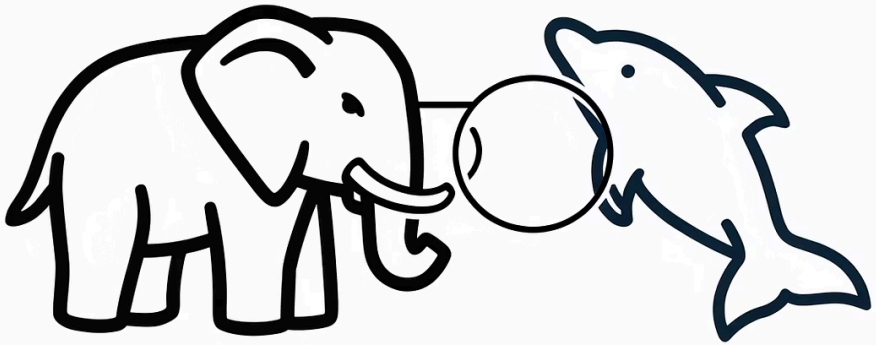


CEFET/RJ

Análise de Planos de Execução de Consultas nos SGBD

Um guia técnico completo sobre como os Sistemas de Gerenciamento de Banco de Dados processam, otimizam e executam consultas SQL. Explore as estratégias internas que transformam comandos SQL em operações eficientes de acesso a dados.

Eduardo Ogasawara

eduardo.ogasawara@cefet-rj.br
https://eic.cefet-rj.br/~eogasawara

FUNDAMENTOS

Esquema de Análise do Banco de Dados

Para compreender os planos de execução, vamos trabalhar com um esquema empresarial clássico que modela departamentos, empregados, projetos e suas inter-relações. Este esquema servirá como base para todas as análises de consultas apresentadas.

1

DEPARTAMENTO

PRIMARY KEY: DNUMERO

Representa as unidades organizacionais da empresa, identificadas por um número único de departamento.

2

EMPREGADO

PRIMARY KEY: CPF

FOREIGN KEYS: GERENTECPF → EMPREGADO(CPF), DNO → DEPARTAMENTO(DNUMERO)

Contém informações dos funcionários, incluindo relacionamento hierárquico com gerentes e vinculação ao departamento.

3

DEPT_LOCALIZACOES

PRIMARY KEY: (DNUMERO, DLOCALIZACAO)

FOREIGN KEY: DNUMERO → DEPARTAMENTO(DNUMERO)

Armazena as múltiplas localizações físicas onde cada departamento opera.

4

PROJETO

PRIMARY KEY: PNUMERO

FOREIGN KEY: DNUM → DEPARTAMENTO(DNUMERO)

Registra os projetos da empresa e sua associação com os departamentos responsáveis.

5

TRABALHA_EM

PRIMARY KEY: (ECPF, PNO)

FOREIGN KEYS: ECPF → EMPREGADO(CPF), PNO → PROJETO(PNUMERO)

Tabela associativa que registra a alocação de empregados em projetos específicos.

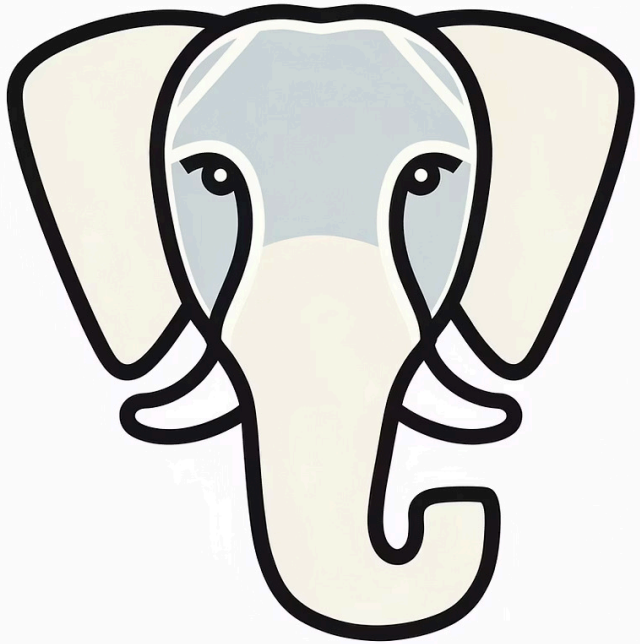
6

DEPENDENTE

PRIMARY KEY: (ECPF, NOME_DEPENDENTE)

FOREIGN KEY: ECPF → EMPREGADO(CPF)

Mantém informações sobre os dependentes de cada empregado para fins de benefícios.



