

UNIVERSIDADE FEDERAL DE SÃO CARLOS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

SISTEMAS DE BANCOS DE DADOS
Profa. Dra. Sahudy Montenegro González

Entrega Final

Descrição do minimundo + MER + Modelo Relacional + script de criação SQL +
consultas + otimização + funções + segurança

Data de entrega: 08/09/2024

GRUPO 2

TEMA - **Empresa de Transportes**

Ana Beatriz Juvencio **801817**

Nícolas Benitiz **813037**

ÍNDICE

1. Consultas	3
2. Projeto Conceitual	4
2.1. Modelo Entidade-Relacionamento	4
3. Projeto Lógico	5
4. Script para criação do Banco	6
5. Perfil dos usuários	8
6. Quantidade de linhas por tabela e distribuição	8
7. Otimização de Consultas	9
7.1. Consulta de busca relativa (Nícolas):	9
7.2. Consulta de busca absoluta (Ana):	12
7.2.1 Testando outros valores:	15
8. Programação Interna do SGBD	16
9. Controle de acesso	18
10. Considerações Finais	20

No centro econômico de Novo Horizonte, uma nova e promissora empresa de transporte, "Viação Estrelar", está prestes a lançar suas operações. Sob a administração do dinâmico empreendedor João Carvalho, que traz uma vasta experiência do setor de logística da capital, a empresa tem como missão revolucionar o transporte, oferecendo uma solução eficaz e confortável para os deslocamentos diários dos cidadãos e visitantes.

"Viação Estrelar" será especializada no gerenciamento de uma frota moderna de veículos e na organização de rotas otimizadas para maximizar a cobertura e minimizar os tempos de viagem. O sistema desenvolvido permitirá aos clientes comprar passagens e selecionar assentos com facilidade através de uma interface amigável.

No sistema de dados da "Viação Estrelar", encontra-se a tabela de Clientes, que armazena informações essenciais como CPF, nome, endereço, telefone e email. Esta tabela é fundamental para garantir um serviço personalizado e eficaz. A tabela de Horário registra os horários de chegada e partida dos veículos, elemento crucial para a organização dos itinerários.

Cada cidade atendida pela empresa é registrada na tabela Cidade, que especifica os pontos de origem e destino para as rotas. A tabela de Rotas lista todas as rotas disponíveis, incluindo detalhes como distância e preço, proporcionando transparência e opções para os planejamentos de viagem dos clientes.

Os detalhes da frota são gerenciados na tabela de Veículos, que inclui informações como modelo, ano, status operacional e capacidade. Essa gestão cuidadosa garante a eficiência da manutenção e operação dos veículos.

1. Consultas

Segue as consultas que cada membro ficou responsável:

- 1) Quais clientes compraram passagens entre os dias X e Y, desde que o nome do cliente comece com certas letras?

```
SELECT Cliente.*
```

```
FROM Cliente
```

```
JOIN Compra ON Cliente.CPF = Compra.CPF
```

```
WHERE Cliente.Nome ILIKE '<letras>%'
```

```
AND data_hora_partida BETWEEN 'X' AND 'Y'
```

ORDER BY nome;

2) Quais pessoas compraram passagem com destino à cidade X no dia Y?

SELECT Cliente.*

FROM Compra

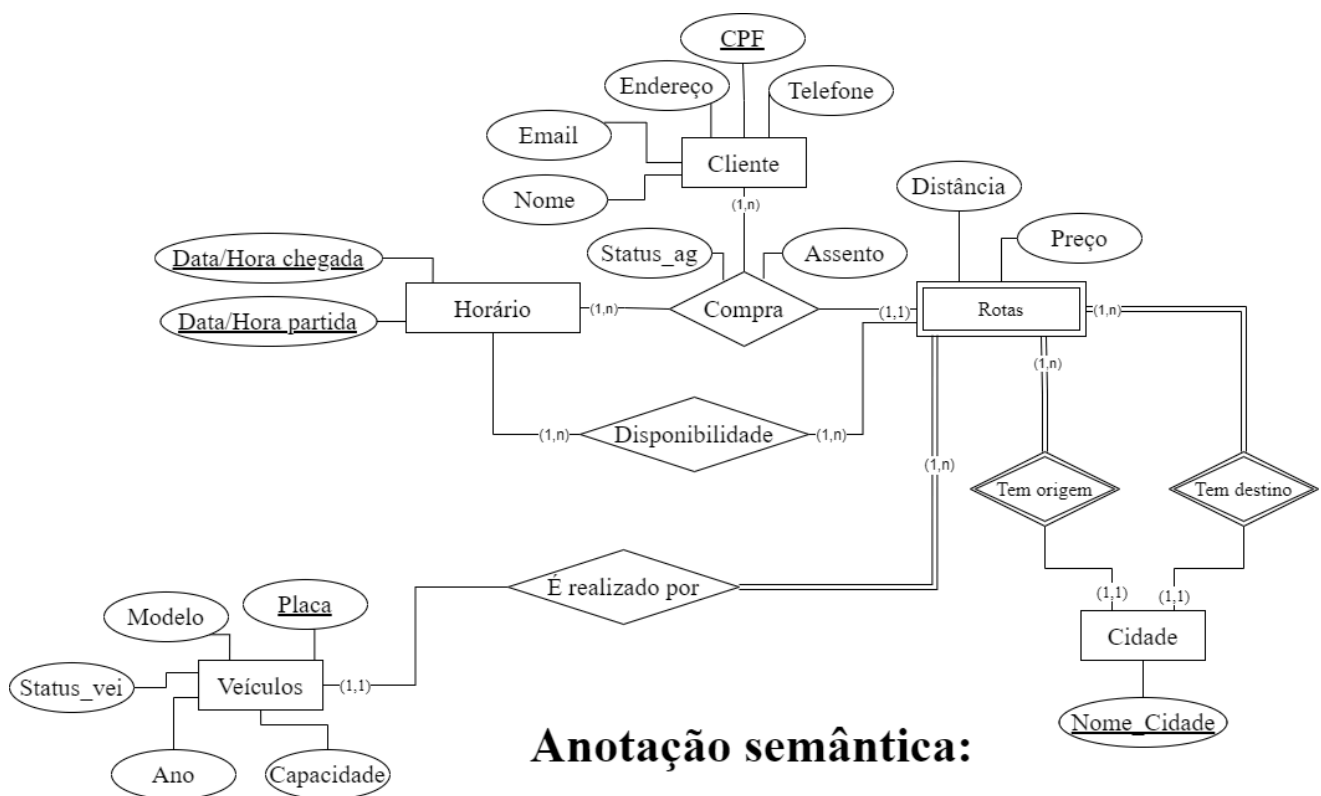
JOIN Cliente ON Cliente.CPF = Compra.CPF

