	-> Parallel Bitmap Heap Scan on hospede ho (cost=177.58..7766.36 rows=4171 width=80) (actual time=4.924..59.005 rows=7015 loops=1)
16	Recheck Cond: (nome ~~ '%Almeida'::text)
17	Heap Blocks: exact=4510
18	-> Bitmap Index Scan on gnome_hospede_ind (cost=0.00..175.08 rows=10010 width=0) (actual time=4.373..4.373 rows=7015 loops=1)
19	Index Cond: (nome ~~ '%Almeida'::text)
20	Planning Time: 2.733 ms
21	Execution Time: 172.413 ms

Figura 5 - Plano de execução da consulta 2 otimizada com índice `btpagamento`

1	Hash Join (cost=9093.12..22081.45 rows=677 width=80) (actual time=58.441..131.772 rows=453 loops=1)
2	Hash Cond: ((r.cpf_hospede)::text = (ho.cpf_hospede)::text)
3	-> Bitmap Heap Scan on reserva r (cost=1128.64..14028.14 rows=33839 width=24) (actual time=7.541..70.036 rows=33039 loops=1)
4	Recheck Cond: (ano_entrada >= 2027)
5	Filter: (modo_pagamento = 'Debito'::text)
6	Rows Removed by Filter: 66933
7	Heap Blocks: exact=11363
8	-> Bitmap Index Scan on ano_entrada_ind (cost=0.00..1120.18 rows=102367 width=0) (actual time=5.758..5.759 rows=99972 loops=1)
9	Index Cond: (ano_entrada >= 2027)
10	-> Hash (cost=7839.35..7839.35 rows=10010 width=80) (actual time=50.837..50.841 rows=7015 loops=1)
11	Buckets: 16384 Batches: 1 Memory Usage: 909kB
12	-> Bitmap Heap Scan on hospede ho (cost=177.58..7839.35 rows=10010 width=80) (actual time=4.066..46.914 rows=7015 loops=1)
13	Recheck Cond: (nome ~~ '%Almeida'::text)
14	Heap Blocks: exact=4510
15	-> Bitmap Index Scan on gnome_hospede_ind (cost=0.00..175.08 rows=10010 width=0) (actual time=3.360..3.360 rows=7015 loops=1)
16	Index Cond: (nome ~~ '%Almeida'::text)
17	Planning Time: 0.471 ms
18	Execution Time: 132.485 ms

Figura 6 - Plano de execução da consulta 2 otimizada com o índice `ano_entrada_ind`

Analisando os planos de consultas

O plano de consulta inicial, figura 4, começa fazendo uma varredura sequencial paralela (*Parallel Seq Scan* | linha 11) na tabela `hospede` e aplica um filtro por meio da condição passada para o nome do hóspede para selecionar aqueles que se adequam, tendo como retorno 2338 tuplas. Em seguida, ocorre uma operação paralela de *Hash (Parallel Hash* | linha 9), que constrói uma tabela *Hash* com base nos resultados da varredura anterior.

Nesse sentido, há uma varredura sequencial paralela (*Parallel Seq Scan* | linha 6) na tabela `reserva` com o filtro sobre os registros '`>= ano_entrada`' e '`= modo_pagamento`' para selecionar os que estão na condição passada. Então, para a etapa de junção (linha 5) tem-se um *Hash Cond* entre `reserva` e `hospede` utilizando uma igualdade entre os campos de mesmo nome '`cpf_hospede`', assim, os resultados das varreduras em ambas tabelas são combinados e são unidos usando a condição de *Hash* com o *Parallel Hash Join* (linha 4).

Portanto, essa foi a estratégia utilizada pelo otimizador para executar a consulta inicial, junto ao Gather que coleta os resultados dos trabalhadores e retorna o resultado final obtido pela consulta. No entanto, observa-se que com a otimização o plano de execução foi

modificado, uma vez que o otimizador seguiu outros caminhos a fim de obter o melhor custo e eficiência de acordo com as estatísticas.

Dessa forma, o plano da consulta otimizada com o índice `btpagamento`, Figura 5, inicia realizando uma busca por índice (Bitmap Index Scan | linha 18) para encontrar os endereços das tuplas correspondentes na tabela `hospede` utilizando o índice `gnome_hospede_ind` com a condição no atributo `nome` passada pelo usuário. Depois disso, o resultado dessa etapa é utilizado para verificar novamente (linha 16) as tuplas que atendem à condição de busca com o *Parallel Bitmap Heap Scan* na linha 15 sobre a tabela `hospede`. Desse modo, o *Parallel Hash* na linha 13 cria uma tabela Hash com base nos resultados obtidos anteriormente, assim, agrupa e organiza os dados em lotes a fim de otimizar o processo subsequente.