PostgreSQL

O PostgreSQL é um dos sistemas de gerenciamento de banco de dados relacional mais avançados e robustos disponíveis. Vamos explorar como este SGBD analisa, otimiza e executa consultas SQL, examinando seus planos de execução detalhados.

O comando `EXPLAIN` do PostgreSQL revela a estratégia escolhida pelo otimizador de consultas, incluindo tipos de varredura, métodos de junção, custos estimados e estatísticas de execução real.

CHAVE PRIMÁRIA

Consulta pela Chave Primária

Busca de empregado a partir do CPF


```
SELECT *
FROM EMPREGADO E
WHERE E.CPF = '999887777'
```

Plano de Execução:

- **Seq Scan:** Varredura sequencial completa
- **Custo:** 1.35

Este é o tipo mais eficiente de consulta. Como CPF é a chave primária, o PostgreSQL utiliza um **Seq Scan**. Apesar da condição sobre a chave primária, o otimizador opta por varredura sequencial devido ao reduzido tamanho da tabela, tornando o custo do acesso via índice desnecessário nesse cenário.

BUSCA POR VALOR

Consulta por Valor sem Índice

Busca de empregado a partir do primeiro nome


```
SELECT *
FROM EMPREGADO E
WHERE E.PNOME = 'Alicia'
```

Plano de Execução:

- **Seq Scan:** Varredura sequencial completa
- **Custo:** 1.35

Sem um índice na coluna PNOME, o PostgreSQL realiza uma **Sequential Scan** (varredura sequencial), lendo todas as tuplas da tabela. Consultas por valor sem índice exigem varredura da tabela e apresentam custo proporcional ao número de linhas, enquanto buscas por chave primária podem ser resolvidas por acesso direto.

SUBCONSULTA

Subconsulta na Projeção

Lista de departamentos com seus respectivos gerentes

Query Editor | Query History

```
1  SELECT D.*,
2     (SELECT PNOME FROM EMPREGADO E WHERE E.CPF = D.GERCPF) AS PNOME
3  FROM DEPARTAMENTO D
4
5
```

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows / Plan
1.	→ Seq Scan on departamento as d (cost=0..5.08 rows=3 width=117)	3
2.	→ Seq Scan on empregado as e (cost=0..1.35 rows=1 width=8) Filter: ((cpf)::bpchar = (d.gercpf)::bpchar)	1

```
SELECT D.*,
  (SELECT PNOME FROM EMPREGADO E
WHERE E.CPF = D.GERCPF) AS PNOME
FROM DEPARTAMENTO D
```

Esta consulta utiliza uma **subconsulta escalar** na lista de projeção. O PostgreSQL executa a varredura sequencial na tabela DEPARTAMENTO e, para cada linha retornada, executa uma busca pontual em EMPREGADO baseada na chave primária CPF.