WHERE Compra.Nome_Cidade_Destino = 'X' AND
DATE(Compra.Data_Hora_Chegada) = 'YYYY-MM-DD';

2. Projeto Conceitual

2.1. Modelo Entidade-Relacionamento

O DER (Figura 1) representa graficamente as tipo-entidades que compõem o Empresa de Transporte, assim como seus atributos e os tipo-relacionamentos envolvidos, facilitando a compreensão do minimundo e do banco de dados. Acesse: [Link](#)



Anotação semântica:

Status_ag: (confirmado, cancelado, realizado)

Status_vei: (disponível, em manutenção)

Não é possível cadastrar uma rota+horario no
ternario que nao esteja na disponibilidade

3. Projeto Lógico

Cliente (CPF, Nome, Endereco, Telefone, Email)

Cidade (Nome_Cidade)

Veiculos (Placa, Modelo, Ano, Status_vei, Capacidade)

Rotas (Nome_Cidade_Origem, Nome_Cidade_Destino, Distância, Preço, Placa)

Nome_Cidade_Origem, Nome_Cidade_Destino referencia Cidade

Placa referencia Veiculos

Horario (Data/Hora_Chegada, Data_Hora_Partida)

Disponibilidade (Data/Hora_Chegada, Data_Hora_Partida, Nome_Cidade_Origem, Nome_Cidade_Destino)

data_hora_Chegada ,data_hora_Partida referencia Horario

Nome_Cidade_Origem ,Nome_Cidade_Destino referencia Rotas

Compra (CPF, Data/Hora_Chegada, Data_Hora_Partida, Nome_Cidade_Origem, Nome_Cidade_Destino, Status_ag, Assento)

data_hora_Chegada, data_hora_Partida referencia Horário

Nome_Cidade_Origem, Nome_Cidade_Destino referencia Rotas

CPF referencia Cliente

4. Script para criação do Banco

```
--criando o bd
CREATE DATABASE empresa_de_transportes;

-- Criação da tabela Cliente
CREATE TABLE Cliente (
    CPF VARCHAR(11) PRIMARY KEY,
    Nome VARCHAR(255) NOT NULL,
    Endereco VARCHAR(255) NOT NULL,
    Telefone VARCHAR(20) NOT NULL,
    Email VARCHAR(100) NOT NULL
);

-- Criação da tabela Cidade
CREATE TABLE Cidade (
    Nome_Cidade VARCHAR(100),
    PRIMARY KEY (Nome_Cidade)
);

-- Criação da tabela Horario
CREATE TABLE Horario (
    Data_Hora_Chegada TIMESTAMP WITHOUT TIME ZONE,
    Data_Hora_Partida TIMESTAMP WITHOUT TIME ZONE,
    PRIMARY KEY (Data_Hora_Chegada, Data_Hora_Partida)
);

-- Criação da tabela Veiculos
CREATE TABLE Veiculos (
    Placa VARCHAR(10) PRIMARY KEY,
    Modelo VARCHAR(50) NOT NULL,
    Ano INT NOT NULL,
    Status_vei VARCHAR(20) CHECK (Status_vei IN ('disponivel', 'em
manutencao')) NOT NULL,
    Capacidade INT NOT NULL
);

-- Criação da tabela Rotas
CREATE TABLE Rotas (
    Nome_Cidade_Origem VARCHAR(100),
    Nome_Cidade_Destino VARCHAR(100),
    Distancia DECIMAL(10,2) NOT NULL,
    Preco DECIMAL(10,2) NOT NULL ,
    Placa VARCHAR(10),
```

```

FOREIGN KEY (Placa) REFERENCES Veiculos (Placa),
FOREIGN KEY (Nome_Cidade_Origem) REFERENCES Cidade(Nome_Cidade),
FOREIGN KEY (Nome_Cidade_Destino) REFERENCES Cidade(Nome_Cidade),
PRIMARY KEY (Nome_Cidade_Origem, Nome_Cidade_Destino),
    CHECK (Nome_Cidade_Origem <> Nome_Cidade_Destino) -- Restrição de
verificação
);

-- Criação da tabela Compra
CREATE TABLE Compra (
    CPF VARCHAR(11),
    Data_Hora_Chegada TIMESTAMP WITHOUT TIME ZONE,
    Data_Hora_Partida TIMESTAMP WITHOUT TIME ZONE,
    Nome_Cidade_Origem VARCHAR(100),
    Nome_Cidade_Destino VARCHAR(100),
    Assento INT NOT NULL,
    Status_ag VARCHAR(20) CHECK (Status_ag IN ('confirmado', 'cancelado',
'realizado')) NOT NULL,
    FOREIGN KEY (CPF) REFERENCES Cliente(CPF),
        FOREIGN KEY (Data_Hora_Chegada, Data_Hora_Partida) REFERENCES
Horario(Data_Hora_Chegada, Data_Hora_Partida),
        FOREIGN KEY (Nome_Cidade_Origem, Nome_Cidade_Destino) REFERENCES
Rotas(Nome_Cidade_Origem, Nome_Cidade_Destino),
    PRIMARY KEY (CPF, Data_Hora_Chegada, Data_Hora_Partida)
);

-- Criação da tabela Disponibilidade
CREATE TABLE Disponibilidade (
    Data_Hora_Chegada TIMESTAMP WITHOUT TIME ZONE,
    Data_Hora_Partida TIMESTAMP WITHOUT TIME ZONE,
    Nome_Cidade_Origem VARCHAR(100),
    Nome_Cidade_Destino VARCHAR(100),
        FOREIGN KEY (Data_Hora_Chegada, Data_Hora_Partida) REFERENCES
Horario(Data_Hora_Chegada, Data_Hora_Partida),
        FOREIGN KEY (Nome_Cidade_Origem, Nome_Cidade_Destino) REFERENCES
Rotas(Nome_Cidade_Origem, Nome_Cidade_Destino),
    PRIMARY KEY (Nome_Cidade_Origem, Nome_Cidade_Destino, Data_Hora_Chegada,
Data_Hora_Partida)
);