Ademais, na linha 11 tem-se um *Bitmap Index Scan* na tabela `reserva` por meio do índice `btpagamento`, com o objetivo de buscar as tuplas correspondentes à condição passada sobre o `modo_pagamento`. Então, há um *Parallel Bitmap Heap Scan* (linha 6) para filtrar as tuplas encontradas pelo índice e que possuem o `ano_entrada` equivalente ao parâmetro da consulta, assim, as que não se encaixam nas condições são removidas e ocorre o *Recheck* sobre o `modo_pagamento` também.

Por fim, como na consulta inicial, os resultados das varreduras em ambas tabelas são combinados e são unidos usando a condição de *Hash* com o *Parallel Hash Join* (linha 4) e o *Gather* reúne o resultado da execução para gerar a saída. Portanto, o otimizador utilizou dos recursos para melhorar o desempenho da consulta e como consequência teve seu tempo de execução diminuído, utilizando sua nova estratégia.

Já o plano da consulta otimizada com o índice `ano_entrada_ind`, figura 6, segue uma estratégia semelhante, porém sem dividir o trabalho em *workers* e utiliza o índice sobre o ano inserido para recuperar as tuplas, uma vez que o otimizador considera esse o melhor plano para o caso em que o número de tuplas retornadas é menor.

	Consulta inicial	Consulta otimizada	Diferença (%)
Tempo de Execução	577,271 ms	172,413 ms	70,14

Tabela 4: Comparação entre a consulta 2 e sua otimização

Processador	Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 1.69 GHz
RAM	16,0 GB RAM
Memória Persistente	HDD

Tabela 5: Especificações Máquina utilizada na consulta 2.

CONSULTA 3

Analisando os planos de consultas

Na consulta original, figura 7, foram feitas operações de *Seq Scan* e *Index Scan* com as chaves primárias. Além disso, a operação de busca do mês e ano de saída é custosa devido ao uso do `EXTRACT()` (linha 16) que é utilizado para retirar as informações de `data_saida`. Outra operação que a consulta original faz uso é o *GroupAggregate* (linha 1) utilizado para consultas que envolvem agregações com agrupamento para grupos de linhas com base em valores de uma ou mais colunas, realidade desta consulta, em que processa os grupos de linhas um a um. O *Nested Loop* (linha 3 e 8) é utilizado para fazer a junção por laço entre tabelas com as suas linhas, mas pode ser uma operação custosa, já que pode resultar em um alto número de operações de acesso a disco.

Na consulta modificada, figura 8, é possível ver uma redução alta no tempo de execução devido ao uso de *hash* para juntar as tabelas: `quarto, hotel e reserva`. Além disso, na operação de busca do mês e do ano de saída utiliza-se o índice `ind_saida` que foi criado com o intuito de facilitar a busca pelas variáveis `mes_saida e ano_saida` o qual ajuda a melhorar a eficácia da pesquisa indicado pela operação "*Bitmap Index Scan ON ind_saida*" (linha 11) que indica o acesso ao índice que fornece os endereços dos dados para o Bitmap Heap Scan que utiliza essa informação para verificar múltiplas condições de pesquisa de uma consulta e faz uma varredura. Outro fator que ajudou a diminuir o custo da operação foi a utilização das variáveis `mes_saida e ano_saida`

(linha 9), pois, com elas não foi necessário utilizar o `EXTRACT()` que é uma operação muito custosa.

A consulta modificada ainda utiliza o *HashAggregate* (linha 1) que usa uma tabela de *Hash* em memória para agrupar as linhas de acordo com as colunas especificadas, e como operação de junção utiliza o *Hash Join* (linha 4 e 6) que usa uma tabela de *Hash* em memória para criar uma estrutura de dados que mapeia os valores de uma coluna e de uma tabela para as linhas correspondentes da outra tabela. Portanto, a otimização também decorre da presença do *Hash Join* onde antes haviam loops aninhados (Nested Loop), porque a troca leva a um menor número de operações de entrada e saída de disco.