01

Seq Scan em DEPARTAMENTO

Custo 0.00 → 5.08, retorna 3 linhas

02

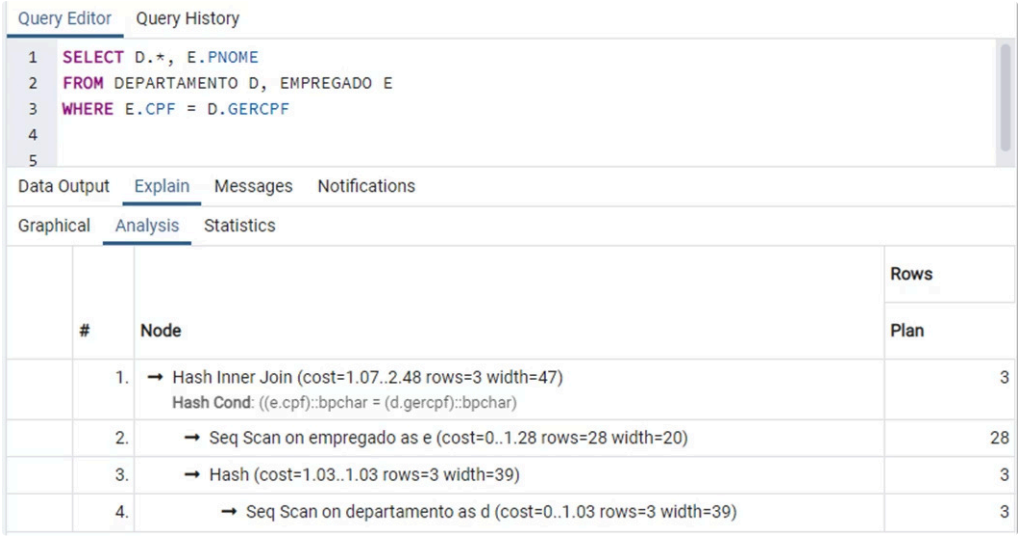
Seq Scan por linha em EMPREGADO

Custo 0.00 → 1.35, busca o gerente correspondente

JUNÇÃO

Consulta Equivalente com Junção

Lista de departamentos com seus respectivos gerentes




```
SELECT D.*, E.PNOME
FROM DEPARTAMENTO D, EMPREGADO E
WHERE E.CPF = D.GERCPF
```

Plano de Execução:

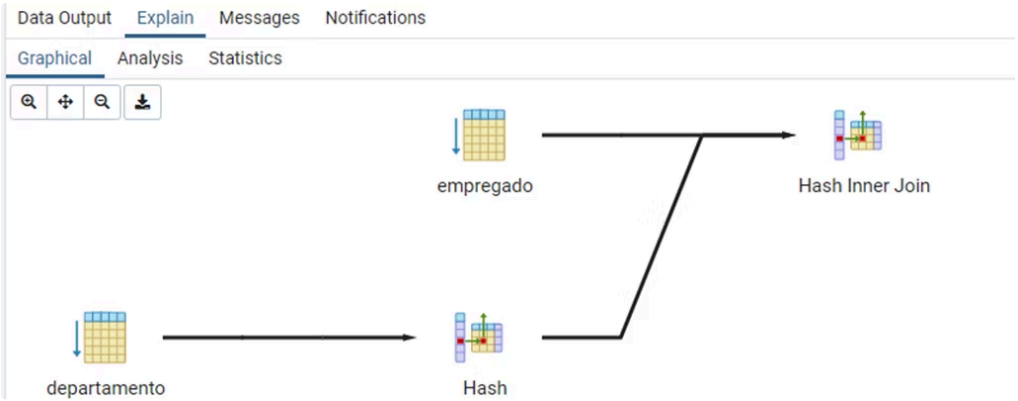
- **Hash Join:** Método de junção escolhido
- **Custo total:** 1.07 → 2.48

Esta consulta produz o mesmo resultado que a subconsulta anterior, mas utiliza uma **junção explícita**. O PostgreSQL escolhe um **Hash Join**, criando uma tabela hash da menor tabela (DEPARTAMENTO) e fazendo o probe na maior (EMPREGADO).

Importante: O custo do Hash Join (2.48) é significativamente menor que a subconsulta na projeção (5.08), demonstrando que junções explícitas geralmente são mais eficientes que subconsultas correlacionadas repetidas.

Árvore de Consulta: Junção Visualizada

Representação visual do plano de execução




```
SELECT D.*, E.PNOME
FROM DEPARTAMENTO D, EMPREGADO E
WHERE E.CPF = D.GERCPF
```

A árvore de execução ilustra a hierarquia de operações realizadas pelo PostgreSQL. No topo, temos o **Hash Join** que combina os resultados das varreduras nas tabelas base. O Hash é construído sobre DEPARTAMENTO e utilizado para buscar correspondências em EMPREGADO.

Seq Scan DEPARTAMENTO

Leitura completa

Seq Scan EMPREGADO

Leitura completa

Hash Join

Combina resultados

SUBCONSULTA IN

Subconsulta Não Correlacionada com IN

Empregados que possuem dependentes

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Hash Inner Join (cost=8.61..10.21 rows=23 width=78) Hash Cond: ((e.cpf)::bpchar = (d.ecpf)::bpchar)	23
2.	→ Seq Scan on empregado as e (cost=0..1.28 rows=28 width=78)	28
3.	→ Hash (cost=8.32..8.32 rows=23 width=12)	23
4.	→ Aggregate (cost=8.09..8.32 rows=23 width=12)	23
5.	→ Seq Scan on dependente as d (cost=0..7.07 rows=407 width=12)	407

```
SELECT *
FROM EMPREGADO E
WHERE E.CPF IN
  (SELECT D.ECPF
   FROM DEPENDENTE D)
```