```

5. Perfil dos usuários

- a. O primeiro tipo de usuário será o cliente, que deseja poder comprar passagens
- b. O segundo será o gerente, que fará o controle da quantidade de passagens que estão sendo compradas para cada destino, horário etc
- c. O terceiro será o gerente, quem ficará responsável por gerenciar a empresa
- d. O quarto será o O DBA (Data Base Administrator) é o profissional responsável pelo funcionamento do banco de dados;
- e. O quinto será o vendedor, responsável por vender as passagens de ônibus
- f. o sexto será o motorista, funcionário responsável por permitir a locomoção dos clientes

6. Quantidade de linhas por tabela e distribuição

Tabela	Quantidade de linhas
Cliente	10.000
Disponibilidade	100.000
Compra	994.584
Veiculos	10.000
Rotas	9.998
Horario	10.000
Cidade	7.171

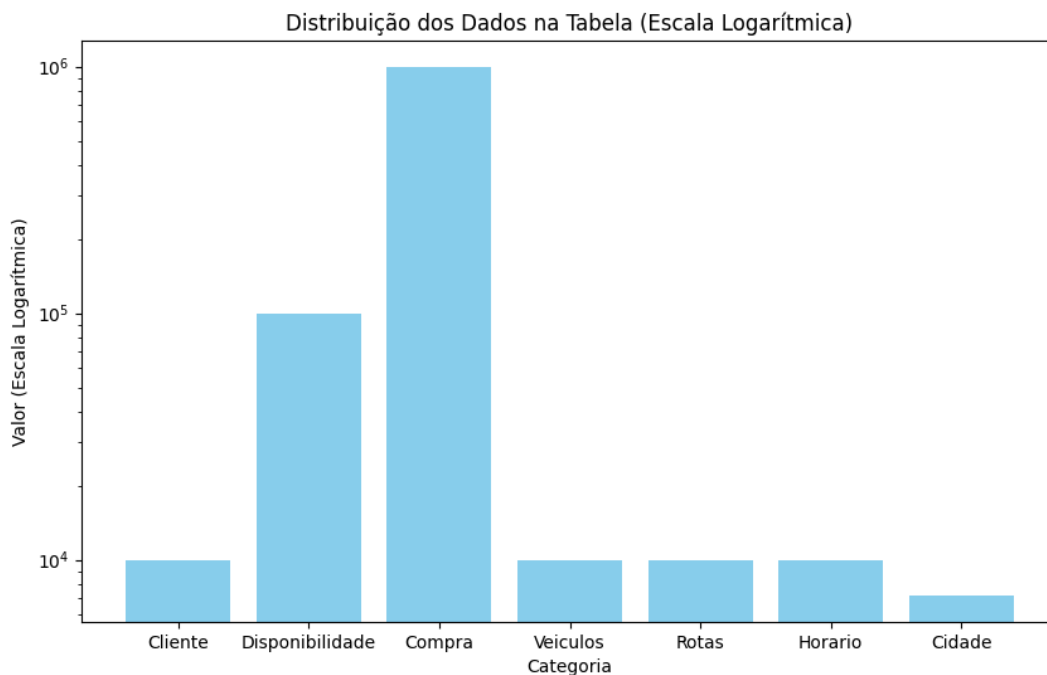


Gráfico 1: Distribuição de dados na tabela em escala logarítmica

7. Otimização de Consultas

7.1. Consulta de busca relativa (Nícolas):

Nesta etapa, a consulta relativa inicial será otimizada utilizando algumas estratégias de otimização de consultas. A versão do PostgreSQL utilizada foi a 16.2. Abaixo, apresentamos um resumo das especificações do computador utilizado para os testes:

- **Processador (CPU):** Intel Core i5 de 10^a geração.
- **Memória RAM:** 8 GB.
- **Armazenamento:** SSD PCIe NVMe de 512 GB.
- **Tipo de Memória Persistente:** SSD (Solid State Drive) para armazenamento.
- **Versão do PostgreSQL:** 16.2
- **Número de amostragem:** 5

(a) Consulta inicial:

```
SELECT Cliente.*
FROM Cliente
JOIN Compra ON Cliente.CPF = Compra.CPF
```

WHERE Cliente.Nome ILIKE '<letras>%'

AND data_hora_partida BETWEEN 'X' AND 'Y'

ORDER BY nome;

Para a consulta inicial, temos a execução em segundo indicada pela tabela 1.