Consulta modificada final

```
SELECT h.id_hotel, h.nome_hotel, h.estrelas, h.endereco,
SUM(q.preco    *    (DATE_PART('day',    AGE(DATE_TRUNC('day',    r.data_saida),
DATE_TRUNC('day', r.data_entrada))))) AS receita_mensal
FROM hotel h natural join quarto q , reserva r
WHERE
  q.numero = r.numero_quarto
  AND q.id_hotel = r.id_hotel
  AND r.mes_saida = 02
  AND r.ano_saida = 2022
GROUP BY h.id_hotel;
```

Índices criado para consulta

```
CREATE INDEX ind_saida ON reserva(mes_saida, ano_saida);
CREATE INDEX ind_mes_saida ON reserva(mes_saida);
CREATE INDEX ind_ano_saida ON reserva(ano_saida);
```

O plano de consulta não utilizou os índices criados de forma separada, `ind_mes_saida` e `ind_ano_saida`. Além disso, também foi testado meses e anos diferentes como valores passados como parâmetros, porém, o plano se manteve o mesmo.

	QUERY PLAN text
1	GroupAggregate (cost=0.58..10011.77 rows=25 width=92) (actual time=5688.373..5863.419 rows=100 loops=1)
2	Group Key: h.id_hotel
3	-> Nested Loop (cost=0.58..10010.89 rows=25 width=98) (actual time=1534.649..5832.021 rows=2662 loops=1)
4	Join Filter: (q.id_hotel = h.id_hotel)
5	Rows Removed by Join Filter: 263538
6	-> Index Scan using hotel_pkey on hotel h (cost=0.14..17.61 rows=100 width=84) (actual time=0.049..1.819 rows=100 loops=1)
7	-> Materialize (cost=0.43..9955.84 rows=25 width=22) (actual time=0.063..57.239 rows=2662 loops=100)
8	-> Nested Loop (cost=0.43..9955.72 rows=25 width=22) (actual time=6.179..5665.211 rows=2662 loops=1)
9	-> Seq Scan on quarto q (cost=0.00..179.00 rows=10100 width=14) (actual time=0.094..21.203 rows=10100 loops=1)
10	-> Memoize (cost=0.43..9.43 rows=1 width=16) (actual time=0.519..0.553 rows=0 loops=10100)
11	Cache Key: q.id_hotel, q.numero
12	Cache Mode: logical
13	Hits: 0 Misses: 10100 Evictions: 0 Overflows: 0 Memory Usage: 835kB
14	-> Index Scan using reserva_pkey on reserva r (cost=0.42..9.42 rows=1 width=16) (actual time=0.509..0.543 rows=0 loops=10...
15	Index Cond: ((numero_quarto = q.numero) AND (id_hotel = q.id_hotel))
16	Filter: ((EXTRACT(month FROM data_saida) = '2'::numeric) AND (EXTRACT(year FROM data_saida) = '2022'::numeric))
17	Rows Removed by Filter: 99
18	Planning Time: 2.553 ms
19	Execution Time: 5866.260 ms

Figura 7 - Plano de execução da consulta 3 inicial

	QUERY PLAN text
1	HashAggregate (cost=6653.24..6654.24 rows=100 width=92) (actual time=79.590..79.664 rows=100 loops=1)
2	Group Key: h.id_hotel
3	Batches: 1 Memory Usage: 48kB
4	-> Hash Join (cost=373.78..6588.34 rows=2596 width=98) (actual time=17.949..45.759 rows=2662 loops=1)
5	Hash Cond: ((h.id_hotel = q.id_hotel) AND (r.numero_quarto = q.numero))
6	-> Hash Join (cost=43.28..6244.21 rows=2596 width=100) (actual time=2.881..25.566 rows=2662 loops=1)
7	Hash Cond: (r.id_hotel = h.id_hotel)
8	-> Bitmap Heap Scan on reserva r (cost=39.03..6232.86 rows=2596 width=16) (actual time=2.606..16.778 rows=2662 lo...
9	Recheck Cond: ((mes_saida = 2) AND (ano_saida = 2022))
10	Heap Blocks: exact=2372
11	-> Bitmap Index Scan on ind_saida (cost=0.00..38.38 rows=2596 width=0) (actual time=1.676..1.676 rows=2662 loop...
12	Index Cond: ((mes_saida = 2) AND (ano_saida = 2022))
13	-> Hash (cost=3.00..3.00 rows=100 width=84) (actual time=0.218..0.220 rows=100 loops=1)
14	Buckets: 1024 Batches: 1 Memory Usage: 20kB
15	-> Seq Scan on hotel h (cost=0.00..3.00 rows=100 width=84) (actual time=0.049..0.108 rows=100 loops=1)
16	-> Hash (cost=179.00..179.00 rows=10100 width=14) (actual time=14.993..14.994 rows=10100 loops=1)
17	Buckets: 16384 Batches: 1 Memory Usage: 602kB
18	-> Seq Scan on quarto q (cost=0.00..179.00 rows=10100 width=14) (actual time=0.077..6.128 rows=10100 loops=1)
19	Planning Time: 2.431 ms
20	Execution Time: 80.401 ms

