



MBA em Inteligência Artificial e Big Data
– Curso 3: Administração de Dados Complexos em Larga Escala –

★ Índices em Bases de dados ★

Caetano Traina Júnior

Grupo de Bases de Dados e Imagens – GBdI
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos

O conceito de índices na Linguagem SQL e as técnicas de como usá-los bem para agilizar as consultas.






Roteiro



- 1 Conceitos
- 2 Declaração de Índices em SQL
- 3 Táticas para o uso de índices
- 4 Gerenciando e mantendo índices





- A indexação de dados é uma das técnicas usadas para acelerar o processamento de consultas em um SGBD Relacional.
- Índices organizam o espaço de busca dos valores de um ou mais atributo de uma tabela...
- ... mas não armazenam nenhum dado além daqueles já incluídos nelas.
-  do ponto de vista de um repositório de dados estático, não são necessários, e podem ser criados e destruídos sem consequência.
-  Mas índices são essenciais para obter **velocidade** de acesso e **escalabilidade** nas operações de recuperação de dados.
- Índices são muito usados sobre atributos que tenham alta seletividade, tais como as chaves das relações mas não apenas sobre eles.
- **Mas lembre-se:**
 -  O conceito de índice não faz parte do Modelo Relacional!





Índices não são tratados pelo Modelo Relacional

- Índices são recursos para agilizar o acesso aos dados (em geral na memória permanente: discos).
- Eles não afetam nem a modelagem nem as propriedades dos dados e das operações executadas sobre eles.

Índices não são padronizados em SQL

Portanto cada gerenciador segue sua própria sintaxe para definir índices, embora todos empreguem uma sintaxe semelhante.

- A falta de um padrão tem a vantagem de permitir que cada SGBD ofereça recursos diferenciados para tratar casos de interesse.





1 Conceitos

2 Declaração de Índices em SQL

- Declaração Implícita de Índices em Restrições de Integridade
- Declaração Explícita de Índices
- Estruturas de Dados para Índices

3 Táticas para o uso de índices

4 Gerenciando e mantendo índices



Declaração de Índices em SQL



- Existem duas maneiras de se declarar índices em SQL:
 - **Implicitamente:** índices são criados automaticamente quando se declara que um conjunto de atributos é uma **chave primária** ou **candidata**;
 - **Explicitamente:** usando o comando `CREATE INDEX`.



Declaração Implícita em Restrições de Integridade



- Apesar de não fazer parte do Modelo Relacional, o conceito de índice é comumente associado ao conceito de **chaves** e às **Restrições de Integridade fundamentais** do modelo:
 - Integridade de Entidade – Chave Primária: **PRIMARY KEY**
 - Integridade de Unicidade – Chave Candidata: **UNIQUE**
 - Integridade Referencial – Chave Estrangeira: **FOREIGN KEY**
- Essas restrições são declaradas nos comandos
CREATE TABLE ou **ALTER TABLE**.
- Os SGBDs usam índices para agilizar buscas que envolvam essas restrições.

Mas o conceito de índice é mais abrangente, e é usado para muitos outros objetivos além do conceito de chaves!

Declaração Implícita em Restrições de Integridade



- As declarações **PRIMARY KEY** e **UNIQUE** implicitamente criam um novo índice.
- A declaração **FOREIGN KEY** não cria índice, mas requer que um já exista (criado usando **PRIMARY KEY**) na relação referenciada.
- Os índices implícitos não podem ser diretamente removidos:
 - ☞ isso é feito removendo a restrição.
 - ☞ No entanto, índices implícitos são índices, iguais a todos os demais.



Declaração Implícita em Restrições de Integridade



- Lembrar que a sintaxe de SQL permite que uma restrição seja declarada como:
 - **Uma restrição de atributo** – também chamada restrição de coluna, ou declarada por coluna
 - **Uma restrição de relação** – também chamada restrição da tabela

Nota:

É uma boa prática de programação identificar todas as restrições (dar um nome explícito), pois isso facilita alterá-las sempre que necessário.



Declaração Implícita em Restrições de Integridade

Restrição de atributo



- Uma **Restrição de atributo** ocorre quando ela é declarada junto com o atributo.
- Por exemplo:

```
NUSP DECIMAL(10) NOT NULL PRIMARY KEY,  
                                - Não identificada  
NUSP DECIMAL(10) NOT NULL  
                                CONSTRAINT ChaveDaRelacao PRIMARY KEY,  
                                - Identificada
```

- Restrições de atributo somente podem ser declaradas quando apenas um atributo está envolvido.



Declaração Implícita em Restrições de Integridade

Restrição de relação

- Uma **Restrição de relação** ocorre quando se declara a restrição de maneira independente.
- Por exemplo:

```
Nome VARCHAR(60) NOT NULL,  
NomeDaMae VARCHAR(60) NOT NULL,  
DataNascimento DATE NOT NULL,  
UNIQUE(Nome, NomeDaMae, DataNascimento),  
                                - Não identificada  
  
CONSTRAINT NaoRepeteNome  
    UNIQUE(Nome, NomeDaMae, DataNascimento),  
                                - Identificada
```

- Restrições de relação podem ser usadas com qualquer número de atributos (usualmente até 32).




Declaração Explícita de Índices




Como não existe padrão, existe muita variedade na sintaxe dos comandos que tratam índices nos diversos SGBDR existentes.

- Índices são um tipo de 'objeto' da base de dados que são declarados e mantidos por comandos da DDL: `CREATE`, `ALTER` e `DROP INDEX`
- O comando `CREATE INDEX` inclui a grande maioria dos recursos para trabalhar com índices,

 ... mas cada SGBD pode ter seus próprios comandos auxiliares, incluindo comandos na DML.

Por exemplo:

Em alguns SGBDs, um índice pode ser declarado *clustering* num comando da DDL, mas em  PostgreSQL, `CLUSTER` é um comando da DML, que executa a '*clusterização*' da tabela usando um índice no momento em que o comando é solicitado.

DDL – Comando CREATE INDEX

Índices podem ser declarados explicitamente usando o comando `CREATE INDEX`, cuja sintaxe geral é:

Forma geral:

```
CREATE INDEX <nome> ON <table>(<atributo> | <expressao>)  
    [INCLUDE <atributo>]  
    [WHERE <predicado>];
```

- Os atributos a serem indexados são indicados como `atributos` ou como componentes de uma `<expressao>` (**índices de expressões**), eles são chamados **Atributos de indexação**; Eles formam a **chave de busca** da estrutura de indexação.
- Atributos indicados na cláusula `INCLUDE` são **Atributos incluídos** para recuperação, mas não são usados para indexação.
- A cláusula `WHERE` indexa apenas as tuplas que atendam ao predicado. Pode-se usar qualquer atributo, mesmo que não estejam indexados. Eles são chamados **Atributos de partição**.



DDL – Comando CREATE INDEX



- O comando de criação de índices também permite indicar a estrutura de dados usada para aquele índice.
- Usualmente estão disponíveis:
 - Btree (ISAM)
 - Hash
 - Bit map
 - R-tree
- Mas cada SGBD tem um formato específico para indicar isso!