Tempo (s)
0.613
0.517
0.497
0.483
0.634

Tabela 1: Tempo de execução da consulta inicial em segundos

(b) Consulta otimizada:

SELECT Cliente.*

FROM Cliente

WHERE Cliente.Nome ILIKE '<letras>%'

AND Cliente.CPF IN (SELECT Compra.CPF FROM Compra WHERE
data_hora_partida BETWEEN 'X' AND 'Y')

ORDER BY nome;

A fim de otimizar a consulta, eliminou-se a junção explícita e substituiu a mesma por uma subconsulta IN. A subconsulta busca CPF em Compra dentro do intervalo de tempo especificado. Além disso, a consulta otimizada aplica filtros na subconsulta, garantindo que apenas os CPF relevantes sejam considerados antes de realizar a filtragem na tabela Cliente. Os resultados de tais melhorias gerou o tempo de execução da tabela 2. Além disso, a tabela 3 reúne a média de tempos de ambas a consulta e a performance em porcentagem.

Tempo (s)
0.077

0.148
0.127
0.097
0.180

Tabela 2: Tempo de execução da consulta otimizada em segundos

	Consulta Inicial	Consulta Otimizada	Diferença (%)
Tempo de execução	0.5488	0.1258	77.07

Tabela 3. Resumo dos Resultado. Fórmula para cálculo do % = $((Vot - Vini)/Vini \times 100)$

(c) Plano de consulta inicial:

	QUERY PLAN text
1	Gather Merge (cost=29833.89..40575.44 rows=92064 width=198) (actual time=376.635..421.490 rows=108498 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Sort (cost=28833.86..28948.94 rows=46032 width=198) (actual time=320.947..335.867 rows=36166 loops=3)
5	Sort Key: cliente.nome
6	Sort Method: external merge Disk: 9592kB
7	Worker 0: Sort Method: external merge Disk: 6840kB
8	Worker 1: Sort Method: external merge Disk: 6800kB
9	-> Hash Join (cost=342.89..20862.10 rows=46032 width=198) (actual time=3.303..159.115 rows=36166 loops=3)
10	Hash Cond: ((compra.cpf)::text = (cliente.cpf)::text)
11	-> Parallel Seq Scan on compra (cost=0.00..19431.15 rows=414328 width=69) (actual time=0.342..87.003 rows=331528 loops=3)
12	Filter: ((data_hora_partida >= '2024-01-30 00:00:00'::timestamp without time zone) AND (data_hora_partida <= '2024-12-29 00:00:00'::timestamp without time zone))
13	-> Hash (cost=329.00..329.00 rows=1111 width=129) (actual time=2.796..2.797 rows=1093 loops=3)
14	Buckets: 2048 Batches: 1 Memory Usage: 193kB
15	-> Seq Scan on cliente (cost=0.00..329.00 rows=1111 width=129) (actual time=0.379..2.468 rows=1093 loops=3)
16	Filter: ((nome)::text ~* 'A%':text)
17	Rows Removed by Filter: 8907
18	Planning Time: 1.702 ms
19	Execution Time: 427.263 ms

Figura 1: Plano de consulta inicial

Este plano de consulta executa uma varredura sequencial na tabela cliente, filtrando por nomes que começam com a letra 'A' (linhas 15 e 16), seguido pela criação de uma tabela hash com os resultados (linha 13). Posteriormente, é realizada uma junção hash entre as tabelas compra e cliente, onde ambas são escaneadas sequencialmente (linha 14). A consulta utiliza paralelismo, com dois trabalhadores planejados e lançados (linhas 2 e 3). Após a junção hash, os dados são ordenados e mesclados (linhas 10 a 4). Por fim, é realizada uma operação de sort para ordenar os clientes pelo nome (linha 5).

(d) Plano de consulta otimizado:

```
5 EXPLAIN ANALYSE
6 SELECT Cliente.*
7 FROM Cliente
8 WHERE Cliente.Nome ILIKE 'A%'
9 AND Cliente.CPF IN (SELECT Compra.CPF FROM Compra WHERE data_hora_partida BETWEEN '2024-01-30' AND '2024-12-29')
10 ORDER BY nome;
...
```

	Data Output	Messages	Notifications
1	Sort (cost=1009.69..1012.47 rows=1111 width=129) (actual time=24.614..24.647 rows=1093 loops=1)		
2	Sort Key: cliente.nome		
3	Sort Method: quicksort Memory: 220kB		
4	-> Nested Loop Semi Join (cost=0.42..953.48 rows=1111 width=129) (actual time=0.071..21.835 rows=1093 loops=1)		
5	-> Seq Scan on cliente (cost=0.00..329.00 rows=1111 width=129) (actual time=0.028..8.875 rows=1093 loops=1)		
6	Filter: ((nome)::text ~* 'A%':text)		
7	Rows Removed by Filter: 8907		
8	-> Index Only Scan using compra_pkey on compra (cost=0.42..6.32 rows=99 width=11) (actual time=0.011..0.011 rows=1 loops=1093)		
9	Index Cond: ((cpf = (cliente.cpf)::text) AND (data_hora_partida >= '2024-01-30 00:00:00':timestamp without time zone) AND (data_hora_partida <= '2024-12-29 00:00:00':timestamp without time ...		
10	Heap Fetches: 0		
11	Planning Time: 0.453 ms		
12	Execution Time: 24.714 ms		

Figura 2: Plano de consulta otimizado

O consulta otimizado utiliza uma varredura sequencial na tabela cliente filtrando por nomes que começam com a letra 'A', seguido por um Nested Loop Semi Join com a tabela compra usando um índice (linhas 5 e 8), o que é mais eficiente do que a junção hash completa usada anteriormente. A utilização de um Index Only Scan na tabela compra melhora a performance, pois evita leituras desnecessárias no heap, resultando em menor tempo de execução, como pode ser observado na tabela 3.