Figura 8 - Plano de execução da consulta 3 otimizada

	Consulta inicial	Consulta otimizada	Diferença (%)
Tempo de Execução	5866.260ms	80.401ms	98,7

Tabela 6: Comparação entre a consulta 3 e sua otimização

Processador	Intel Atom Quad-Core
RAM	4,0 GB RAM
Memória Persistente	HDD

Tabela 7: Especificações Máquina utilizada na consulta 3.

5. PROGRAMAÇÃO COM BANCO DE DADOS

Foram construídas três *Stored Procedures* referentes às consultas, cada uma parametrizada com os filtros da consulta correspondente. As funções foram criadas utilizando a linguagem pgsql e serão descritas a seguir.

CONSULTA 1

A função a seguir é responsável por realizar a consulta 1 de forma dinâmica e segura, assim, como argumentos recebe um estado em texto, uma categoria em texto, uma capacidade em inteiro e duas datas como dates, representando a data de entrada e a data de saída, respectivamente. Desse modo, a função retorna uma tabela contendo as informações dos quartos e seus hotéis que estão disponíveis entre as datas especificadas, retornando o número do quarto, o identificador, o nome, o endereço, e as estrelas do hotel e o preço daquele quarto.

```
CREATE OR REPLACE FUNCTION consulta_1 (estado text, categoria text, capacidade
int, data_entrada date, data_saida date)
RETURNS TABLE ( numero INT, id_hotel INT, nome_hotel TEXT, estrelas INT,
endereco TEXT, preco DECIMAL) AS $$
BEGIN
RETURN QUERY EXECUTE '
    SELECT numero, t1.id_hotel, nome_hotel, estrelas, endereco, preco
    FROM (
        SELECT ho.id_hotel, numero, nome_hotel, estrelas, endereco, preco
```

```
        FROM hotel ho
        NATURAL JOIN quarto q
        WHERE estado = $1
        AND categoria = $2
        AND capacidade = $3
    ) t1
    WHERE NOT EXISTS (
        SELECT 1
        FROM reserva r
        WHERE r.id_hotel = t1.id_hotel
        AND r.numero_quarto = t1.numero
        AND r.data_entrada <= $5 AND r.data_saida > $4
    );'
    USING estado, categoria, capacidade, data_entrada, data_saida;
END
$$ LANGUAGE plpgsql;
```

A fim de prevenir possíveis SQL *Injections,* a providência foi utilizar passar os parâmetros usando $ (placeholder) com o uso do EXECUTE … USING, pois com isso os parâmetros especificados no USING são passados de corretamente para o placeholder, de forma isolada e literal, impedindo que o programa trate inputs como códigos SQL.

CONSULTA 2

A função a seguir é responsável por realizar a consulta 2 de forma dinâmica e segura, assim, como argumentos recebe um sobrenome em texto, um modo de pagamento em texto e um ano em inteiro. Desse modo, a função retorna uma tabela contendo as informações do hóspede (nome e endereço), o identificador do hotel e as datas de entrada e saída da reserva.

```
CREATE OR REPLACE FUNCTION consulta_2 (sobrenome text, modo_pagamento text,
ano_entrada int)
    RETURNS TABLE (nome text, endereco text, id_hotel int, data_entrada date,
data_saida date) AS $$
    BEGIN
     RETURN QUERY EXECUTE 'SELECT ho.nome, ho.endereco, r.id_hotel,
r.numero_quarto, r.data_entrada, r.data_saida
        FROM hospede ho
        NATURAL JOIN reserva r
        WHERE ho.nome LIKE $1
        AND r.modo_pagamento = $2
        AND r.ano_entrada >= $3'
```

```
        USING '%' || sobrenome, modo_pagamento, ano_entrada;
    END;
$$ LANGUAGE plpgsql;
```

Para prevenir SQL *Injections* a providência tomada foi utilizar o $ para cada variável passadas como parâmetro, porque esse recurso é utilizado quando o valor de um dado é passado como parâmetro e verifica se o tipo do valor passado é correspondente ao definido. Desse modo, os valores são tratados de forma segura porque utiliza-se o `USING EXECUTE`, que lida corretamente com parâmetros e evita injeção de SQL. Portanto, impede que comandos SQL sejam interpretados como comandos, assim, no caso das variáveis do tipo texto os comandos serão lidos como parte do texto, já para o tipo inteiro ocorre um erro de entrada inválida por sintaxe.