DDL – Comando CREATE INDEX

Cria um novo índice numa tabela –



CREATE INDEX –



```
CREATE [UNIQUE] INDEX [<nome>] ON <table> [USING <metodo>]
    ({<atributos> | (<expressao>)}
    [ASC | DESC]
    [NULLS {FIRST | LAST}] [, ...]    )
    [INCLUDE <atributo>]
    [WHERE <predicado>];
```

Em PostgreSQL <method> é:

- **BTREE** – é o mais usado, **UNIQUE**, ordem
- **HASH** – muito rápido para comparações por identidade ('=', '≠')
- **GIST** – múltiplos métodos, inclui **R-tree** e outros, usuário define
- **GIN** – (Generalized Inverted Files)



DDL – Comando CREATE INDEX



Exemplo – Índice simples:

```
CREATE INDEX IdxNUSP ON Matricula(NUSP);
```

Exemplo – Escolhendo a estrutura de indexação:

```
CREATE INDEX IdxNivel ON Professor USING HASH (Nivel);
```

Exemplo – Índice sobre expressão:

```
CREATE INDEX IdxUpNome ON Professor (Upper(Nome));
```

útil quando usado em consultas comparando:

```
... WHERE Upper(Professor.Nome)='RAIMUNDO'
```



DDL – Comando ALTER INDEX



- Modificações em índices, do ponto de vista lógico, se restringem à alteração do nome.
- Alterar qualquer outro parâmetro requer destruir o índice e reconstruí-lo de novo.

ALTER INDEX

```
ALTER INDEX <nome> [IF EXISTS] RENAME TO  
<novo_nome>
```

- Se especificado `[IF EXISTS]` e o índice não existir, então não acusa erro.

Por exemplo:

```
ALTER INDEX IdxUpNome RENAME TO IdxProfessor_UpNome;
```



DROP INDEX

```
DROP INDEX [IF EXISTS] <nome> [, ...]  
          [CASCADE | RESTRICT];
```

- Se especificado **IF EXISTS**, não acusa erro quando o índice não existe.
- Se especificado **CASCADE**, apaga os objetos que dependem desse índice.
- Se especificado **RESTRICT**, não apaga o índice se houver objetos que dependem desse índice.

Por exemplo:

```
DROP INDEX IF EXISTS IdxUpNome RESTRICT;
```



- Os índices são usados para organizar os dados de um subconjunto de atributos de uma relação, criando o que se chama “**Caminho de Acesso**” aos dados.
- Índices são criados sobre um ou mais atributos.
Quando ele é criado sobre dois ou mais atributos, a chave de busca é a concatenação dos valores dos atributos envolvidos.
- SGBDs em geral aceitam até 32 atributos concatenados por índice.
- Os valores usados na estrutura interna são chamados “**Chaves de acesso**” ao índice.



Estruturas de Dados para Índices

Terminologia

Terminologia:

- A indexação em SGBD envolve duas áreas da computação:
 - Bases de Dados
 - Algoritmos e Estruturas de Dados
- Cada área usa um significado para o termo “chave”:
 - Em Bases de Dados, chave é um valor que não pode repetir em outra tupla;
 - Em Algoritmos e Estruturas de Dados, chave é o valor usado para buscar um elemento armazenado na estrutura, não existe restrição de que ele seja único.
- Nesta apresentação usamos o termo **chave da relação** ou simplesmente **chave** para indicar a chave das relações como é usado em Bases de Dados, indicando que não pode ter valores repetidos,
- e o termo **chave de busca** ou **chave de acesso** para indicar o valor de busca em Algoritmos e Estruturas de Dados, onde repetições são permitidas.

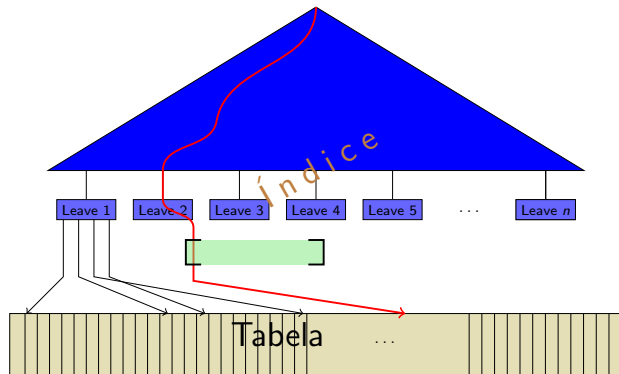




- Toda estrutura de dados tem duas partes:
 - ① A estrutura interna, que organiza os dados e constitui a estrutura propriamente dita;
 - ② A lista de tuplas, que contém ponteiros para as tuplas.
- Índices são estruturas que, em geral, podem ser criadas e apagadas a qualquer instante sem perda de dados, pois os dados básicos são mantidos sempre nas tuplas;
- Os valores dos atributos usados nos índices são copiados para as estruturas, mas a tupla é mantida íntegra.



- Estrutura de índice usando uma B-tree:



- Quando é feita a busca por um predicado, é feita a **busca indexada** da primeira tupla que atende àquele predicado, e depois continua com uma **busca sequencial** até a última tupla que atende ao predicado.

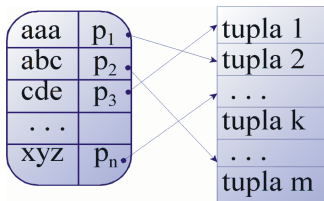


Estruturas de Dados para Índices

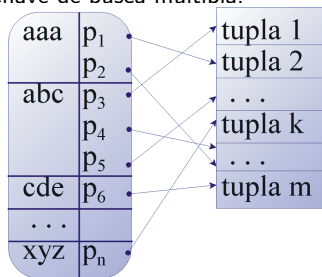


- Pode-se imaginar um índice como uma coleção de pares $\langle \text{Valor}, \text{RowId} \rangle$, onde **Valor** é a chave de busca do índice, e **RowId** é o endereço físico onde a tupla indexada por aquele **Valor** está armazenada. (PostgreSQL chama **RowId** de **TId**: *Tuple Id*)
- A estrutura interna organiza a coleção de pares $\langle \text{Valor}, \text{RowId} \rangle$:

Chave de busca única:



Chave de busca múltipla:



Estruturas de Dados para Índices

Terminologia



- Uma relação é um dado **Dado primário**, que somente pode ser alterado por solicitação explícita do usuário.
- Um índice é um **Dado secundário**, e pode ser apagado e recriado a partir da relação.
- Portanto, índices pode ser criados e apagados a qualquer instante.
- Se uma tupla é inserida, atualizada ou removida na relação, então cada índice existente nessa relação tem que ser atualizado, numa operação chamada **Atualização por Instância**.
- Se um índice é criado (ou re-criado) sobre uma relação já alimentada, o índice é criado numa operação única chamada **Carga rápida** (ou **bulk-load**).
- Se uma relação vai sofrer um grande número de atualizações, em geral é mais barato desligar os índices, fazer as atualização e restaurar os índices, pois uma operação de carga rápida tende a ser bem mais rápida do que fazer muitas atualizações por instâncias.