7.2. Consulta de busca absoluta (Ana):

Nesta etapa, realizaremos a otimização da consulta:

```
EXPLAIN ANALYSE SELECT Cliente.*
FROM Compra
JOIN Cliente ON Cliente.CPF = Compra.CPF
WHERE Compra.Nome_Cidade_Destino = 'Recife' AND
DATE(Compra.Data_Hora_Chegada) = '2024-08-04';
```

Segue as informações necessárias do notebook e do PostgreSQL:

- **Processador (CPU):** Intel Core i7 8ª geração.
- **Memória RAM:** 8 GB.
- **Armazenamento:** SSD 240 GB no disco C.
- **Tipo de Memória Persistente:** SSD disco C.
- **Versão do PostgreSQL:** 16
- **Número de amostragem:** 5

a) Segue o Query Plain da consulta 2 não otimizada:

	QUERY PLAN text
1	Nested Loop (cost=1000.28..21485.43 rows=1 width=129) (actual time=7.528..282.650 rows=13 loops=1)
2	-> Gather (cost=1000.00..21477.12 rows=1 width=11) (actual time=7.366..281.874 rows=13 loops=1)
3	Workers Planned: 2
4	Workers Launched: 2
5	-> Parallel Seq Scan on compra (cost=0.00..20477.02 rows=1 width=11) (actual time=80.226..235.470 rows=4 loops=...
6	Filter: (((nome_cidade_destino)::text = 'Recife)::text) AND (date(data_hora_chegada) = '2024-08-04'::date))
7	Rows Removed by Filter: 331524
8	-> Index Scan using cliente_pkey on cliente (cost=0.29..8.30 rows=1 width=129) (actual time=0.055..0.055 rows=1 loops=...
9	Index Cond: ((cpf)::text = (compra.cpf)::text)
10	Planning Time: 4.000 ms
11	Execution Time: 283.296 ms

A operação de Nested Loop indica que o PostgreSQL está usando um loop aninhado para combinar as tabelas Compra e Cliente. Com 1000.28 que é o custo inicial e 21485.43 é o custo total estimado. O número estimado de linhas que a operação retornará é de 1 linha estimada, sendo a largura média das linhas retornadas (em bytes), de 129. O tempo real gasto nesta operação foi de 7.528 ms e terminando em 282.650 ms. Foi retornado um total de 13 linhas. E a operação foi executada apenas uma vez.

O Gather reúne os resultados de várias execuções paralelas, com 2 trabalhadores planejados e realmente iniciados, com um total de 13 linhas retornadas e um custo estimado de 1000.00..21477.12. Foi realizada também uma leitura sequencial paralela na tabela Compra, com o tempo real gasto de 80.226..235.470, com as condições do WHERE aplicadas como filtro. Pela minha surpresa, o otimizador decidiu utilizar um index, a chave primária cliente_pkey na tabela Cliente. Com custo estimado da operação de leitura de 0.29..8.30, tempo real gasto: 0.055..0.055 e condições de índice aplicadas: (cpf)::text = (compra.cpf)::text), ou seja, recupera os registros correspondentes com base no campo CPF. Tempo de execução: 283.296 ms.

Tempo (s)
0,283
0,267
0,255
0,243
0,243

Tabela 4: Tempo de execução da consulta inicial em segundos

b) Segue a consulta e o Query Plain da consulta 2 otimizada:

```
CREATE INDEX idx_compra_cidade_data ON compra (nome_cidade_destino,
data_hora_chegada);

EXPLAIN ANALYSE SELECT Cliente.*
FROM Compra
JOIN Cliente ON Cliente.CPF = Compra.CPF
WHERE Compra.Nome_Cidade_Destino = 'Recife' AND Data_Hora_Chegada >=
'2024-08-04 00:00:00'
AND Data_Hora_Chegada < '2024-08-05 00:00:00';
```

Modificamos a consulta tirando o operador DATE(), pois ele adiciona uma complexidade maior, com ele, é preciso converter cada linha de Timestamp para Date e só depois comparar com a data que colocamos no WHERE. E para facilitar ainda mais nossa consulta, criamos um índice composto, pois o índice pode ser usado para encontrar rapidamente todas as linhas que atendem à condição da cláusula WHERE.

QUERY PLAN	
text	
1	Nested Loop (cost=0.71..53.73 rows=4 width=129) (actual time=0.660..0.748 rows=13 loops=1)
2	-> Index Scan using idx_compra_cidade_data on compra (cost=0.42..20.51 rows=4 width=11) (actual time=0.646..0.660 rows=13 loops=1)
3	Index Cond: (((nome_cidade_destino)::text = 'Recife'::text) AND (data_hora_chegada >= '2024-08-04 00:00:00'::timestamp without time zone) AND (data_hora_chegada < '2024-08-05 00:00:00'::timestamp without time zone))
4	-> Index Scan using cliente_pkey on cliente (cost=0.29..8.30 rows=1 width=129) (actual time=0.006..0.006 rows=1 loops=13)
5	Index Cond: ((cpf)::text = (compra.cpf)::text)
6	Planning Time: 1.590 ms
7	Execution Time: 0.835 ms

Foi utilizado o Nested Loop Join que é um método de junção onde para cada linha da tabela externa, o banco de dados procura correspondências na tabela interna.