Operações principais:

- Hash Semi Join
- Seq Scan em DEPENDENTE
- Seq Scan em EMPREGADO
- Custo total: 10.21

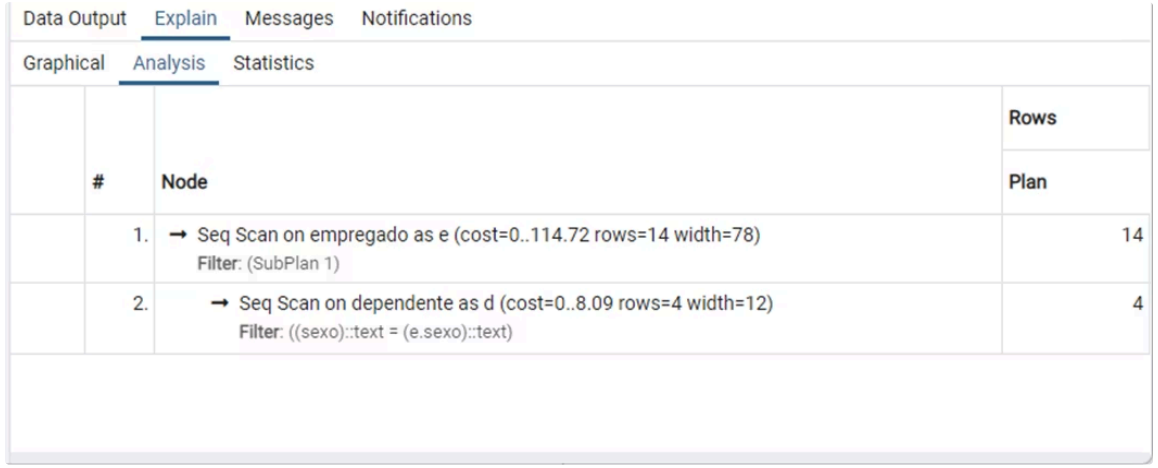
Uma **subconsulta não correlacionada** é executada uma única vez, independentemente da consulta externa. O PostgreSQL otimiza a cláusula IN usando um **Hash Semi Join**, que constrói uma tabela hash dos CPFs em DEPENDENTE e verifica cada empregado contra essa hash.

O Semi Join é eficiente porque para de procurar assim que encontra a primeira correspondência, já que estamos apenas testando existência, não recuperando valores da subconsulta.

CORRELACIONADA

Subconsulta Correlacionada com IN

Empregados com dependentes do mesmo sexo




```
SELECT *
FROM EMPREGADO E
WHERE E.CPF IN
 (SELECT D.ECPF FROM DEPENDENTE D
WHERE D.SEXO = E.SEXO)
```

Esta é uma **subconsulta correlacionada** porque referencia uma coluna da consulta externa (E.SEXO).

Seq Scan

O otimizador não conseguiu (ou não considerou seguro, devido às restrições semânticas do IN em presença de correlação) reescrever a subconsulta correlacionada como um *hash semijoin*, optando por varreduras sequenciais para avaliar o predicado.

Custo Total

114.72 - ligeiramente maior devido à condição adicional

✓ EXISTS

Subconsulta Correlacionada com EXISTS

Empregados com dependentes do mesmo sexo

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows / Plan
1.	→ Hash Inner Join (cost=10.13..11.64 rows=7 width=78) Hash Cond: (((e.cpf)::bpchar = (d.ecpf)::bpchar) AND ((e.sexo)::text = (d.sexo)::text))	7
2.	→ Seq Scan on empregado as e (cost=0..1.28 rows=28 width=78)	28
3.	→ Hash (cost=9.52..9.52 rows=41 width=14)	41
4.	→ Aggregate (cost=9.11..9.52 rows=41 width=14)	41
5.	→ Seq Scan on dependente as d (cost=0..7.07 rows=407 width=14)	407

```
SELECT *
FROM EMPREGADO E
WHERE EXISTS
 (SELECT * FROM DEPENDENTE D
  WHERE D.ECPF = E.CPF
  AND D.SEXO = E.SEXO)
```

Esta é uma **subconsulta correlacionada** porque referencia uma coluna da consulta externa (E.CPF e E.SEXO).