CONSULTA 3

A função a seguir é responsável por realizar a consulta 3 de forma dinâmica e segura, recebendo os argumentos: mês e ano. Desse modo, a função retorna uma tabela contendo as informações `id_hotel`, `nome_hotel`, `estrelas`, `endereço` e `receita_mensal`, a qual é calculada com o mês e ano passados como argumento.

```
CREATE OR REPLACE FUNCTION consulta_3(mes_param INT, ano_param INT)
      RETURNS TABLE (
          id_hotel INT,
          nome_hotel TEXT,
          estrelas INT,
          endereco TEXT,
          receita_mensal DOUBLE PRECISION
      ) AS $$
      BEGIN
          RETURN QUERY
          EXECUTE '
          SELECT h.id_hotel, h.nome_hotel, h.estrelas, h.endereco,
          SUM(q.preco * (DATE_PART(''day'', AGE(DATE_TRUNC(''day'',
r.data_saida), DATE_TRUNC(''day'', r.data_entrada))))) AS receita_mensal
          FROM hotel h
          NATURAL JOIN quarto q
          JOIN reserva r ON q.numero = r.numero_quarto AND q.id_hotel =
r.id_hotel
```

```
            WHERE r.mes_saida = $1 AND r.ano_saida = $2
            GROUP BY h.id_hotel, h.nome_hotel, h.estrelas, h.endereco'
            USING mes_param, ano_param;
        END;
$$ LANGUAGE plpgsql;
```

A fim de prevenir possíveis SQL *Injections,* a providência foi utilizar o `$` para as variáveis passadas como parâmetro, uma vez que esse recurso é utilizado quando o valor de um dado é passado como parâmetro e verifica se o tipo do valor passado é correto. Desse modo, os valores são tratados de forma segura porque utiliza-se o `USING EXECUTE`, que lida corretamente com parâmetros e evita a injeção de SQL. Portanto, impede que comandos SQL sejam interpretados como comandos, assim, no caso do tipo inteiro ocorre um erro de entrada inválida por sintaxe.

6. CONTROLE DE ACESSO DE USUÁRIOS

Para definição do controle de acesso de usuários criou-se cinco *Roles* com as permissões descritas na tabela 8. Nesse sentido, o `administrador` é o responsável por administrar o sistema, assim, este papel é o dono do banco de dados, tem acesso a todas as operações do sistema e é responsável por conceder as permissões aos outros usuários.

O `gerente_rede` tem o papel de gerenciar a rede, isso significa que informações sobre os hotéis e detalhes sobre os quartos em geral ele é quem tem poder de alteração caso haja necessidade, pois está relacionado a rede de hotéis. Já o `gerente_hotel` é gerente do hotel em que trabalha, por isso tem limitações sobre quais atributos dos quartos pode atualizar e não tem permissão de alterar dados na tabela `hotel`. Dessa forma, ambos *roles* tem permissões sobre `reserva` e `hospede`, pois mesmo que diferentes, ainda são gerentes e são quem garante aos recepcionistas a permissão de alterar essas tabelas.

O `recepcionista` tem o papel de atender o cliente, assim, ele precisa ter permissão para visualizar o hotel, os quartos, o hóspede e a reserva para obter as informações dessas tabelas. Além disso, ele precisa da permissão para alterar a tabela `hospede` podendo inserir e atualizar caso tenha algum cliente novo ou que queira atualizar seus dados. E a reserva que necessita das permissões para as operações de atualizar caso algum cliente

resolva mudar sua reserva, de inserir caso algum cliente queira fazer uma reserva naquele momento e de deletar caso alguma reserva seja cancelada.