O número de linhas estimadas é de 4, e linhas reais 13. A largura de 129 que é uma média (em bytes) de cada linha produzida.

O banco de dados está usando um índice (idx_compra_cidade_data) para acessar as linhas da tabela compra. Aqui, a largura foi de 11, custo 0.42..2051, 4 linhas planejadas e 13 retornadas e foram aplicados a condições do WHERE.

O banco também está usando um índice (cliente_pkey) para acessar as linhas da tabela cliente. Foi criado automaticamente pelo PostgreSQL quando definimos a coluna como chave primária.

Tempo (s)
0,000835
0,000227
0,000183

0,000195
0,000196

Tabela 5: Tempo de execução da consulta inicial em segundos

	Consulta Inicial	Consulta Otimizada	Diferença (%)
Tempo de execução	258,2 ms	0,3272 ms	99,80%

Tabela 6. Resumo dos Resultado. Fórmula para cálculo do % = $((Vot - Vini)/Vini \times 100)$

7.2.1 Testando outros valores:

```
EXPLAIN ANALYSE SELECT Cliente.*
FROM Compra
JOIN Cliente ON Cliente.CPF = Compra.CPF
WHERE Compra.Nome_Cidade_Destino = 'Areias' AND Data_Hora_Chegada >=
'2024-07-09 00:00:00'
AND Data_Hora_Chegada < '2024-08-10 00:00:00';
```

Podemos perceber que o intervalo das datas é maior e a cidade também é diferente, nesse caso ele utilizou o plano:

	QUERY PLAN
1	Hash Join (cost=434.66..799.69 rows=97 width=129) (actual time=2.487..2.626 rows=162 loops=1)
2	Hash Cond: ((compra.cpf)::text = (cliente.cpf)::text)
3	-> Bitmap Heap Scan on compra (cost=5.66..370.44 rows=97 width=11) (actual time=0.056..0.141 rows=162 loops=1)
4	Recheck Cond: (((nome_cidade_destino)::text = 'Areias'::text) AND (data_hora_chegada >= '2024-07-09 00:00:00'::timestamp without time zone) AND (data_hora_chegada <
5	Heap Blocks: exact=161
6	-> Bitmap Index Scan on idx_compra_cidade_data (cost=0.00..5.64 rows=97 width=0) (actual time=0.035..0.035 rows=162 loops=1)
7	Index Cond: (((nome_cidade_destino)::text = 'Areias'::text) AND (data_hora_chegada >= '2024-07-09 00:00:00'::timestamp without time zone) AND (data_hora_chegada <
8	-> Hash (cost=304.00..304.00 rows=10000 width=129) (actual time=2.349..2.350 rows=10000 loops=1)
9	Buckets: 16384 Batches: 1 Memory Usage: 1738kB
10	-> Seq Scan on cliente (cost=0.00..304.00 rows=10000 width=129) (actual time=0.012..0.533 rows=10000 loops=1)
11	Planning Time: 0.299 ms
12	Execution Time: 2.822 ms

```
EXPLAIN ANALYSE SELECT Cliente.*
FROM Compra
JOIN Cliente ON Cliente.CPF = Compra.CPF
```

```
WHERE Compra.Nome_Cidade_Destino = 'São José do Rio Pardo' AND
Data_Hora_Chegada >= '2024-07-16 00:00:00'
AND Data_Hora_Chegada < '2024-07-17 00:00:00';
```

Neste caso, utilizamos outra cidade com o intervalo de apenas 1 dia e o resultado foi a utilização de um plano parecido com a primeira consulta otimizada.

	QUERY PLAN text
1	Nested Loop (cost=0.71..41.40 rows=3 width=129) (actual time=0.027..0.149 rows=22 loops=1)
2	-> Index Scan using idx_compra_cidade_data on compra (cost=0.42..16.49 rows=3 width=11) (actual time=0.015..0.027 rows=22 loops=1)
3	Index Cond: (((nome_cidade_destino)::text = 'São José do Rio Pardo'::text) AND (data_hora_chegada >= '2024-07-16 00:00:00'::timestamp without time zone))
4	-> Index Scan using cliente_pkey on cliente (cost=0.29..8.30 rows=1 width=129) (actual time=0.005..0.005 rows=1 loops=22)
5	Index Cond: ((cpf)::text = (compra.cpf)::text)
6	Planning Time: 0.293 ms
7	Execution Time: 0.172 ms

8. Programação Interna do SGBD

Em ambos casos foi utilizado o “USING”, que evita SQL Injection ao transformar as variáveis passadas em string.

