

SGBDs Relacionais

– Representação de Dados em Três Níveis Conceituais

Prof. Dr. Ives Renê V. Pola

ivesr@utfpr.edu.br

Departamento Acadêmico de Informática – DAINF
UTFPR – Pato Branco DAINF
UTFPR
Pato Branco - PR

Esta apresentação mostra o conceito da Representação de Dados em Três Níveis conceituais: externo (visões), interno (conceitual) e físico (implementação), e sua aplicação pelos SGBDs Relacionais.



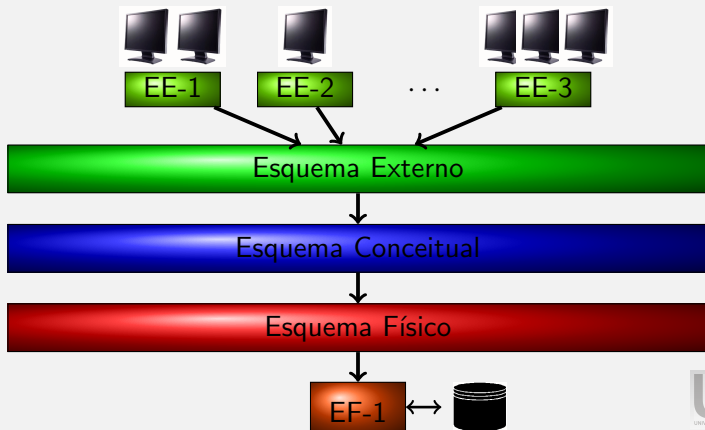
Roteiro

- 1 Conceitos e Motivação
- 2 O Esquema Físico em SQL
- 3 Esquemas Externos no Modelo Relacional
- 4 Visões
- 5 Triggers
- 6 Conclusão

Conceitos e Motivação




O modelo de representação de dados em três níveis

- Um SGBD Relacional é baseado na ideia da modelagem baseada em três níveis conceituais, para garantir a **Integração** e a **Independência de Dados** da aplicação quanto à armazenagem dos dados:



Conceitos e Motivação

O modelo de representação de dados em três níveis

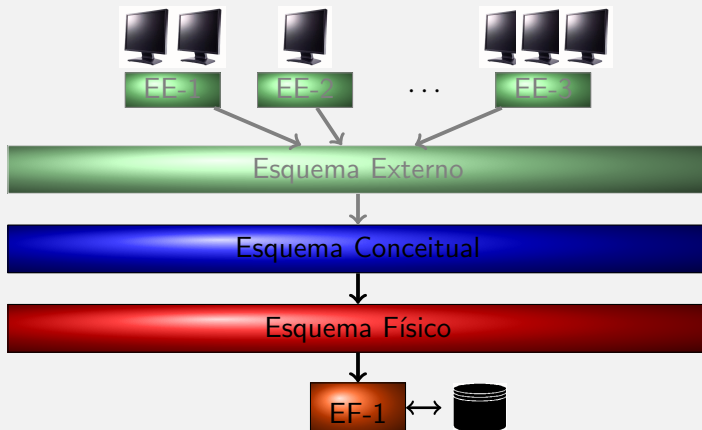
- A noção de um modelo de representação de dados em três níveis conceituais é conhecido como o **Modelo ANSI/X3/SPARC** (Standards Planning and Requirements Committee):
- Os **Esquemas Externos** representam a porção dos dados necessários a cada aplicação, no formato que a aplicação precisa dos dados;
 **Visão do Usuário**
- O **Esquema Conceitual** representa a integração de todos os conceitos que a organização precisa representar usando seus dados;
 **Visão do Sistema**
- O **Esquema Físico** descreve o formato interno dos dados, tal como armazenado no meio físico;  **Visão do Computador**

Conceitos e Motivação

Modelagem Relacional representada em três níveis

- O Modelo Relacional pode ser usado para representar dados em três níveis conceituais
- O nível correspondente ao **Esquema Conceitual** corresponde ao conjunto de todas as relações e atributos da base de dados;
- Portanto, os comandos de definição de relações (CREATE TABLE) correspondem à definição do esquema conceitual.
- O **Esquema Físico** deve indicar como os dados são organizados fisicamente nos meios de armazenagem. Em SQL, isso é indicado por **Índices** e **Espaços para Tabelas**.
- Os **Esquemas Externos** são indicados por um conceito do Modelo Relacional denominado **Visões**