Assim, o `cliente`, aquele que representa um hóspede cadastrado ou não, os privilégios concedidos são os de leitura sobre os dados dele registrados e as informações permitidas ao público sobre os hotéis e quartos. Para o cliente nos baseamos em sites de reservas online, onde o usuário tem acesso a ler os dados listados na tabela 8. Já para leitura da tabela reserva, ele pode ler informações sobre suas reservas e os atributos disponíveis para leitura não são todos porque não há necessidade do usuário ler as datas de entrada e saída compostas.. Portanto, o arquivo "*usuarios_segurança.sql*" possui essas definições em SQL, utilizado para implementação.

	gerente_hotel	recepcionista	cliente	gerente_rede	cliente_auxiliar	administrador
Hotel	S	S	S(nome,estrelas, endereco)	S,I,U	Sem acesso	S,I,U,D
Quarto	S,U(capacidade, categoria,preco)	S	S(capacidade, categoria, preco)	S,I,U	Sem acesso	S,I,U,D
Hospede	S,I,U,D	S,I,U	S	S,I,U,D	S	S,I,U,D
Reserva	S,I,U,D	S,I,U,D	S(cpf_hospde,numero_quarto,id_hotel,data_entrada,data_saida,modo_pagamento)	S,I,U,D	S	S,I,U,D

Tabela 8: Perfis de Usuários, Papéis e Privilégios.
`(SELECT (S),INSERT (I), DELETE (D) e UPDATE (U))`

Observa-se também que para que o usuário `cliente` tivesse a permissão de ler somente os dados associados ao cpf dele, criamos a função `acesso_hospedes_por_cpf()` para que ele possa consultar a tabela `hospede` para ler suas informações e o `administrador` garante a execução dela ao `cliente`.

Para isso na função ocorre uma troca de papéis, porque o cliente não tem permissão de fazer um `SELECT` em `hospede` ou em `reserva`, assim, o usuário que executa a consulta é o `cliente_auxiliar`(permissão apenas de leitura nas tabelas `hospede` e `reserva`, papel criado para garantir segurança nesse momento de mudança de *roles* e após a realização da consulta ocorre a alteração de usuário novamente para retornar ao `cliente`.

Já para a leitura de das reservas pelo `cliente`, criamos a função `acesso_reservas_por_cpf()`, semelhante a `acesso_hospedes_por_cpf()`, e também a visão `cliente_acesso_reserva`, que executa a função, logo, o administrador do banco permite o usuário cliente fazer o `SELECT` da visão. Percebe-se que o modo de acessar as tabelas `hospede` e `reserva` por meio do `cliente` são semelhantes, mas como foi possível implementar duas soluções funcionais decidimos ter ambas presentes no projeto prático. Portanto, suas diferenças estão no modo de passar o parâmetro as funções, uma vez que ao permitir executar a função, basta executar:

```
select * from acesso_hospedes_por_cpf('<numero_do_cpf>');,
```

porém ao utilizar a visão, o número do CPF precisa ser setado antes de chamar a *view*, assim tem-se o conjunto de comandos:

```
SET cliente.cpf TO '<numero_do_cpf>';
select * from cliente_acesso_reserva;
```

Dessa maneira, a figura 9 mostra como as permissões garantidas no sistema foram distribuídas.