Estruturas de Dados para Índices

Terminologia



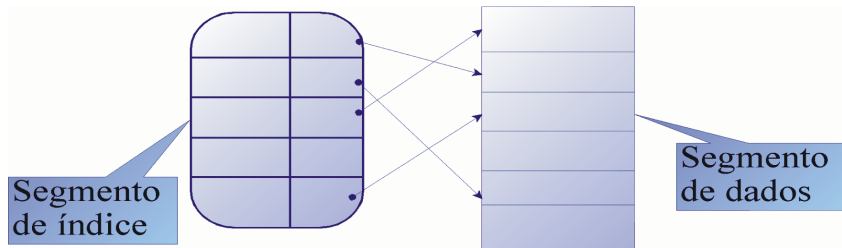
- Cada relação e cada índice é considerado um “bloco de dados”, cada um gerenciado independentemente pelo SGBD.
- Cada relação e cada índice é armazenado num espaço de memória secundária (permanente) chamado **Segmento**.
- Existem diversos tipos de segmentos em um SGBD.
Por exemplo: um **segmento de dados** armazena uma relação,
um **segmento de índice** armazena um índice.
- Cada segmento, independente de seu tipo, tem propriedade diferentes, tais como donos (*owner*) e privilégios de acesso (*access grants*).



Estruturas de Dados para Índices

Terminologia

Além disso, segmentos de dados e de índices têm estruturas de acesso a disco e de armazenagem diferentes (*extent sizes*, *FILL FACTOR*, etc.)



Implicitamente, os registros de dados (tuplas) de uma relação e os registros de seus índices são colocados na mesma unidade de armazenagem, chamada em SQL de **tablespace**, mas o usuário pode (e deve) especificar onde colocar a tabela e onde colocar cada um de seus índices.



Estruturas de Dados para Índices

Terminologia: Índice Denso/Esparso



- Os ponteiros dos índices podem apontar para:

- 1 A tupla e o registro físico onde ela está armazenada: **Índice denso**

`<Valor, RowId=<#RegFis, #Tupla>>`


- 2 Exemplo em PostgreSQL: `SELECT * FROM Aluno WHERE CTID='(0,1)';`

- 3 Apenas para o registro físico, e uma vez acessado, a tupla tem que ser procurada localmente: **Índice esparso**

`<Valor, RowId=<#RegFis>>`

- Relação benefício/custo:

- Índices Densos são mais rápidos para consultas **SELECT**;
- Índices esparsos requerem manutenção mais simples em atualizações **INSERT**, **UPDATE**, **DELETE**.

 Se uma tupla for movida para outro registro físico, todos os índices que apontam para ela precisam ser atualizados. Se ela for movida dentro do mesmo registro, apenas índices densos precisam ser atualizados.

Estruturas de Dados para Índices

Terminologia: Índice Clusterizado



- É possível que **um dos índices de uma relação** defina a ordem de armazenagem das tuplas no meio físico: **Índice clusterizado** (também chamado índice primário);
 - Caso contrário ele é dito ser um **Índice não-clusterizado** (também chamado índice secundário).
 - Só pode existir um por relação.

Índice Clusterizado

É um índice que define a ordem em que as tuplas de uma relação estão fisicamente armazenadas.



Estruturas de Dados para Índices


Terminologia: Índice Clusterizado



- Um índice pode ser usado para *clusterizar* uma tabela
 - **Permanentemente** – qualquer atualização é feita primeiro no índice e daí refletida na tabela, o que pode gerar altos custos de atualização;

ou

- **Momentaneamente** – atualizações subsequentes na tabela podem ir “*desclusterizando*” a ordem (embora um processo de *reclusterização* tenda a ser bem mais rápido).

-  PostgreSQL usa a segunda técnica:

Comando: `CLUSTER <tabela> [USING <indice>]`



Estruturas de Dados para Índices

Terminologia: Índice de Cobertura



- Um índice cobre uma consulta quando todos os atributos indicados na consulta estão armazenados no índice. Ou seja:

Índice de Cobertura

Quando uma consulta puder ser respondida acessando somente o índice, e portanto evitando ter que seguir os ponteiros para obter as tuplas completas, ele é chamado de Índice de Cobertura para aquela consulta.



Estruturas de Dados para Índices

Índice de Cobertura: Exemplo



👉 A seguinte consulta pode ser respondida sem acessar as páginas de dados, usando apenas acesso a um índice de cobertura:

```
SELECT Nome, Idade, Cidade  
FROM Aluno  
WHERE CIDADE='Americana';
```

para o índice criado como:

```
CREATE INDEX IdxCidadeNomeIdade  
ON Aluno (Cidade, Idade, Nome);
```



Estruturas de Dados para Índices

Índice de Cobertura: Exemplo



Já para a consulta:

```
SELECT Nome, Idade, Cidade  
FROM Aluno  
WHERE CIDADE='Americana'  
ORDER BY Nome;
```

o índice seguinte, além de ser de cobertura, pode auxiliar na ordenação:

```
CREATE INDEX IdxCidadeNomeIdade  
ON Aluno (Cidade, Nome, Idade);
```





- Existem cinco tipos de estruturas básicas usadas para índices em SGBDs Relacionais, cada uma podendo prover um caminho de acesso distinto aos dados:
 - 1 Árvores B^+ -tree – também chamados índices ISAM;
 - 2 Estruturas Hash;
 - 3 Índices Bitmap;
 - 4 Arquivos Invertidos;
 - 5 Árvores Multidimensionais.
- Além disso, sempre se pode escolher a **Busca Sequencial** como caminho de acesso. Esta opção sempre existe, o que permite realizar qualquer operação mesmo quando não existe um índice para a auxiliar.



Índices usados em Operadores Relacionais



- Índices podem ser usados nos diversos operadores relacionais, dependendo de seu tipo:

- ★ Unários:

- Seleção: $\sigma_{(C)} R$ ➡ qualquer índice (depende do tipo de consulta e do tipo de dados)
- Agrupamento: $\gamma_{\{A\}} R$ ➡ B-tree, Hash, Bitmap
- Ordenação: $\omega_{\{A\}} R$ ➡ B-tree, Arquivos invertidos
- Eliminação de Duplicatas: $\tau(R)$ ➡ B-tree, Hash

- ★ Binários:

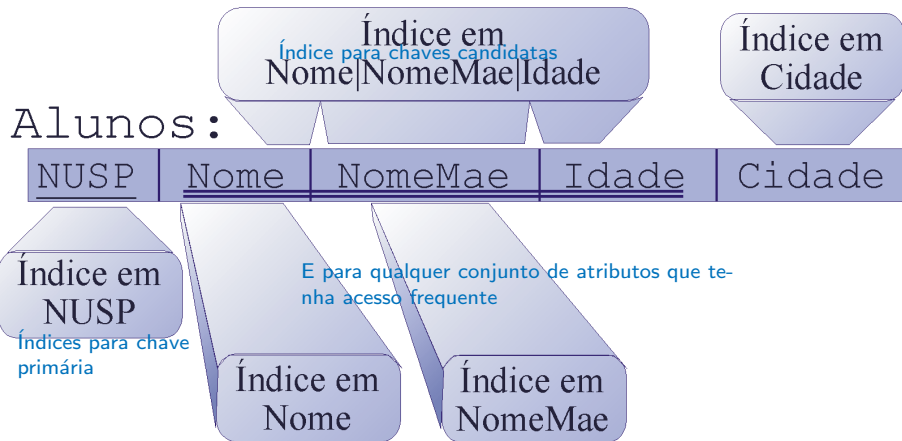
- Junção: $R_1 \overset{\theta}{\bowtie} R_2$ ➡ B-tree, Hash
- Operadores de Conjunto: $R_1 \cup R_2$, $R_1 \cap R_2$ e $R_1 - R_2$ ➡ B-tree, Hash