- Custo total: 11.64

A cláusula **EXISTS** testa apenas a existência de linhas, não seus valores. Neste caso, o PostgreSQL gera um plano de execução com Hash Semi-Join.

Boas Práticas: Use EXISTS em vez de IN quando a subconsulta retorna muitas colunas ou quando você está apenas testando existência, não buscando valores específicos.

AGREGAÇÃO

Consulta com Agregação e Group By

Quantidade de empregados que trabalham em mais de um projeto

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Aggregate (cost=2.92..3.12 rows=5 width=16) Filter: (count(*) > 1)	5
2.	→ Hash Inner Join (cost=1.63..2.84 rows=16 width=8) Hash Cond: ((te.ecpf)::bpchar = (e.cpf)::bpchar)	16
3.	→ Seq Scan on trabalha_em as te (cost=0..1.16 rows=16 width=12)	16
4.	→ Hash (cost=1.28..1.28 rows=28 width=20)	28
5.	→ Seq Scan on empregado as e (cost=0..1.28 rows=28 width=20)	28

```
SELECT E.PNOME, COUNT(*) AS QTD
FROM EMPREGADO E JOIN TRABALHA_EM TE ON
E.CPF = TE.ECPF
GROUP BY E.PNOME HAVING COUNT(*) > 1
```

Esta consulta demonstra operações de agregação com filtro HAVING. O PostgreSQL executa uma sequência sofisticada de operações:

Hash Join

Une EMPREGADO e TRABALHA_EM usando CPF = ECPF

HashAggregate

Agrupa por PNOME e conta as ocorrências

Filter (HAVING)

Mantém apenas grupos com COUNT(*) > 1

O **HashAggregate** usa uma tabela hash para agrupar linhas eficientemente por PNOME. O custo total de 3.12 reflete a junção mais a agregação. A cláusula HAVING filtra os resultados agregados, retornando apenas empregados multi-projeto.

OTIMIZAÇÃO

Criação de Índice: Impacto no Desempenho


```
CREATE INDEX EMP_PNOME_I ON EMPREGADO(PNOME)
```

Índices são fundamentais para otimização de consultas, mas devem ser usados estrategicamente. Colunas frequentemente usadas em cláusulas WHERE, JOIN, e ORDER BY são candidatas ideais para indexação.

Deve-se sempre analisar com cautela. Após criar um índice na coluna PNOME, a consulta `WHERE E.PNOME = 'Alicia'`, o SGBD não fez uso do índice. *Se existe um índice em* `PNOME`*, por que o PostgreSQL optou por* `Seq Scan` *em vez de* `Index Scan`*?*

Consideração importante: Índices aceleram leituras, mas adicionam overhead em operações de escrita (INSERT, UPDATE, DELETE). Analise o perfil de uso antes de criar índices adicionais.

MySQL: Otimização de Consultas e Índices

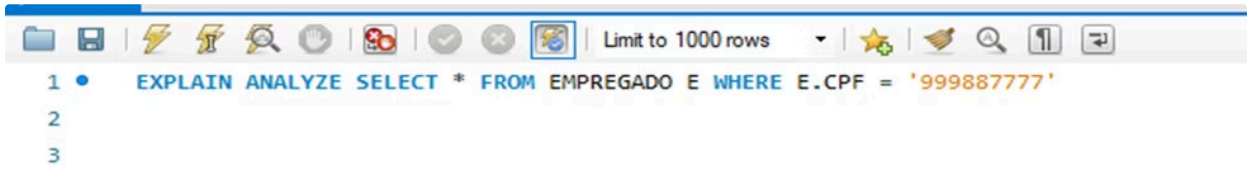
Uma análise técnica e comparativa entre MySQL e SQL Server, explorando estratégias de consulta, uso de índices e otimizações de performance para desenvolvedores e DBAs.

MYSQL

Consultas pela Chave Primária

Busca por CPF do Empregado

A consulta pela chave primária é a operação mais eficiente no MySQL. O banco de dados utiliza automaticamente o índice da chave primária (CPF) para localizar o registro de forma extremamente rápida, geralmente em tempo O(log n).