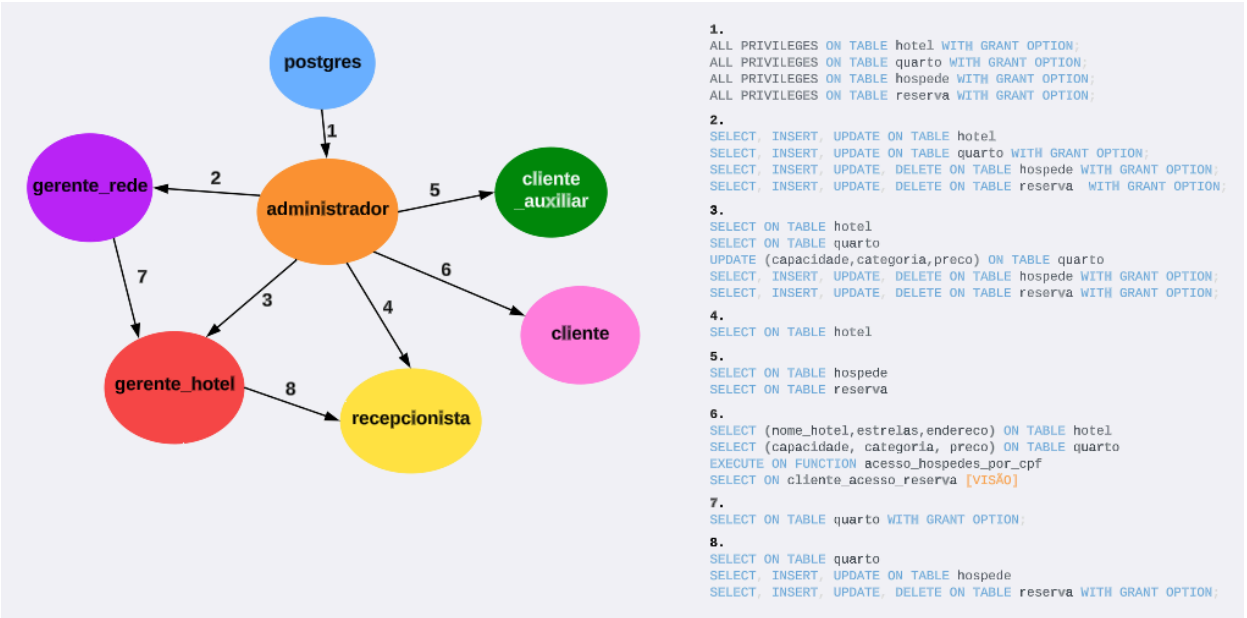

Figura 9: Grafo de concessões

A figura 10, mostra como a aplicação e os *roles* descritos conectam-se com o banco de dados.