Declaração de Índices em SQL



Podem ser criados muitos índices sobre uma mesma relação:



Roteiro



1 Conceitos

2 Declaração de Índices em SQL

3 Táticas para o uso de índices

- Indexando expressões
- Índices Parciais
- índices Compostos
- Índices com atributos incluídos
- Índices UNIQUE
- Índices clusterizados
- Índices B-tree e cláusula ORDER BY
- Exemplo: Relação ExamLabs da Base USP-FAPESP-Covid-19

4 Gerenciando e mantendo índices





Embora existam muitos conceitos em comum para o tratamento de índices nos diversos SGBDs, os recursos e variações que cada um oferece variam bastante, em termos de:

- Tipos de estruturas de dados usadas
 - Todos dispõem de B^+ -tree – compara e ordena por qualquer comparador;
 - *Hash* é muito usado para organizar uma tabela (índices *clusterizados* ou índices de cobertura) – só compara por identidade;
 - *BitMap* é usado em relações quase-estáticas, ou para conjugar índices;
 - Índices Espaciais e outros índices para aplicações especiais (p. ex: índices invertidos).
- Variações da aplicação dos índices:
 - ☞ Únicos, densos, esparsos, parciais, *clustered*, Compostos, etc.



Indexando expressões

Conceitos



- Pode-se indexar as expressões que são usadas com frequência nas consultas;
- É útil para indexar expressões que comparam os atributos usados na consulta com expressões dos dados armazenados.
- Note-se que o otimizador de um SGBD compara apenas expressões que são idênticas na consulta e na definição do índice.



Indexando expressões

Exemplos

Comparar um texto de maneira não-sensível à caixa das letras:

```
CREATE INDEX IdxUpNome ON Professor(Upper(Nome));
```

-- Esse índice é usado na seguinte consulta:

```
SELECT *  
FROM Professor  
WHERE Upper(Nome)='RAIMUNDO';
```

-- Mas não é usado na seguinte consulta:

```
SELECT *  
FROM Professor  
WHERE Nome='RAIMUNDO';
```



Indexando expressões

Exemplos



Encontrar as **Matriculas** aceitas em determinada data, a qual foi armazenada no atributo **Aceita** de tipo **datetime**, mas comparando apenas a data:

```
CREATE INDEX IdxDiaMatricula ON  
    Matricula(Date(Aceita));
```

-- Esse índice é usado na seguinte consulta:

```
SELECT *  
    FROM Matricula  
    WHERE Date(Aceita)=Date('2016-03-01');
```



Indexando expressões

Exemplo



Encontrar os alunos com Média superior a 5,0:

```
CREATE INDEX IdxMedia ON Matricula  
((NotaP1+NotaP2)/2);
```

Esse índice será usado em comandos de seleção tais como:

```
SELECT NUSP, Sigla, (NotaP1+NotaP2)/2 AS Media  
FROM Matricula  
WHERE (NotaP1+NotaP2)/2 > 5,0  
ORDER BY NUSP;
```





- Um Índice Parcial indexa apenas parte dos dados armazenados na tabela: apenas as tuplas em que os **atributos de partição** usados na cláusula **WHERE** do índice atendem à condição especificada;
- A ideia é tornar mais eficiente a busca na porção dos dados usada com frequência, reduzindo o tamanho do índice para indexar apenas aquela porção;
- Um índice menor ocupa menos espaço e é mais rápido para atualizar e consultar;
- Índices Parciais podem também ajudar o acesso concorrente quando índices parciais ou mesmo o índice completo estiverem sendo atualizados ou (re-)criados.



Índices Parciais

Exemplo

Se a relação Professor tiver um atributo que indica se o professor está aposentado:

```
Professor=(Nome, Idade, Nivel, NNFunc, Aposentado)
```

```
CREATE INDEX IdxNomeAtivo ON Professor(Nome)
      WHERE Aposentado='Nao';
```

-- Esse índice é usado nas seguintes consultas:

```
SELECT Count(*)
      FROM Professor
      WHERE Aposentado='Nao';
```

```
SELECT *
      FROM Professor
      WHERE Nome='Raimundo' AND Aposentado='Nao';
```

-- Mas não em:

```
SELECT *
      FROM Professor
      WHERE Nome='Raimundo';
```



Indexando mais de um atributo

Conceitos



- Em geral, um índice permite indexar um ou mais atributos.
- Quando mais de um atributo é indexado no mesmo índice, a chave de busca é a concatenação dos diversos atributos.
- ★ por isso, tais índices são chamados **índices multi-atributos**, **índices concatenados** ou **índices Compostos**.
- Por exemplo, assuma que é criado o índice:

```
CREATE INDEX IdxNomeIdade ON Professor(Nome, Idade);
```

- Então a consulta

```
SELECT *  
FROM Professor  
WHERE Nome='Raimundo' AND Idade=25;
```

terá como chave de busca o valor 'Raimundo0025'.

(Assumindo Idade como `SmallInt` ∴ 2 bytes)



Indexando mais de um atributo

Conceitos

- Veja que um índice que indexa dois atributos é diferente de dois índices indexando um atributo cada um:

```
CREATE INDEX IdxNome ON Professor(Nome);
```

```
CREATE INDEX IdxIdade ON Professor(Idade);
```

- Nesse caso, se a consulta:

```
SELECT *  
FROM Professor  
WHERE Nome='Raimundo' AND Idade=25;
```

usar esses índices, as chaves de busca terão os valores 'Raimundo' e '0025', respectivamente.

- Se os três índices estiverem definidos, o otimizador do SGBD escolhe o plano de execução que usa `IdxNomeIdade` ou o plano que usa `IdxNome` e `IdxIdade` e faz a intersecção das tuplas indicadas por cada índice.

☞ Será usado o plano que for estimado ser o mais rápido (provavelmente o que usa `IdxNomeIdade`, porque ele é mais seletivo).



Indexando mais de um atributo

Conceitos



- Um índice somente pode ser usado se algum(ns) atributo que ele indexa estiver presente nas condições da consulta.
- Uma condição na cláusula **WHERE** que usa o atributo a_i pode se beneficiar de um índice se:
 - 1 a_i é o primeiro atributo do índice;
 - 2 ou todos os atributos anteriores a a_i nesse índice também ocorrem na consulta;
 - 3 e a condição restringe o resultado a uma faixa continua de valores no índice (por exemplo, não pode usar o operador \neq).
- Portanto índices com mais atributos são mais seletivos (e recuperam os dados mais rapidamente) mas podem ser usados em menos consultas.
- Veja que **a ordem dos atributos na consulta é irrelevante**, mas **a ordem dos atributos no índice é importante**.