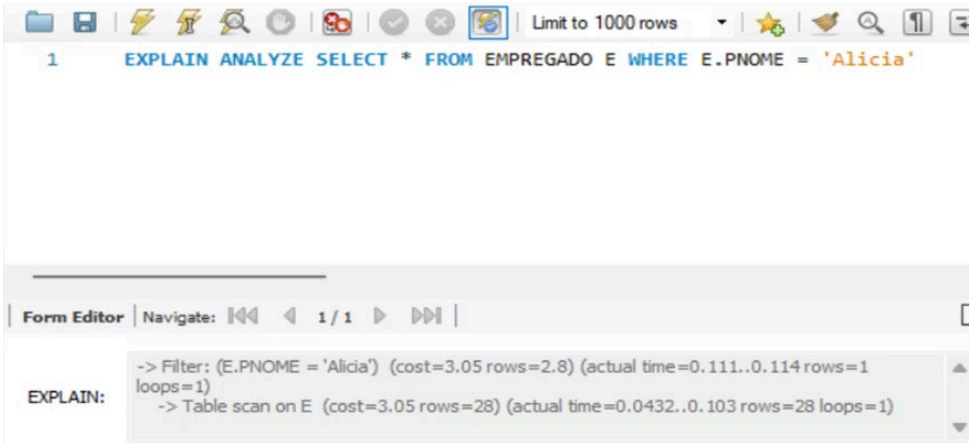



```
SELECT * FROM EMPREGADO E
WHERE E.CPF = '999887777'
```

O otimizador reconhece a igualdade sobre a chave primária e determina, ainda no planejamento, que o acesso será feito via índice, resolvendo a consulta antes da execução.

MYSQL

Consulta por Valor Sem Índice



Query SQL

```
SELECT * FROM EMPREGADO E
WHERE E.PNOME = 'Alicia'
```

Comportamento

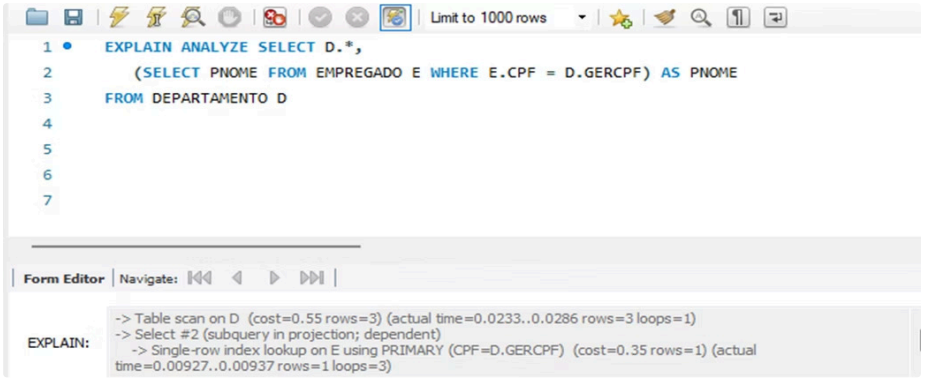
Sem índice no campo PNOME, o MySQL realiza um table scan completo, percorrendo todos os registros sequencialmente.

Impacto no Desempenho

Como o predicado não restringe o acesso por índice, o custo da consulta cresce linearmente com o tamanho da tabela, tornando essa estratégia pouco escalável.

MYSQL

Sub-consulta na Projeção



Lista de Departamentos com Gerentes

```
SELECT D.*,
  (SELECT PNOME FROM EMPREGADO E
   WHERE E.CPF = D.GERCPF)
  AS PNOME
FROM DEPARTAMENTO D
```

A subconsulta na lista de seleção é dependente da consulta externa, fazendo com que o MySQL execute um *lookup* por chave primária em EMPREGADO para cada linha retornada de DEPARTAMENTO.

O plano evidencia um padrão equivalente a um *nested loop*, ainda que não expresso como operador de junção explícito.

Atenção: Embora cada acesso interno seja eficiente, o custo total cresce proporcionalmente ao número de linhas da consulta externa, pois a subconsulta é reexecutada para cada tupla.