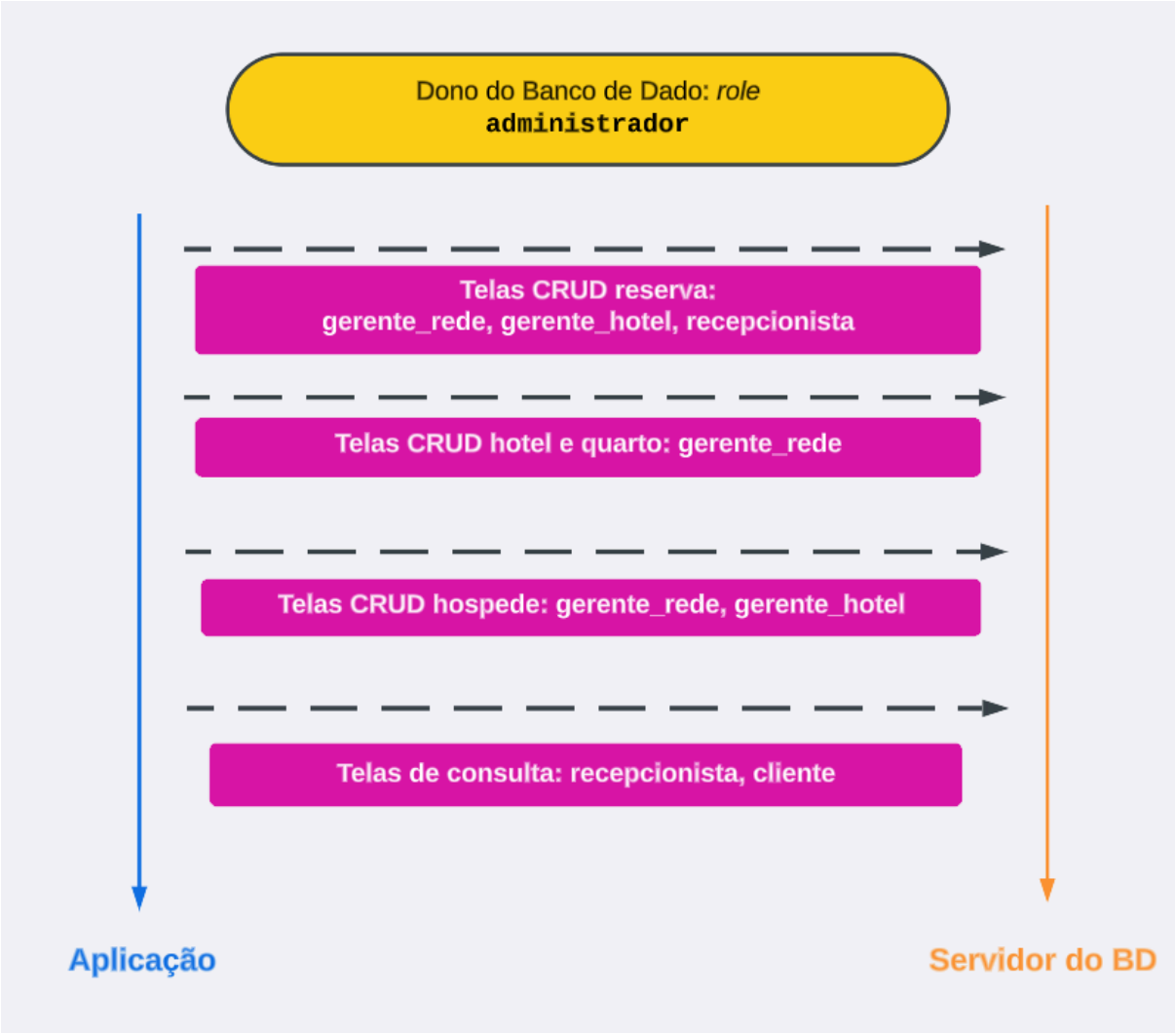

Figura 10: Conexões usando perfis de usuário.

7. CONSIDERAÇÕES FINAIS

Para a execução do projeto aplicamos diversas técnicas e métodos vistos durante as aulas ministradas de Sistemas de Banco de Dados e os conceitos aprendidos na disciplina de Banco de Dados. Então, antes da primeira etapa, definição das consultas, não imaginávamos a complexidade e a necessidade de planejamento prévio e análise posterior sobre uma consulta para seu desenvolvimento e execução.

A princípio, com as consultas definidas e banco de dados criado, foi preciso popular as tabelas. Para isso foram desenvolvidos *scripts* em python, logo, o grupo precisou entender qual seria a melhor forma e caminho a seguir para gerar os dados necessários. Então, escolhemos gerar os dados com bibliotecas e funções python e para inserção correta utilizamos *triggers,* captura e tratamento de exceções.

Desse modo, ao longo das fases seguintes do projeto em conjunto as aulas assistidas, procuramos compreender quais passos seriam necessários para melhor tomada de decisão, uma vez que precisaríamos otimizar as consultas e entender os motivos da otimização. Assim, surgiram as dúvidas iniciais: “Qual será a melhor maneira de otimizar determinada consulta?”, “Criar índices?”, “Porque o otimizador não opta por utilizar o índice criado?”. Nesse sentido, a melhor resposta que encontramos foi testar as opções pensadas, compará-las e tirar conclusões. Além disso, o apoio da realização dos questionários da disciplina junto a documentação também auxiliaram o desenvolvimento do projeto prático.

Já para a fase final, segurança, nos baseamos nas práticas dos portfólios para entender e aplicar os comandos para prevenir as injeções SQL e criar os usuários. Nessa etapa, a parte que gerou dificuldade foi estabelecer quais usuários eram necessários e como manter a segurança do banco ao conceder determinados privilégios. Então, demoramos para encontrar uma alternativa que permitisse o cliente visualizar apenas informações relacionadas ao seu CPF de forma segura, mas desenvolvemos a ideia de criar uma função para executar a consulta, fazer uma mudança de *roles* dentro da função para que a consulta fosse executada com permissão garantida e criamos um cliente auxiliar para fazer essa troca. Além disso, também criamos uma visão para garantir essa consulta, assim, para fazer o `select` na tabela `hospede` o usuário cliente tem `grant` de executar a função e na tabela `reserva` ele tem `grant select` da *view.*

8. FONTES

Booking.com: A maior seleção de hotéis, pousadas e casas de temporada. Disponível em: <https://www.booking.com/index.pt-br.html?aid=397594&label=gog235jc-1FCAEoggI46AdILVgDaCCIAQGYAS24ARfIAQzYAQHoAQH4AQKIAgGoAgO4ApSaz6YGwAIB0gIkZjRiMGFlNmItMzFiMS00NDJjLWIzNDQtYmEzNzI4ZTg0MjIx2AIF4AIB&sid=cca8a4e07722c92a5a901464013bc20a&>. Acesso em: 7 ago. 2023.

Reservar hotéis baratos com a ibis - Todos os nossos hotéis. Disponível em: <https://ibis.accor.com/brasil/index.pt-br.shtml?utm_term=mar&gclid=CjwKCAjw8symBhAqEiwAaTA__B_uS1jnKTqeg6k92vWCf6Bv7CLuIuySdfwZNLJ2ebSOyiJ1LftIOxoCyhIQAvD_BwE&utm_campaign=ppc-ibi-mar-goo-br-pt-pure-mix-sear-&utm_medium=cpc&utm_source=google&utm_content=br-pt-all-all>. Acesso em: 9 ago. 2023.