Indexando mais de um atributo

Exemplos

Por exemplo, assuma que existem os 3 índices sobre a relação de professores:

```
CREATE INDEX IdxNome ON Professor(Nome);  
CREATE INDEX IdxIdade ON Professor(Idade);  
CREATE INDEX IdxNomeIdade ON Professor(Nome, Idade);
```

Quais deles são usados nas relações seguintes?

```
SELECT *  
FROM Professor  
WHERE Nome='Raimundo' AND Idade=25;
```

-- Pode usar só **IdxNomeIdade** ou **IdxNome** junto com **IdxIdade** fazendo a intersecção entre eles, mas **IdxNomeIdade** é mais seletivo

```
SELECT *  
FROM Professor  
WHERE Idade=25 AND Nome='Raimundo';
```

-- Essa consulta é analisada e executada exatamente da mesma maneira que a anterior



Indexando mais de um atributo

Exemplos

```
SELECT * FROM Professor  
WHERE Nome='Raimundo' AND Cidade='Ibate';
```

-- Pode usar só **IdxNomeIdade** ou só **IdxNome**, e ambos têm a mesma seletividade (portanto **IdxNome** é redundante se não houver outra consulta que o justifique)

```
SELECT * FROM Professor  
WHERE Cidade='Ibate' AND Idade=25;
```

-- Só **IdxIdade** pode ser usado

```
SELECT * FROM Professor  
WHERE Nome<>'Raimundo' AND Idade>30;
```

-- Só **IdxIdade** pode ser usado

```
SELECT * FROM Professor  
WHERE Nome<>'Raimundo' AND Cidade='Araras';
```

-- Nenhum pode ser usado



Índices com atributos **incluídos**

Conceitos



- Atributos **incluídos** são acrescentados ao registro de dados do índice sem fazer parte da chave de busca:

Chave_de_busca Atributo_incluido

- Exemplo:

```
CREATE UNIQUE INDEX IdxUnNUSP ON Aluno(NUSP)  
INCLUDE Nome;
```

- ★ Este índice cria uma B^+ -tree sobre o atributo **NUSP** e acrescenta o atributo **Nome** nas folhas.
- ★ Ele obriga que o atributo **NUSP** não tenha valores repetidos (**UNIQUE**).



Índices com atributos **incluídos**

Conceitos



- ➡ Atributos incluídos não são considerados para garantir unicidade da chave;
 - Somente índices B^+ -tree e *hash* têm implementações para atributos **incluídos**:
 - B^+ -tree – Atributos incluídos são colocados na folha da B^+ -tree, mas não nos nós-diretório, assim, afetam pouco o tamanho interno da estrutura e não servem para a busca;
 - Hash* – Atributos incluídos são colocados nos *buckets* de dados, mas não são usados na função de *hash*.
 - A grande vantagem de índices com atributos incluídos é serem usados como **índice de cobertura** em consultas *index only scan*.



Índices com atributos **incluídos**

Exemplo



- Assuma que é frequente recuperar o nome do aluno pelo número USP.
- Além disso, deve existir um índice **UNIQUE** sobre o **NUSP** para garantir a restrição de integridade de chave da entidade.

Se o índice for criado como:

```
CREATE UNIQUE INDEX IdxUnNUSP ON Aluno(NUSP);
```

ele usa uma B^+ -tree sobre o atributo **NUSP**. Então a consulta

```
SELECT Nome  
FROM Aluno  
WHERE NUSP=1234321;
```

precisa:

- 1 acessar o índice para recuperar o ponteiro para a tupla na tabela,
- 2 acessar a tabela para recuperar o nome solicitado.



Índices com atributos **incluídos**

Exemplo

Porém, se o índice for criado como:

```
CREATE UNIQUE INDEX IdxUnNUSP ON Aluno(NUSP)  
INCLUDE(Nome);
```

cria-se uma B^+ -tree sobre o atributo **NUSP** acrescentando o atributo **Nome** ao campo de dados das folhas.

Agora, a mesma consulta

```
SELECT Nome  
FROM Aluno  
WHERE NUSP=1234321;
```

já obtém o **Nome** do aluno na folha da B^+ -tree e sem acessar a tabela.

O índice se torna um **índice de cobertura** para essa consulta, e a execução da consulta pode ser feita com um acesso *index only*.



Índices com atributos **incluídos**

Índice Composto ou índice com atributos incluídos?



- Uma decisão de projeto a ser tomada é escolher entre Índice Composto ou incluídos. Ou seja, qual opção é melhor entre:

```
CREATE INDEX Idx1 ON Tab1 (Atr1, Atr2);
```

ou

```
CREATE INDEX idx1 ON Tab1 (Atr1) INCLUDE (Atr2);
```

- Algumas regras básicas são:
 - Atributos incluídos devem aparecer apenas na cláusula **SELECT**, e não nas cláusulas **JOIN**, **WHERE**, **GROUP BY**, **ORDER BY** das consultas;
 - Quando atributos são usados em condições, é geralmente melhor que estejam na chave do índice, a menos que a condição gere uma faixa não-continua de valores no índice.



Índices com atributos **incluídos**

Índice Composto ou índice com atributos incluídos? – Exemplo

Por exemplo, obter os alunos matriculados em disciplinas do SCC nos últimos 5 anos:

```
Matricula=(Nusp, Sigla, Ano, Semestre, NotaP1, NotaP2, Freq, Registro, Aceito)
```

```
SELECT NUSP  
FROM Matricula  
WHERE Sigla LIKE 'SCC-%' AND Ano > 2011;
```

Um índice composto

```
CREATE INDEX IdxMatrSiglaAno ON Matrícula (Sigla, Ano)  
INCLUDE (NUSP);
```

irá gerar uma lista não contínua das **matrículas** separadas por **sigla**, com faixas separadas de **matrículas** que têm os **anos** solicitados para cada **sigla**,
👉 e portanto a ordenação dada pelo índice não será usada.




Índices com atributos **incluídos**

Índice Composto ou índice com atributos incluídos? – Exemplo

Nesse caso, é melhor que o atributo **Ano** (ou **Sigla**) seja colocado como atributo incluído, pois o índice pode ser usado para localizar a faixa contínua da **Sigla** (ou **Ano**) e o atributo **Ano** (ou **Sigla**) é usado apenas como atributo numa condição de filtro:

```
CREATE INDEX IdxMatrSiglaAno ON Matrícula (Sigla)
                                     INCLUDE (Ano, NUSP);
```

 O SGBD pode usar o índice da seguinte maneira:

- Usa o atributo **Sigla** para a busca, obtendo a lista de todas as matrículas feitas em disciplinas do SCC;
- Usa o atributo **Ano** para filtrar sequencialmente apenas as matrículas da lista que têm **Ano>2011**;
- Usa o atributo **NUSP** para completar os atributos necessários na resposta e ele se torna um índice de cobertura para a consulta.





- Índices **UNIQUE** servem a dois propósitos:
 - Integridade de dados** – Garante que a relação não terá duas tuplas com os mesmos valores nos atributos indexados;
 - Eficiência de acesso** – Acessa os poucos dados que atendem a um predicado sobre os atributos indexados evitando a varredura sequencial.



Índices UNIQUE


Conceitos – Integridade de dados



- Existe pouca diferença entre definir uma restrição de integridade **UNIQUE** ou criar um índice **UNIQUE** sobre os mesmos atributos.
- De fato, o índice é a maneira de “implementar” a restrição.





- Em  PostgreSQL, é possível associar um índice criado explicitamente como o índice usado para validar uma restrição:

```
ALTER TABLE <Tabela>  
    ADD [UNIQUE | PRIMARY KEY] USING INDEX <Índice>;
```

Todos os atributos não incluídos serão considerados na restrição.

- Por exemplo:

```
CREATE UNIQUE INDEX IdxPKIncAluno ON Aluno (NUSP)  
    INCLUDE (NOME);  
ALTER TABLE <Tabela>  
    ADD [UNIQUE | PRIMARY KEY] USING INDEX <índice>;
```



cria a restrição que **NUSP** é chave primária de **Aluno**, mas acrescentando o nome como atributo **incluído** no índice de validação, sem ser parte da restrição.