MYSQL

Consulta com Agregação

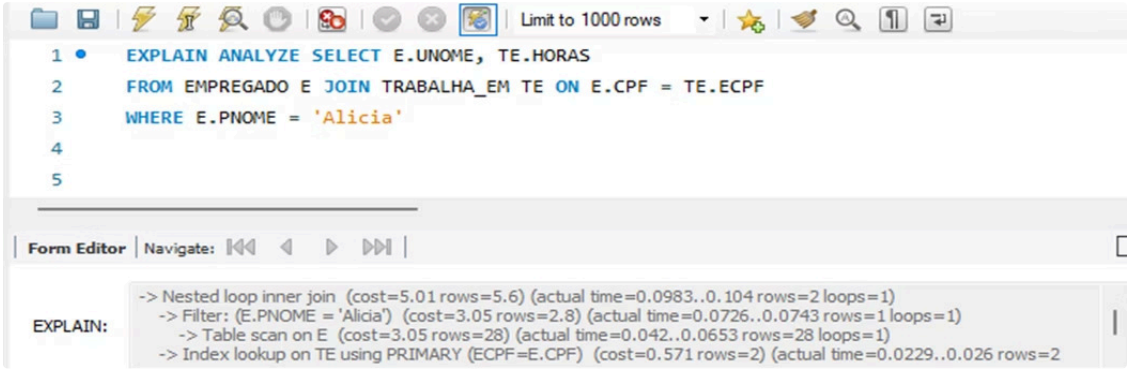

```
SELECT E.PNOME, COUNT(*) AS QTD
FROM EMPREGADO E
JOIN TRABALHA_EM TE ON E.CPF = TE.ECPF
GROUP BY E.PNOME
HAVING COUNT(*) > 1
```

O plano executa inicialmente a junção entre EMPREGADO e TRABALHA_EM por meio de um *nested loop*, produzindo o conjunto de pares necessários para a agregação.

Em seguida, o MySQL materializa os grupos em uma tabela temporária para calcular `COUNT(*)`, aplicando a cláusula `HAVING` apenas após a agregação

MYSQL

Junção com Busca por Valor



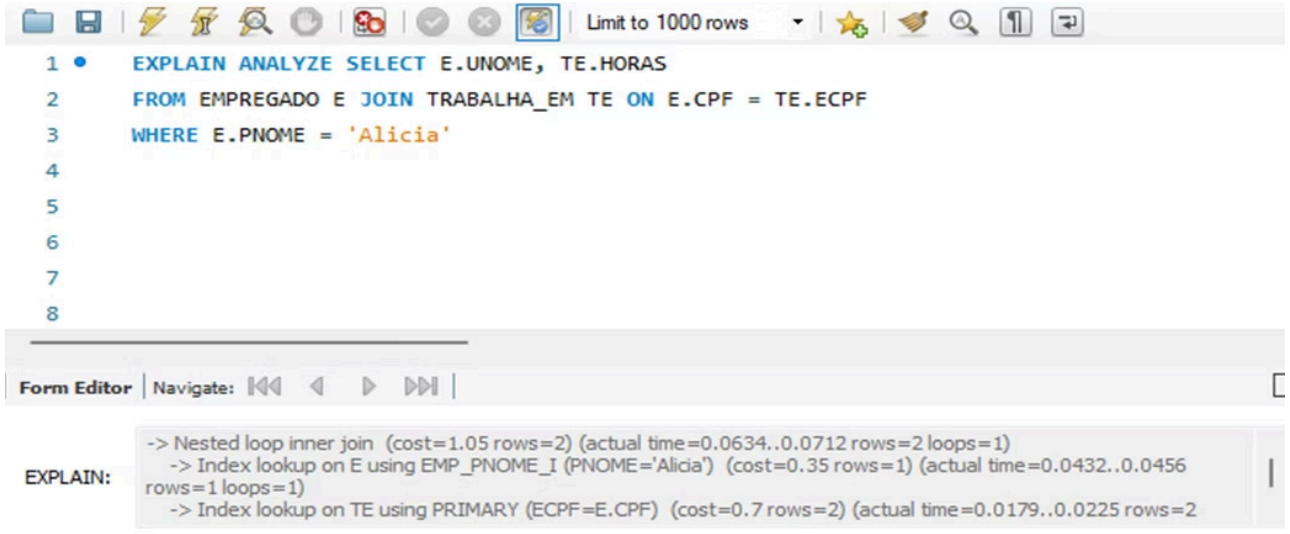

```
SELECT E.UNOME, TE.HORAS
FROM EMPREGADO E
JOIN TRABALHA_EM TE
  ON E.CPF = TE.ECPF
WHERE E.PNOME = 'Alicia'
```

O plano executa uma varredura completa em EMPREGADO para aplicar o filtro sobre `PNOME`, e para cada linha resultante realiza um *lookup* indexado em TRABALHA_EM, caracterizando um *nested loop join*.