O Esquema Físico em SQL



O Esquema Físico em SQL

O **Esquema Físico** é indicado em SQL criando:

- 1 INDEX
- 2 TABLESPACE

Índices

- Índices são estruturas que organizam os dados (em geral na memória permanente: discos) para agilizar a recuperação dos dados.
- Índices são criados:
 - Automaticamente: quando se declara conjuntos de atributos como chaves PRIMARY KEY ou UNIQUE;
 - Explicitamente: com os comando CREATE/ALTER/DROP INDEX.
- Os índices criados para as restrições PRIMARY KEY ou UNIQUE não podem ser diretamente removidos: é necessário que se remova a restrição.
- No entanto, eles também são índices, como todos os demais.

DDL – Comando CREATE INDEX

Cria um novo índice numa tabela

Sintaxe Básica:

CREATE INDEX

```
CREATE [UNIQUE] INDEX [<nome>] ON <table> [USING <metodo>]
    ({<atributos> | (<expressao>)}
    [ASC | DESC]
    [NULLS {FIRST | LAST}]
    [, ...]
    )
    [WITH (storage_parameter = valor [, ...] ) ]
    [TABLESPACE nome_tbs]
    [WHERE <predicado>];
```

Em Postgresql <metodo> pode ser:

- BTREE – múltiplos atributos, UNIQUE, ordem
- HASH –
- GIST – múltiplos atributos, inclui rtree.
- GIN – (Generalized Inverted Files)

DDL – Comando CREATE INDEX

- Os atributos a serem indexados são indicados como <atributos> ou como componentes de uma (<expressao>);
- A cláusula WHERE indexa apenas as tuplas que atendam ao predicado. Pode-se incluir qualquer atributo da tabela, mesmo os que não estejam indexados, para filtrar as tuplas.
- A cláusula TABLESPACE indica qual espaço de armazenamento usar, caso não indicado, é usado o default_tablespace.
- A cláusula WITH permite informar parâmetros para os índices
 - FILLFACTOR (Btree, Hash, GIST): Porcentagem de ocupação dos nós no índice. **DICA:** tabelas estáticas devem ter configuradas fillfactor 100 para otimizar o tamanho do índice em disco.

DDL – Comando CREATE INDEX

Comando CREATE INDEX – Exemplos

Exemplos:

```
CREATE INDEX IdxRA ON Matricula (RA);
```

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

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

útil quando usado em consultas como

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

```
CREATE INDEX IdxNivel ON Professor USING hash (nivel);
```

DDL – Comando CREATE INDEX

- Dependendo do índice criado, o gerenciador não utiliza ele na consulta.
- Por exemplo na tabela aluno:
`CREATE TABLE aluno (ra integer primary key, nome varchar);`
- Criamos um índice no campo nome, caso existirem muitas consultas nele:
`create index on aluno (nome);`

DDL – Comando CREATE INDEX

- insira algumas tuplas (5) de exemplo, entre elas um aluno com o nome Joao.
- Para analisar como o comando SQL é executado, use o comando explain:

```
explain select * from aluno where nome = 'Joao';
```

- Veja que o sequencial scan é feito.

DDL – Comando CREATE INDEX

- Vamos popular a tabela. Execute o código PLPGSQL:

```
do $$  
begin  
for i IN 10..100000 LOOP  
insert into aluno values (i, 'nnn');  
end loop;  
end $$;
```

- Faça novamente o explain.
explain select * from aluno where nome = 'Joao';

DDL – Comando CREATE INDEX

- Agora compare os seguintes resultados explain:

```
explain select * from aluno where nome = 'Joao';
```

```
explain select * from aluno where lower(nome) = 'joao';
```

DDL – Comando CREATE INDEX

- Agora vamos criar um índice apropriado para este campo.
- O seguinte comando criará um índice bitmap no campo nome.
`create index on aluno (lower(nome));`
- Refaça:
`explain select * from aluno where lower(nome) = 'joao';`
- Compare com:
`explain select * from aluno where ra=1;`

DDL – Comando CREATE INDEX

Comando CREATE INDEX – Exemplos com a cláusula WHERE

Exemplos:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
          client_ip < inet '192.168.100.255');
```

útil quando usado em consultas como

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

mas não será útil em consultas como

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

DDL – Comando CREATE INDEX

Comando CREATE INDEX – Exemplos com a cláusula WHERE

Exemplos:

```
CREATE INDEX IdxRACapital ON Aluno (RA)  
WHERE CIDADE='Curitiba';
```

```
CREATE INDEX IdxRAInterior ON Aluno (RA)  
WHERE CIDADE='Pato Branco';
```

útil quando usado em consultas como

```
SELECT * FROM Aluno  
WHERE CIDADE='Curitiba' AND Idade>20;
```

```
SELECT * FROM Aluno  
WHERE CIDADE='Pato Branco' AND Idade>20;
```

```
SELECT * FROM Aluno  
WHERE CIDADE='Pato Branco' AND RA=1234;
```

mas não será usado em consultas como

```
SELECT * FROM Aluno  
WHERE Idade>20;
```

Índices de chaves compostas

- Índices criados em chaves compostas devem ser utilizados com cuidado nas consultas SQL.

- Por exemplo:

```
create table tab (campo1 varchar, campo2 integer,  
campo3 numeric, constraint chave primary key  
(campo1,campo2));
```

```
do $$
```

```
begin
```

```
for i in 1..6000 loop
```

```
  insert into tab values (texto(3), i, i*3.3);
```

```
  end loop;
```

```
end
```

```
$$ language plpgsql
```

- Teste as consultas: (Quais usam o índice?)

```
explain select * from tab where campo1 = 'aaa' and  
campo2 = 1;
```

```
explain select * from tab where campo2 = 1;
```

```
explain select * from tab where campo1 = 1;
```

Índices de chaves estrangeiras

- É muito comum que se esqueça de criar índices para chaves estrangeiras.
- Isso reduz bastante o desempenho das junções.
- Exemplo:

Considerando as Tabelas

```
create table cliente (idCliente serial primary key,  
                      nome varchar, datanasc date);  
  
create table carro(id serial primary key,  
                  ano integer, modelo varchar,  
                  idCliente integer references  
                      cliente(idCliente));
```

Índices de chaves estrangeiras

Populando as tabelas e calculando as métricas

```
do $$
begin
for i in 1..100000 loop
    insert into cliente (nome,datanasc)
        values (texto(6), data());
end loop;
end
$$ language plpgsql

do $$
begin
for i in 1..100000 loop
    insert into carro (ano,modelo,idCliente) values
        ((random()*100)::integer, texto(6), i);
end loop;
end
$$ language plpgsql

analyze cliente;
analyze carro;
```

Índices de chaves estrangeiras

- Retorne o nome e o modelo do carro do cliente ID = 30

Repita a consulta com e sem o índice

```
EXPLAIN ANALYZE
SELECT NOME, MODELO
FROM CLIENTE NATURAL JOIN CARRO
WHERE IDCLIENTE=30

create index idxCliCarro on carro(idCliente);
```

Índices para busca de trechos de texto

- Buscas por substrings usando o LIKE e ILIKE com Prefixo
- Indexar um atributo de texto pode auxiliar a pesquisa caso exista um prefixo na consulta.

Exemplo

```
create table teste(nome varchar);

do $$
begin
  for i in 1..100000 loop
    insert into teste values (texto(10));
  end loop;
end; $$
language plpgsql;

analyze teste;
```

Índices para busca de trechos de texto

Crie o índice e teste as consultas:

```
create index idxtext on teste(nome);
```

```
explain analyze
```

```
select * from teste where nome LIKE 'eQi%';
```

```
explain analyze
```

```
select * from teste where nome LIKE '%eQi%';
```


Índices para busca de trechos de texto

Criar uma vez para cada DATABASE

```
CREATE EXTENSION pg_trgm;
```

Criando o índice

```
create index idxtextTrgm on teste  
using GIN(nome gin_trgm_ops);
```

Bitmap Indexes

- Quando um campo a ser indexado possui poucos valores (até 10), é interessante indexá-lo com um Bitmap Index.

Habilitando BITMAP INDEX

```
create extension btree_gin;
```

Bitmap Indexes

```
create table pessoa (  
    id serial primary key,  
    nome varchar,  
    genero varchar(1) check (genero in ('M','F')));  
  
do $$  
begin  
    for i in 1..100000 loop  
        insert into pessoa(nome, genero) values (texto(10),'M');  
    end loop;  
end; $$  
language plpgsql;
```

Repita a injeção para gêneros femininos.

Bitmap Indexes

- Teste a consulta:

```
explain analyze
```

```
select * from pessoa where genero = 'M';
```

- Sem Index.
- Com index B-tree