Índices UNIQUE

Exemplos

- Um índice parcial pode ser **UNIQUE** sobre os mesmos atributos que repetem quando se considera a relação inteira.
- Por exemplo, alunos podem se matricular mais de uma vez numa mesma disciplina se não forem aprovados, mas não num mesmo ano/semestre.

```
Matricula=(Nusp, Sigla, Ano, Semestre, NotaP1, NotaP2, Freq, Registro, Aceito)
```

```
CREATE INDEX IdxNuspSigla ON Matricula(NUSP,Sigla);  
CREATE UNIQUE INDEX IdxNuspSiglaCorrente  
ON Matricula(NUSP,Sigla)  
WHERE Year(Aceito)=2022 AND Semestre=1;
```

- A condição da cláusula **WHERE** do índice tem que ser constante.



Por exemplo, não pode ser indicada a condição `Year(Aceito)=Year(SYSDATE)`, já que **SYSDATE** é uma variável.



Índices *clusterizados*

Conceitos

- Índices *clusterizados* organizam a ordem de gravação das tuplas nas páginas de dados da tabela, para que fiquem na mesma ordem das respectivas chave de acesso no índice.
- Nessa organização se diz que o índice é **primário**, e a organização da tabela é com alocação primária.

☞ Caso contrário a organização da tabela é dita ser *em heap*.

- Acessar uma, ou poucas tuplas de um *heap* usando os ponteiros de um índice não é problema;
- mas para acessar muitas tuplas, a falta de sincronismo entre a sequência do índice e a sequência da tabela causa uma **“chuva de ponteiros”**, que é muito custosa para acessos a disco;

☞ A organização em **CLUSTER** faz com que depois de acessada a primeira tupla, muitas das próximas tuplas estarão no mesmo registro já acessado.



Índices *clusterizados*

Exemplo

Assuma que é frequente recuperar os alunos de uma cidade, que existem em média 50 alunos de cada cidade e foi criado o índice:

```
CREATE INDEX IdxCidade ON Aluno(Cidade);
```

Então a consulta:

```
SELECT Nome, Numero  
FROM Aluno  
WHERE Cidade='Campinas';
```

- recupera os ponteiros para todas as tuplas necessárias com um ou dois acessos a disco nas folhas da B^+ -tree,
- mas causa uma chuva de 50 acessos em média nos registros na tabela (a 10ms por leitura em disco, é mais de meio segundo para responder as consultas).



Índices *clusterizados*

Exemplo

Se uma tabela for clusterizada:

```
CLUSTER Aluno USING IdxCidade;
```

Então a mesma consulta:

```
SELECT Nome, Numero  
FROM Aluno  
WHERE Cidade='Campinas';
```

- 👉 recupera os ponteiros para todas as tuplas necessárias com um ou dois acessos a disco nas folhas da B^+ -tree,
- 👉 e como as tuplas estão na mesma ordem na tabela, recupera todas com apenas dois ou três acessos a disco
(a 10ms por leitura em disco, fica em torno de 50ms).



Índices B^+ -tree e a cláusula `ORDER BY`

Conceitos



- B^+ -tree é a estrutura de indexação que permite manter os atributos indexados ordenados;
- Por *default*, a ordem do índice é ascendente para todos os atributos, e nulos são colocados no final da lista ordenada;
- Para a resolução de consultas, a ordem em geral não é importante, especialmente para índices sobre apenas um atributo.
- Porém, quando o índice é usado em comandos `ORDER BY` ou existem mais de um atributo indexado cuja ordem relativa é importante, ou quando a posição dos nulos é importante, então é importante considerar a ordem do índice.



Índices B^+ -tree e a cláusula ORDER BY

Exemplo

Considere listar as matrículas dos alunos começando pelo ano mais recente, mantendo o primeiro semestre primeiro, e listar as disciplinas pela ordem decrescente da sigla:

```
SELECT NUSP, Sigla, Ano, Semestre  
FROM Matricula  
ORDER BY Ano DESC, Semestre ASC, Sigla DESC;
```

Se não houver um índice sobre os atributos usados no **ORDER BY**, então deve ser feita uma varredura sequencial de toda a tabela e a seguir uma ordenação em tempo de execução da consulta, o que pode ser demorado.

Isso pode ser muito acelerado com o índice:

```
CREATE INDEX IdxAnoSemSigNUSPDAD  
ON Matricula(Ano DESC, Semestre ASC, Sigla DESC)  
INCLUDE NUSP;
```



Índices B^+ -tree e a cláusula ORDER BY

Exemplo-cont.



- Colocando **NUSP** como atributo incluído, a consulta pode ser respondida sem acessar a tabela, usando o índice como um índice de cobertura.
- Especificamente para a ordenação, mesmo que **NUSP** não seja incluído, os atributos indexados suprem a ordenação necessária.

-
- Veja que o seguinte índice não pode ser usado para a ordenação:

```
CREATE INDEX IdxAnoSemSigNUSP
ON Matricula(Ano, Semestre, Sigla)
INCLUDE NUSP;
```



Embora índices possam ser percorridos em qualquer ordem, o atributo **Semestre** está em ordem reversa dos demais, portanto “quebra” a sequência de resultados do índice, e impede seu uso.



Exemplo: Relação ExamLabs



- Vamos considerar a tabela `ExamLabs` do esquema `D2`, que inclui os hospitais Beneficência Portuguesa e Sirio-Libanês, totalizando 6.803.127 tuplas de exames. (embora não seja *big*, já é suficiente para perceber boas diferenças de tempo.)
- Como ocorreu frequentemente em aulas e exercícios anteriores, as análises frequentemente requerem montar tabelas temporárias com as medidas de diversos analitos em cada tupla.



Agilizar a montagem dessas tabelas é um alvo interessante.

- Quais são os principais tipos de exames registrados?

```
SELECT DE_Exame, Count(*)  
FROM ExamLabs  
GROUP BY 1  
ORDER BY 2 DESC  
LIMIT 20;
```

Total query runtime: 907 msec.
20 rows affected.

então selecionamos as classes principais:

`Plasma`, `Hemograma` e `Gasometria`.

Digamos que estamos interessados também em `Covid` e `Colesterol`.



Exemplo: Relação ExamLabs



- Vamos acrescentar um atributo à tabela, para indicar a classe de cada exame:
usar o tipo **INTEGER** como uma representação binária da classe:

```
ALTER TABLE ExamLabs ADD COLUMN TP_Exame INTEGER;

UPDATE ExamLabs
  SET TP_Exame=(De_Exame ~* 'colest')::INT *      b'000001'::INT +
              (De_Exame ~* 'hemograma')::INT *    b'000010'::INT +
              (De_Exame ~* 'plasma')::INT *        b'000100'::INT +
              (De_Exame ~* '(covid)|(sars.cov.2)')::INT * b'001000'::INT +
              (De_Exame ~* 'gasometria')::INT *    b'010000'::INT;

UPDATE 6.803.127 Tuples.  Query returned successfully in 2 min 39 secs.
```



Exemplo: Relação ExamLabs



- Vamos contabilizar a quantidade de tuplas com cada tipo de exame, e verificar se existe alguma tupla que seja de dois tipos:

```
SELECT Count(*) FILTER (WHERE TP_Exame&b'000001'::INT!=0) AS Colest,  
       Count(*) FILTER (WHERE TP_Exame&b'000010'::INT!=0) AS Hemograma,  
       Count(*) FILTER (WHERE TP_Exame&b'000100'::INT!=0) AS Plasma,  
       Count(*) FILTER (WHERE TP_Exame&b'001000'::INT!=0) AS Covid,  
       Count(*) FILTER (WHERE TP_Exame&b'010000'::INT!=0) AS gasometria,  
       Count(*) FILTER (WHERE TP_Exame=0) AS Outros
```

FROM ExamLabs;

Total query runtime: 44 secs 124 msec.
1 rows affected.