Na ausência de um índice seletivo em `PNOME`, o custo da consulta cresce com o tamanho da tabela externa, pois o filtro precisa ser avaliado antes da junção.

OTIMIZAÇÃO

Criação de Índice para Otimização



Comando de Criação

```
CREATE INDEX EMP_PNOME_I
ON EMPREGADO(PNOME)
```

Com o índice em `PNOME`, o otimizador substitui a varredura completa por um *index lookup* em EMPREGADO, reduzindo significativamente o conjunto de linhas antes da junção.

Isso permite que a consulta seja executada como um *nested loop* totalmente indexado, com acessos pontuais em ambas as tabelas.

RECURSO AVANÇADO

Índice de Cobertura de Consulta

01

Conceito Fundamental

Um índice de cobertura contém **todas as colunas** necessárias para processar uma consulta, eliminando a necessidade de acessar a tabela base

02

Atributos de Seleção

Primeiro, inclua os campos usados no WHERE (mais restritivos primeiro) para filtrar eficientemente

03

Atributos de Junção

Segundo, inclua campos usados em JOIN para facilitar operações de junção sem lookup adicional

04

Atributos de Projeção

Terceiro, inclua colunas necessárias à projeção para permitir cobertura da consulta

Um índice de cobertura contém todas as colunas necessárias para resolver uma consulta, permitindo que o SGBD produza o resultado sem acessar a tabela base. Ao evitar leituras adicionais da tabela, essa estratégia reduz significativamente o custo de I/O e pode alterar de forma decisiva o plano de execução.

POSTGRESQL E MYSQL

Criando um Índice de Cobertura

Sintaxe com INCLUDE

POSTGRESQL

```
CREATE INDEX idx_emp_pnome
ON empregado (pnome)
INCLUDE (cpf, unome)
```

MYSQL

```
CREATE INDEX idx_emp_pnome
ON empregado (pnome, cpf, unome)
```

Este índice contém:

- **Chave:** PNOME (campo de busca)
- **Incluídos:** CPF (junção) e UNOME (projeção)

Ordem Importa

A ordem das colunas na chave afeta quais consultas podem usar o índice eficientemente

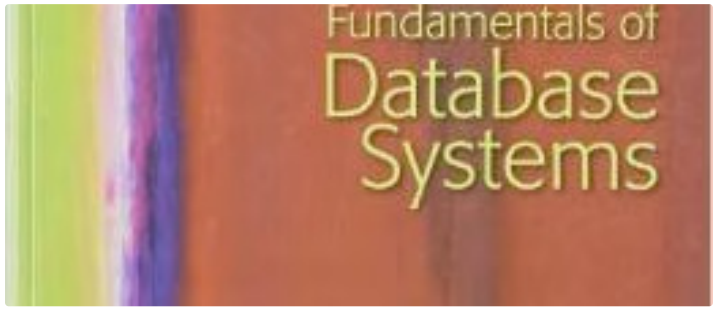
Custo-Benefício

Índices de cobertura ocupam mais espaço mas eliminam operações de lookup, valendo a pena para consultas frequentes

Manutenção

Mais colunas no índice significam maior custo em INSERT/UPDATE/DELETE - balance conforme necessidade

Referências



Elmasri & Navathe

Fundamentals of Database Systems

Pearson, 2016

Referência abrangente sobre fundamentos de sistemas de bancos de dados, cobrindo aspectos teóricos e práticos.

Korth, Sudarshan & Silberschatz

Database System Concepts

McGraw-Hill, 2019

Texto fundamental que serviu como base para a maioria dos exemplos apresentados nesta apresentação.

Özsu & Valduriez

Principles of Distributed Database Systems

Springer Nature, 2019

Obra especializada em sistemas de bancos de dados distribuídos, essencial para compreensão avançada.

Nota: A maioria dos slides apresenta exemplos e conceitos adaptados do livro Database System Concepts, reconhecido mundialmente como referência na área.