```
create index idxgeneroBTree on pessoa (genero);
```

- Com index bitmap

```
create index idxgeneroBitmap on pessoa using gin (genero);
```

Indexando JSON

- Considere a tabela
students (info jsonb);
 - com os dados
do \$\$
declare
vartype varchar[] = '{"quiz", "exam", "homework"}';
begin
for i in 1..1000000 loop
for j in 1..3 loop
insert into students values (('
- ```

{
 "student" : ' || i || ',
 "type" : " " || vartype[j] || ",
 "score" : ' || round(random()*100) || '
}'))::json);
end loop;
end loop;
end; $$
language plpgsql;
```

# Índices em JSON

- Faça a consulta: (sem índice).

```
explain analyze
Select info
FROM students
where info->>'type' = 'quiz'
```

- Crie um índice Btree desta maneira:

```
create index idxtype on students
using BTREE ((info->>'type'));
```

# Índices em JSON

- Faça a consulta: (sem índice).

```
explain analyze
```

```
Select *
```

```
FROM students
```

```
WHERE info @> '{"type": "quiz"}';
```

- Crie um índice GIN desta maneira:

```
create index idxJSON on students using GIN (info);
```

# Índices em JSON

- Veja agora as consultas (agilize se necessário):

```
Select info->>'student'
FROM students
where info->>'score' = '10'
group by info->>'student'
```

```
explain analyze
Select *
FROM students
WHERE info @> '{"student": 3212}' and info->>'type' = 'exam';
```

```
explain analyze
Select info->>'score'
FROM students
WHERE info->>'student' = '3212'
```



## DDL – Comando ALTER INDEX

### ALTER INDEX