Colest	Plasma	Hemograma	Covid	Gasometria	Outros
49.085	3.156.949	1.352.954	71.554	787.932	1.384.653

```
SELECT *, (TP_Exame&1)      + (TP_Exame>>1)&1 + (TP_Exame>>2)&1 +  
          (TP_Exame>>3)&1 + (TP_Exame>>4)&1 + (TP_Exame>>5)&1  
FROM ExamLabs  
WHERE ((TP_Exame&1)      + (TP_Exame>>1)&1 + (TP_Exame>>2)&1 +  
       (TP_Exame>>3)&1 + (TP_Exame>>4)&1 + (TP_Exame>>5)&1 ) >1  
LIMIT 100;
```

Successfully run. Total query runtime: 1 secs 987 msec.
0 rows affected.



Exemplo: Relação ExamLabs



- Vamos medir o tempo de execução de uma consulta que recupera todos os exames de `Colesterol`, como antes:

```
SELECT DE_Exame, DE_Analito, DE_Resultado
FROM ExamLabs
WHERE de_exame ~* 'Colest';
```

Successfully run. Total query runtime: 2 secs 597 msec.
49085 rows affected.

- E se usarmos o novo atributo (evita ter que processar o padrão de expressão regular nas 6,8 MTuplas):

```
SELECT DE_Exame, DE_Analito, DE_Resultado
FROM ExamLabs
WHERE TP_Exame=1;
```

Successfully run. Total query runtime: 736 msec.
49085 rows affected.

reduz para menos de
1/3 do tempo.



Exemplo: Relação ExamLabs



- Vamos criar um índice adequado para essa consulta:
- Pode ser criado um índice
 - parcial,
 - incluindo atributos que permitam executar consultas por cobertura:

```
CREATE INDEX ExLab_DeColestPDeAnalitoDeResult  
ON ExamLabs(DE_Exame)  
INCLUDE (DE_Analito, DE_Resultado)  
WHERE TP_Exame=1;
```

Query returned successfully in 3 secs 782 msec.

- Agora, re-executando a consulta:

```
SELECT DE_Exame, DE_Analito, DE_Resultado  
FROM ExamLabs  
WHERE TP_Exame=1;
```

Successfully run. Total query runtime: 96 msec.
49085 rows affected.

ela executa 26 vezes mais rápido do que antes.



Exemplo: Relação ExamLabs



- Vamos criar um índice equivalente para exames de [hemograma](#)
 - isso envolve um tipo de exames com $\approx 20\%$ do total de tuplas:

```
CREATE INDEX ExLab_DeHemogramaPDeAnalitoDeResult  
ON ExamLabs(DE_Exame)  
INCLUDE (DE_Analito, DE_Resultado)  
WHERE TP_Exame=2
```

Query returned successfully in 1 secs 92 msec.



Exemplo: Relação ExamLabs



- Veja que, se não for indicada a condição `TP_Exame=2` numa consulta, então o índice não pode ser usado:

```
SELECT DE_Exame, DE_Analito, DE_Resultado
FROM ExamLabs
WHERE DE_Exame='Hemograma, sangue total' AND
      DE_Analito ~* 'bastonetes';
```

Successfully run. Total query runtime: 3 secs 414 msec.
46 rows affected.

- usando a condição, o índice passa a ser usado numa consulta por cobertura:

```
SELECT DE_Exame, DE_Analito, DE_Resultado
FROM ExamLabs
WHERE TP_Exame=2 AND
      DE_Exame='Hemograma, sangue total' AND
      DE_Analito ~* 'bastonetes';
```

Successfully run. Total query runtime: 77 msec.
46 rows affected.

46 vezes mais
rápido



Exemplo: Relação ExamLabs



- Vamos trabalhar com a base completa de **Todos** os hospitais:

```
SET SEARCH_Path to Todos, public;
```

```
SELECT COUNT(*) FROM ExamLabs;
```

Successfully run. Total query runtime: 1 secs 146 msec.
1 rows affected.

Count
31.991.313

- Repentem-se os mesmos passos:
 - Acrescentar o atributo **TP_EXame**;
 - Criar o índice;
 - Executar as consultas:

👉 Sem usar o índice de cobertura: 15 secs 179 msec.

👉 Usando o índice de cobertura: 117 msec.

★ $\frac{15.179}{117} \approx 130$ vezes mais rápido

- De fato, quanto maior a cardinalidade do conjunto de dados, **maior é a**
efetividade do índice !



Exemplo: Relação ExamLabs

Conclusões



- A estrutura de recuperação de dados em um SGBD oferece diversas alternativas para agilizar uma consulta .
 - Em sua maioria, a técnica fundamental é criar: uma arquitetura de estruturas de dados para o acesso aos dados, para gerar um “caminho de acesso” que permita recuperar os dados de interesse mais rapidamente.
 - Os SGBDs disponibilizam recursos para que essa arquitetura represente a lógica da solução com um alto nível de abstração .
 - No exemplo mostrado aqui, foram usados recursos para
 - Disponibilizar juntos os dados que uma consulta precisa – permitindo consultas de cobertura, e
 - Índices parciais – que essencialmente particionam os índices.
 - O passo seguinte é particionar os próprios dados . `(CREATE TABLE ...
INHERITS ...;)`
- ★ Mas isso é assunto para estudos mais profundos sobre o tema.



Roteiro



- 1 Conceitos
- 2 Declaração de Índices em SQL
- 3 Táticas para o uso de índices
- 4 Gerenciando e mantendo índices
 - Solicitar informações sobre o plano de consulta
 - Conclusões





- O padrão SQL não tem comandos que se referem aos planos de consulta. No entanto, tal como ocorre com índices, existe um “padrão aproximado” para:
 - Solicitar ao gerenciador informações sobre o plano de consulta gerado/executado para uma consulta;
 - Influenciar o gerenciador para escolher determinados operadores de acesso físico.
- Vale notar que que esses recursos têm uma sintaxe geral atendida por todos os SGBDR, mas a sintaxe detalhada é completamente dependente de cada produto, versão, etc.



Solicitar informações sobre os planos de consulta



Usualmente o comando `EXPLAIN query` está disponível para o analista verificar como cada comando SQL é transformado no plano físico de acesso. A sintaxe geral é

EXPLAIN query

```
EXPLAIN [ANALYZE] <comando>
```

- Onde `<comando>` pode ser qualquer comando da DML:
`SELECT`, `INSERT`, `UPDATE` ou `DELETE`.
- A maneira como o plano é mostrado depende do gerenciador:
 - pode ser listado na console,
 - armazenado em uma relação pré-formatada, onde os atributos são a indicação do operador, seus parâmetros e estimativas, ou
 - gerar um documento de texto como uma *string* em formato `JSON` ou `XML`.