Procedure da consulta 1:

```
CREATE OR REPLACE FUNCTION get_consulta_1(
    _letras VARCHAR,
    p_start_date DATE,
    p_end_date DATE
)
RETURNS TABLE (
    cpf VARCHAR,
    nome VARCHAR,
    endereco VARCHAR,
    telefone VARCHAR,
    email VARCHAR
) AS $$
BEGIN
    RETURN QUERY
    EXECUTE
    'SELECT Cliente.*
    FROM Cliente
    WHERE Cliente.Nome ILIKE $1 || "%"
    AND Cliente.CPF IN (SELECT Compra.CPF FROM Compra WHERE
    data_hora_partida BETWEEN $2 AND $3)'
```

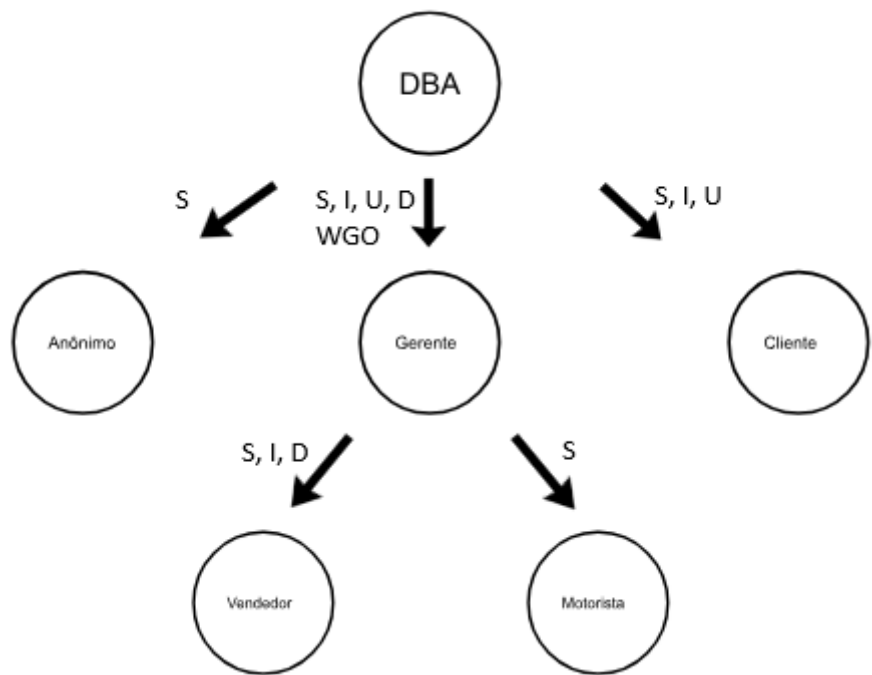


```
ORDER BY nome;'
  USING _letras, p_start_date, p_end_date;
END;
$$ LANGUAGE plpgsql;
```

Procedure da consulta 2:

```
CREATE OR REPLACE FUNCTION get_consulta_2(
  _cidade VARCHAR,
  p_start_date DATE,
  p_end_date DATE
)
RETURNS TABLE (
  cpf VARCHAR,
  nome VARCHAR,
  endereco VARCHAR,
  telefone VARCHAR,
  email VARCHAR
) AS $$
BEGIN
  RETURN QUERY
  EXECUTE
  '
SELECT Cliente.*
FROM Compra
JOIN Cliente ON Cliente.CPF = Compra.CPF
WHERE Compra.Nome_Cidade_Destino = $1 AND Data_Hora_Chegada >= $2
  AND Data_Hora_Chegada < $3'
  USING _cidade, p_start_date, p_end_date;
END;
$$ LANGUAGE plpgsql;
```

9. Controle de acesso

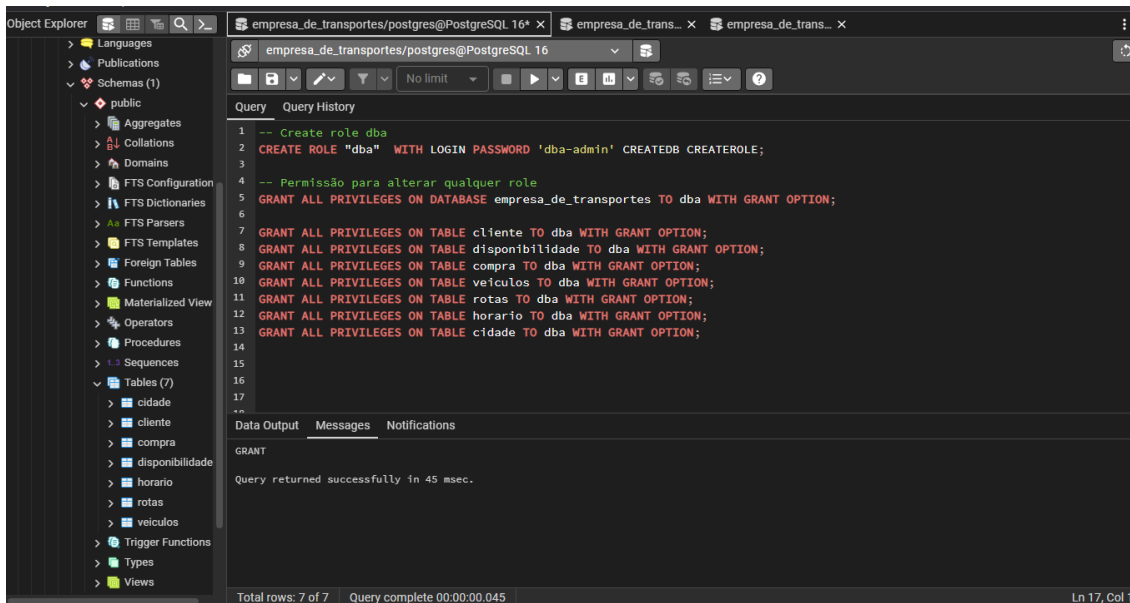


Grafo 1. Controle de acesso

Tabelas/Usuários	DBA	Gerente	Vendedores	Motorista	Clientes	Anônimo
Cliente	S, I, D,U	S, U	S, U	Sem acesso	S, U	Sem acesso
Disponibilidade	S, I, D,U	S, U	S	Sem acesso	S	S
Compra	S, I, D,U	S, I, U, D	S, I, D	S	S, I	Sem acesso
Veículos	S, I, D,U	S, U	S	Sem acesso	Sem acesso	Sem acesso
Rotas	S, I, D,U	S, U	S	Sem acesso	S	Sem acesso
Horário	S, I, D,U	S, U	S	Sem acesso	Sem acesso	Sem acesso
Cidade	S, I, D,U	S	S	Sem acesso	S	Sem acesso