```
ALTER INDEX <nome> RENAME TO <novo_nome>
```

Por exemplo:

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

# DDL – Comando DROP INDEX

## DROP INDEX

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

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

# Estruturas de Dados para Índices

- 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 mais de uma tupla;
  - Em 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 Estruturas de Dados, onde repetições são permitidas.

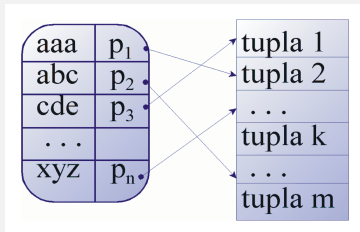
# Estruturas de Dados para Índices

- 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.

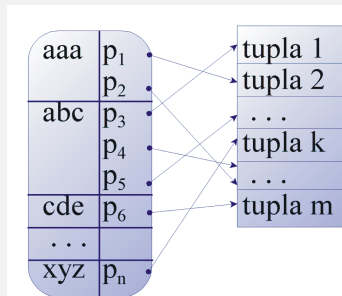
# Estruturas de Dados para Índices

- 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 de onde a tupla indexada por aquele **Valor** está armazenada.
- A estrutura interna organiza a coleção de pares  $\langle \text{Valor}, \text{RowId} \rangle$ :

Chave de busca única:

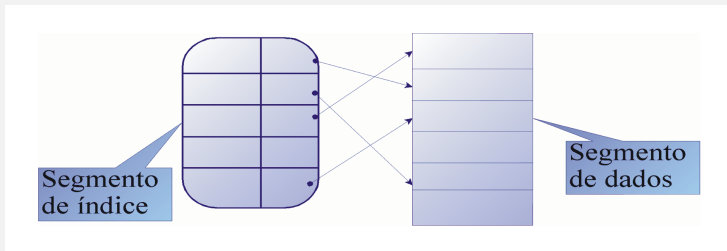


Chave de busca múltipla:



# Indexação - Introdução

Os registros físicos em disco de dados e de índices têm estruturas de acesso e de armazenagem diferentes.



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

# Estruturas de Dados para Índices

- É possível que um dos índices defina a ordem física das tuplas na relação: **Índice de agrupamento**;
- Os ponteiros dos índices podem apontar para a tupla ou um registro:
  - ① **Índice denso**: um registro de índice ocorre para cada valor de chave de busca.
  - ② **Índice de agrupamento denso**: o registro de índice tem o ponteiro para o primeiro registro de dados com o valor da chave de busca.
  - ③ **Índice esperso**: Uma entrada de índice para apenas alguns valores da chave de busca. As demais chaves são buscadas sequencialmente a partir da chave de busca de valor próximo.
- **Índice de Cobertura**: índices que armazenam também os valores de alguns atributos (fora a chave de busca), junto com os ponteiros de registros. Algumas consultas podem ser respondidas somente utilizando esse tipo de índice.

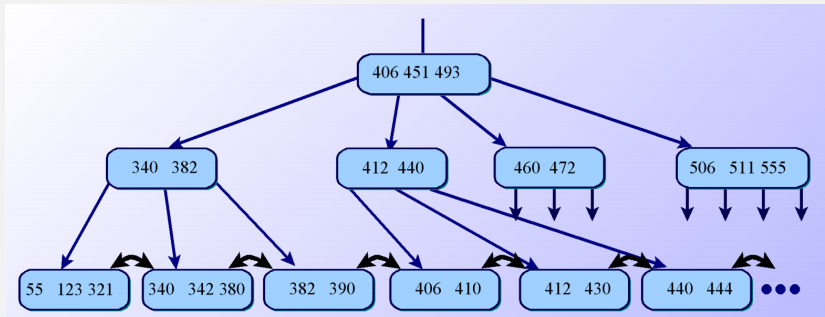


# Estruturas de Dados para Índices

- 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.

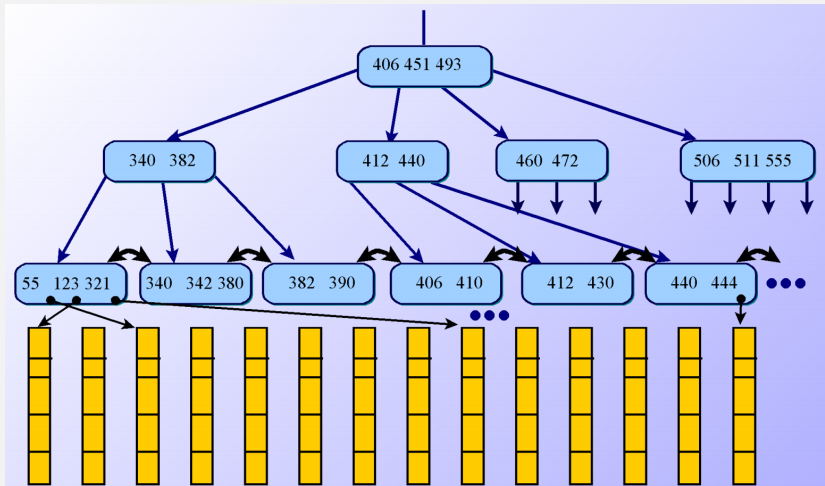
# Árvore B<sup>+</sup>-tree

- Utiliza uma estrutura de dados B-tree, com as folhas ligadas (ponteiros para irmãos).



# Árvore B<sup>+</sup>-tree

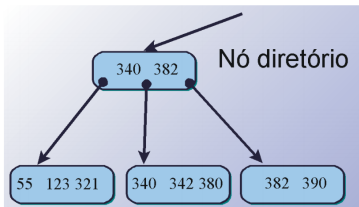
- As folhas apontam para as tuplas nas páginas de dados que armazenam as relações, formando uma “**chuva de ponteiros**”.



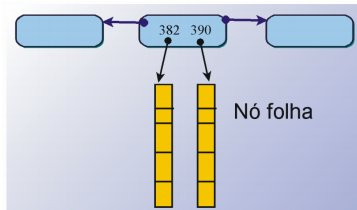
# Árvore B<sup>+</sup>-tree

- Uma B-tree usada na estrutura de memória física de um SGBD tem várias propriedades específicas:
  - Cada nó é armazenado em uma página de um segmento de índice;
  - Um nó pode ser “nó folha” ou “nó diretório”

Um nó diretório contém  $n$  chaves de acesso e  $n + 1$  ponteiros para outros nós da B-tree.



Um nó folha contém  $n$  chaves de acesso,  $n$  ponteiros para tuplas e 2 ponteiros irmãos para os nós anterior e sucessor da sequência de chaves.









# Árvore B<sup>+</sup>-tree

- Uma B-tree usada em um SGBD tem várias propriedades específicas:
  - Um nó é limitado pelo tamanho da página em disco: o número de chaves de acesso que ele pode conter vai depender do tamanho em bytes de cada chave;
  - Novas chaves vão sendo inseridas até atingir o fillfactor do nó, após isso o nó divide.
  - Cada nó fica sempre pelo menos 50% cheio;
  - O nó raiz sempre tem pelo menos duas chaves, caso contrário a árvore reduz um nível;
  - Como o crescimento da árvore se dá pela quebra de nós folha e a propagação da quebra em direção à raiz, a árvore sempre fica balanceada.
  - A busca entre as chaves armazenadas em um nó pode ser feita usando o método da busca binária.

# ISAM

- ISAM significa acesso Indexado e Sequencial.
  - O acesso Indexado requer navegar a árvore desde a raiz até a folha, percorrendo todos os níveis da árvore;
  - Em geral, é lida uma única página de cada nível da árvore, portanto não tem sentido ler mais do que um nó diretório em cada acesso ao disco.
  - O acesso Sequencial parte de um nó folha a que se chegou por acesso indexado, e continua a navegação seguindo a lista sequencial pelos ponteiros entre nós folha;
- Procedimento para carga massiva de dados:
  - 1 Desativar todos índices;
  - 2 Carregar os dados;
  - 3 Reativar os índices.

# Operadores Relacionais que usam Índices

- Os índices são usados para operadores:
- Unários:
  - Seleção:  $\sigma_{(C)} R$   qualquer índice (depende do tipo de consulta e do tipo de dados)
  - Agrupamento:  $\gamma_{\{A\}} R$   B-tree, Bitmap, Arquivos invertidos
  - Ordenação:  $\omega_{\{A\}} R$   B-tree
  - 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

# Táticas para criação de índices

- 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.



# Melhorando a performance dos índices

- Uma das maneiras de melhorar a performance de consultas usando índices no PostgreSQL é utilizando índices clusterizados. Para isso, deve ser gerado um índice clusterizado na tabela alvo.
- Por exemplo:

```
create table aluno (ra integer, nome varchar, idade
integer, cidade varchar, constraint pk_aluno primary key
(ra));
```

```
ALTER TABLE aluno CLUSTER ON pk_aluno;
analyze aluno;
cluster aluno;
```

OBS: No SQL-Server a ordem é mantida para novas entradas, mas no PostgreSQL deve-se realizar o comando cluster toda vez em que se deseja reordenar as tuplas em disco.

# Melhorando a performance dos índices

- O FILLFACTOR de um índice deve ser configurado corretamente, de acordo com a taxa de atualização na tabela.
- Ele pode ser atribuído no momento de criação do índice:  
`CREATE INDEX nome_idx on tabela(coluna) WITH (FILLFACTOR=80);`
- Isso pode ajudar a realizar menos acessos a disco, se as tabelas:
  - São pouco atualizadas: FILLFACTOR=99
  - São atualizadas apenas 10% da tabela por semana: FILLFACTOR=90
- Automatização pode ser feita usando cron jobs (em sistemas unix/linux): `crontab cron.txt`, ou usando o PGAgent (pelo pgAdmin 3 em diante).

# Espaços para Tabelas

- Todos os dados de uma base de dados (tabelas, índices, áreas de trabalho, etc.) são armazenados em memória persistente (discos, memória flash, discos óticos, etc.);
- Os espaços de armazenagem podem ser
  - 1 Arquivos do sistema operacional;
  - 2 Meios físicos não estruturados (*raw disks*);
  - 3 Partições de um disco.
- Cada um desses espaços é chamado um **Espaço para tabela - Tablespace**.
- Cada objeto de dados (uma tabela, um índice, etc.) tem que ser armazenado inteiro em um único *tablespace*, mas um *tablespace* pode armazenar vários objetos.

# DDL – Comando CREATE TABLESPACE

Postgres

- *Tablespaces* devem ser declarados explicitamente usando o comando `CREATE TABLESPACE`;
- A sintaxe desse comando varia muito entre os produtos.
- Exemplos de sintaxe em Postgres e Oracle são:

## CREATE TABLESPACE – Postgres

```
CREATE TABLESPACE tblspc_name [LOCATION file_name]
 <outras...>
```

# Comando DDL – Comando CREATE TABLESPACE

Oracle

## CREATE TABLESPACE – Oracle