Solicitar informações sobre os planos de consulta



- **EXPLAIN** mostra o plano gerado em uma linha para cada operador físico, junto com as respectivas estimativas. Em geral se mostra:
 - Custo (tempo) de execução,
 - Uso de memória em bytes,
 - Número de tuplas recuperadas, etc.
- O custo de execução é mostrado numa unidade arbitrária: serve apenas para ter uma ideia da proporção de custo esperada entre os diversos operadores ou entre planos distintos gerados;
- No jargão do **IBM DB2**, diz-se que o custo é expresso em unidades "*Timeron*". Um *Timeron* não pode ser convertido para uma unidade de tempo real.



Solicitar informações sobre os planos de consulta




- O comando `EXPLAIN <comando>;` só gera o plano, mas não executa o `<comando>`.
- Se for solicitado `EXPLAIN ANALYZE <comando>;` então o `<comando>` é executado e se mostra também o valor real obtido pela execução, junto com as estimativas iniciais.
- ☞ Note-se que no caso de `<comando>` ser `INSERT`, `UPDATE` ou `DELETE`, as atualizações são efetivamente realizadas na base de dados.
- Isso pode ser evitando usando:

```
BEGIN;  
    EXPLAIN ANALYZE ...;  
ROLLBACK;
```



Solicitar informações sobre os planos de consulta



 PostgreSQL provê o comando `EXPLAIN query` para verificar como cada comando SQL é transformado no plano físico de acesso.

EXPLAIN query – PostgreSQL

```
EXPLAIN [ANALYZE] [VERBOSE] <comando>  
ou EXPLAIN (<param>[, ...]) <comando>
```

onde <param> é um de:

ANALYZE [boolean]

VERBOSE [boolean]

COSTS [boolean]

BUFFERS [boolean]

TIMING [boolean]

FORMAT {TEXT|XML|JSON|YAML}

- O resultado é mostrado na console.



Solicitar informações sobre os planos de consulta



Por exemplo:

```
EXPLAIN SELECT * FROM Alunos;
```

QUERY PLAN

```
Seq Scan on alunos (cost=0.00..1529.00 rows=80000 width=39)
```

- Este resultado indica que o plano é constituído apenas do método 'Seq Scan' sobre a relação `Alunos`,
- não precisa esperar nada `0.00` para começar a obter resultados, mas custa `1529` unidades de tempo para executar tudo,
- e estima-se que serão obtidas `80.000` tuplas com média de `39` bytes cada.

Solicitar informações sobre os planos de consulta



Por exemplo:

```
EXPLAIN (ANALYZE on, VERBOSE off, COSTS on,  
        BUFFERS off, TIMING on)  
SELECT * FROM Alunos;
```

QUERY PLAN

```
-----  
Seq Scan on alunos (cost=0.00..1529.00 rows=80000 width=39)  
    (actual time=0.025..14.185 rows=80000 loops=1)  
Planning time:  0.138 ms  
Execution time: 17.986 ms
```

Planning time: gastou 0.138 ms para compilar e otimizar o plano físico,

Execution time: gastou 17.986 ms para executar o plano.



Solicitar informações sobre os planos de consulta



Por exemplo:

```
EXPLAIN ANALYZE SELECT * FROM Alunos  
WHERE Nome='José da Silva';
```

QUERY PLAN

```
-----  
Seq Scan on alunos (cost=0.00..1729.00 rows=5 width=39)  
    (actual time=23.904..23.907 rows=10 loops=1)  
    Filter: ((nome)::text = 'José da Silva'::text)  
    Rows Removed by Filter: 79990  
Planning time: 0.290 ms  
Execution time: 23.972 ms
```

Solicitar informações sobre os planos de consulta



Continuando o exemplo:

```
CREATE INDEX IX_Alunos_Nome ON Alunos(Nome);  
EXPLAIN ANALYZE SELECT * FROM Alunos  
  WHERE Nome='José Lucena';
```

QUERY PLAN

Index Scan using ix_alunos_nome

on alunos (cost=0.42..8.44 rows=1 width=39)

(actual time=0.139..0.140 rows=1 loops=1)

Index Cond: ((nome)::text = 'José Lucena'::text)

Planning time: 0.187 ms

Execution time: 0.107 ms

Solicitar informações sobre os planos de consulta



Continuando o exemplo:

```
-CREATE INDEX IX_Alunos_Nome ON Alunos(Nome);  
EXPLAIN ANALYZE SELECT * FROM Alunos  
  WHERE Nome='José da Silva';
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on alunos (cost=4.46..23.28 rows=5 width=39)  
    (actual time=0.136..0.138 rows=10 loops=1)  
    Recheck Cond: ((nome)::text = 'José da Silva'::text)  
    Heap Blocks:  exact=1  
    -> Bitmap Index Scan on alunos_pk (cost=0.00..4.46 rows=5 width=0)  
        (actual time=0.117..0.117 rows=10 loops=1)  
        Index Cond: ((nome)::text = 'José da Silva'::text)  
Planning time:  0.295 ms  
Execution time: 0.255 ms
```



- Criar um índice requisita um travamento da relação enquanto o item é criado.
- Diversos SGBDs permitem solicitar travamento do índice apenas para operações de escrita e liberar leitura concorrente.



PostgreSQL

– Permite declarar a criação de um índice concorrentemente: `CREATE INDEX CONCURRENTLY ...`. A criação é mais lenta mas não trava a tabela.



Gerenciando e mantendo índices

Conclusões

- É sempre uma boa ideia testar as ações de manutenção de um SGBD numa **base de teste** antes de aplicá-las numa **base de produção**.



No caso de índices, a decisão final por alguma alteração deve ser feita depois de confirmada a adequação na **base de produção**.

Isso porque:

- O desempenho de um índice depende da **carga de dados** real (amostras reduzidas de dados distorcem o comportamento);
- Ter um índice escalonado para uso numa consulta depende da **seletividade de todos os índices** das tabelas envolvidas estarem corretamente coletadas (se alguma seletividade é estimada muito diferente da real, o modelo de custo de cada consulta é afetado);
- A relação benefício/custo entre agilizar consultas mas demorar nas atualizações depende da **carga de consultas** total (agilizar uma consulta vezes quantas vezes a consulta é usada versus demorar todas as consultas que atualizam uma tabela vezes quantas atualizações ocorrem).



Gerenciando e mantendo índices

Conclusões

- Índices agilizam (muito) a execução de operações de **consulta**, o que inclui **SELECT**, **UPDATE**, **DELETE** e **INSERT**;
- embora em cada consulta somente sejam úteis os índices que envolvem os atributos usados na consulta.
- Mas índices atrapalham (um pouco) a execução de operações de **atualização**, o que inclui **UPDATE**, **DELETE** e **INSERT**;
- e isso inclui todos os índices, mesmo que não estejam explicitamente usados na consulta.

Portanto, deve-se ser judicioso para somente criar índices que ajudem na **carga total de consultas** de todo um modo de operação da empresa.



MBA em Inteligência Artificial e Big Data
– Curso 3: Administração de Dados Complexos em Larga Escala –

★ Índices em Bases de dados ★

Caetano Traina Júnior

Grupo de Bases de Dados e Imagens – GBdI
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos

Índices
FIM