Tabela 7. Exemplo de perfis de usuários, seu papel e privilégios (SELECT (S), INSERT (I), DELETE (D) e UPDATE (U))

Print da criação do perfil administrador:



Conexão com cada tela:

Usuário Anônimo: Quando um usuário não autenticado acessa a aplicação, ele opera sob o perfil de "Usuário Anônimo". Esse perfil é configurado para ter permissões bastante limitadas, permitindo apenas a visualização de telas públicas. Essas telas podem incluir informações gerais ou uma página inicial genérica, sem acesso a dados sensíveis ou funcionalidades críticas. O "Usuário Anônimo" possui acesso apenas para consultas (SELECT) em tabelas públicas, garantindo que nenhum dado crítico seja exposto sem a devida autenticação.

Perfil Motorista e Vendedor: Uma vez autenticado, o usuário que desempenha o papel de motorista ou vendedor é direcionado para telas de consulta específicas dentro da aplicação. Esses perfis são configurados para acessar informações que são essenciais para suas operações diárias, informações de clientes, e detalhes da compra. As permissões atribuídas a esses perfis permitem apenas operações de leitura (SELECT) ou não tem acesso nas tabelas relevantes. Isso garante que esses usuários possam acessar as informações de que necessitam sem alterar ou comprometer os dados existentes.

Perfil Gerente: O perfil de gerente é configurado com um nível mais elevado de acesso, refletindo suas responsabilidades de supervisão e administração. Os gerentes têm acesso a telas que permitem operações completas de gerenciamento de dados, conhecidas como CRUD (Create, Read, Update, Delete). Essas operações são essenciais para a manutenção e atualização de dados críticos, como informações de compra e rotas. Um gerente pode, por exemplo, inserir novos registros, atualizar informações existentes, ou até mesmo excluir dados desatualizados ou incorretos. As permissões associadas ao perfil de gerente incluem SELECT, INSERT, UPDATE, e

DELETE em tabelas-chave, como "Compra", "Rotas", e "Horário", proporcionando a flexibilidade necessária para a gestão eficaz dos processos internos.

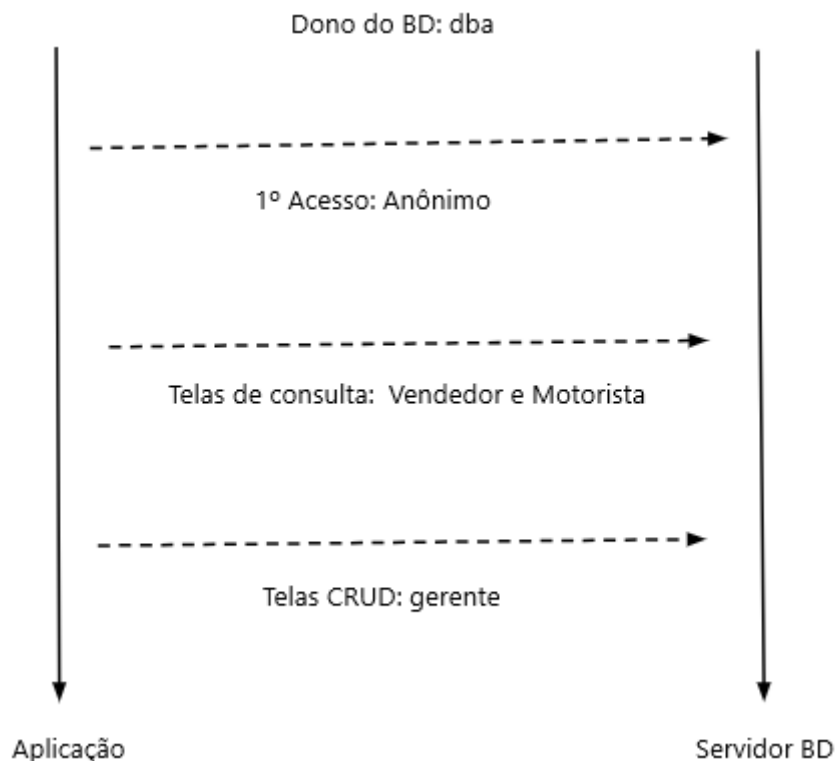


Gráfico 2. Conexões usando perfis de usuários

10. Considerações Finais

Desde o início, foi priorizado o desenvolvimento de um modelo de dados sólido, com a criação de um modelo Entidade-Relacionamento que capturou as complexidades do negócio da "Viação Estrelar". A transição deste modelo conceitual para o projeto lógico foi realizada com cuidado, assegurando que as relações entre as entidades fossem otimizadas e que o esquema do banco de dados suportasse as consultas e operações necessárias com eficiência.

O script SQL desenvolvido para a criação do banco de dados refletiu o rigor e a atenção aos detalhes exigidos pelo projeto.

O projeto cumpre seu objetivo de suportar a operação diária da empresa, facilitando a gestão de clientes, veículos, rotas e horários, ao mesmo tempo que assegura a satisfação e a confiança dos clientes através de um sistema confiável e eficiente. Este sistema é uma base sólida sobre a qual a "Viação Estrelar" pode construir seu futuro de crescimento e inovação no setor de transportes.