```
CREATE TABLESPACE tblspc_name
 [DATAFILE file_name [SIZE ext_size] [REUSE]]
 AUTOEXTEND OFF |
 AUTOEXTEND ON [NEXT add_size [MAXSIZE {UNLIMITED |
 max_size}],
 [DEFAULT STORAGE ([INITIAL init_size] [NEXT next_size]
 [MINEXTENTS n] MAXEXTENTS {n | UNLIMITED})
 [PCTINCREASE n])
 [MINIMUM EXTENT min_size]
 <outras...>
```

onde qualquer size é um <integer>[K, M, G, T, P, E]

# Comando CREATE TABLESPACE

A cláusula `LOCATION` em Postgres ou `DATAFILE` em Oracle servem para indicar onde os dados serão armazenados.

- Postgres somente usa arquivos do sistema operacional.
- *Raw disk partitions* (Oracle) Manipulam diretamente o dispositivo sem utilizar o sistema operacional.
  - Caíram em desuso, pois os novos sistemas de arquivos são muito eficientes e tolerantes a falhas.

# Comando CREATE TABLESPACE – Exemplos

Exemplos:

```
CREATE TABLESPACE TSGraduacao LOCATION '/Grad/dbs';
```

(Postgres)

```
CREATE TABLESPACE TSGradIndice
 DATAFILE '$ORACLE_HOME/rdbms/Grad/dbidx.dat'
 SIZE 20M AUTOEXTEND ON;
```

(Oracle)

```
CREATE TABLESPACE TSGrads2
 DATAFILE '/DEV/DK2';
```

(Oracle)

## Comando DDL – Comandos ALTER/DROP TABLESPACE

### ALTER TABLESPACE

```
ALTER TABLESPACE tblspc_name RENAME TO new_name
```

Exemplo:

```
ALTER TABLESPACE TSGraduacao RENAME TO Jupiter;
```

### DROP TABLESPACE

```
DROP TABLESPACE [IF EXISTS] tblspc_name
```

Exemplo:

```
DROP TABLESPACE TSTempoDeMatriculas;
```



# Comando CREATE TABLE

- Os comandos de declaração de tabelas e de índices têm construções especiais para indicar as *tablespaces* onde devem ser armazenados.
- Existem alterações no comando CREATE TABLE para indicar onde as tabelas são armazenadas:

## CREATE TABLE

```
CREATE TABLE tbl-name (definição de atributos e restrições)
[TABLESPACE tblspc_name];
```

- e onde os índices das restrições PRIMARY KEY e UNIQUE são armazenados:

## Restrições UNIQUE e PRIMARY KEY do comando CREATE TABLE

```
[USING INDEX TABLESPACE tblspc_name]
```

# Comando CREATE TABLE – Exemplo

Exemplo:

```
CREATE TABLE Turma (
 sigla char(7) NOT NULL,
 numero decimal(2) NOT NULL,
 codigo decimal(4)
 PRIMARY KEY USING INDEX TABLESPACE TSGradIndice,
 NNalunos decimal(3),
 FOREIGN KEY SiglaDaTurma (Sigla)
 REFERENCES Discip (Sigla)
 ON DELETE CASCADE
 ON UPDATE CASCADE,
 UNIQUE SiglaNumero (Sigla, Numero)
 USING INDEX TABLESPACE TSGradIndice,
 CHECK LimiteDeVagas (NNalunos<50)
) TABLESPACE TSGraduacao;
```

# Default Tablespace

- Caso não seja especificado, tudo é armazenado no tablespace padrão, que é o default\_tablespace.
- Pode-se alterar o local padrão de armazenamento através do comando:

```
SET default_tablespace = space1;
CREATE TABLE Tab(i int);
```

- O tablespace temporário pode-se ser alterado desta maneira:

```
SET temp_tablespace = space2;
```

- Os nomes dos tablespaces podem ser acessados no catálogo:

```
SELECT spcname FROM pg_tablespace;
```

# Tablespaces para Database

- Inclusive, podemos alocar uma database inteira para um tablespace

```
CREATE DATABASE base1 TABLESPACE = space1;
```

# Comando CREATE INDEX

O comando CREATE INDEX também tem uma extensão para indicar onde as tabelas são armazenadas:

## CREATE INDEX

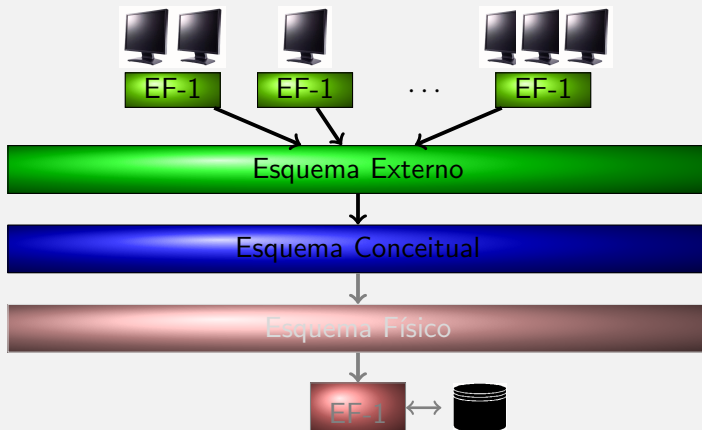
```
CREATE [UNIQUE] INDEX idx-name
 ON table [USING method]
 ({columns_def})
 [TABLESPACE tablespace]
 [WHERE predicate]
 <outras...>
```

# Comando CREATE INDEX – Exemplo

Exemplo:

```
CREATE INDEX IdxRA ON Matricula (RA)
TABLESPACE TSGradIndice;
```

# Os Esquemas Externos no Modelo Relacional



# Os Esquemas Externos no Modelo Relacional

- Os **Esquemas Externos** são representados em SQL pelo conceito de **Visões**, auxiliado pelo conceito de **Triggers**.
- As relações criadas com o comando CREATE TABLE são as que existem de fato no sistema, sendo chamadas de “**Relações=base**”.
- As relações-base são armazenadas em memória persistente.
- No entanto, o resultado de expressões de consulta são também tabelas, mas elas não são (necessariamente) armazenadas, existindo como resultado do processamento da expressão sobre as relações-base.
- Se for dado um nome para o resultado de uma expressão, essa relação passa a ser chamada de “**Visão**”.
- Visões podem ser consultadas exatamente como uma relação-base, e com o auxílio de **Triggers**, podem também ser atualizadas (o que implica em atualizar as relações-base usadas para calcular a visão).



# DDL – Comando CREATE VIEW

Cria uma nova visão

Sintaxe Básica:

## CREATE VIEW

```
CREATE [OR REPLACE] VIEW View_name [(Attrib_name [, ...])]
AS query
```

- A lista opcional de atributos (Attrib\_name, ...) indica os nomes dos atributos que compõem a relação resultado. Se não indicada, ela é deduzida da consulta query;
- A consulta query é um comando SELECT ou uma lista de valores VALUES;
- Em princípio, a tabela resultante da consulta query não é armazenada, sendo recalculada sempre que essa visão for mencionada em algum comando;

# DDL – Comando CREATE VIEW – Exemplos

Exemplos:

```
CREATE VIEW Doutores AS
```

```
 SELECT *
```

```
 FROM Professores
```

```
 WHERE Nivel = 'MS-3';
```

Professores=(Nome, Idade, Nivel, NNFunc)

~~Doutores=(Nome, Idade, Nivel, NNFunc)~~

```
CREATE VIEW Niveis (Nivel, Título) AS
```

Niveis=(Nivel, Título)

```
 VALUES ('MS-1', 'Auxiliar'), ('MS-2', 'Mestre'), ('MS-3', 'Doutor'),
 ('MS-5', 'Livre docente'), ('MS-6', 'Titular');
```

```
CREATE OR REPLACE VIEW Doutores (Nome, Idade, NumFuncional) AS
```

```
 SELECT Nome, Idade, NNunc
```

```
 FROM Professores P, Niveis N
```

```
 WHERE P.Nivel = N.Nivel AND
 N.Título='Doutor';
```

Doutores=(Nome, Idade, NumFuncional)

# Gatilhos (*Triggers*)

## Gatilhos – *Triggers*

São regras do tipo  $\langle \text{Evento}, \text{Condição}, \text{Ação} \rangle$  (regras ECA).

Ou seja, elas atuam quando ocorre um **Evento** predeterminado e uma dada **Condição** é satisfeita, executando uma **Ação**.

- *Triggers* são associados a relações (tanto relações-base quanto visões).
- **Evento** é uma solicitação de operação de atualização da relação, ou seja: INSERT, UPDATE ou DELETE. A finalização de uma transação é também um outro evento.
- Quando o evento ocorre, a **Condição** é testada.
- Quando a condição é satisfeita, a **Ação** do *Trigger* é executada.

# Gatilhos (*Triggers*)

*Triggers* são executados no servidor, sempre são associados a uma tabela ou a uma view, e a um comando *INSERT*, *UPDATE* ou *DELETE*.

- Procedimentos *Trigger* não podem ser chamados explicitamente.
- Sempre que o comando associado é executado na tabela associada, o procedimento *trigger* é executado.
- A execução pode ser repetida para cada tupla afetada ou apenas uma vez para cada comando.

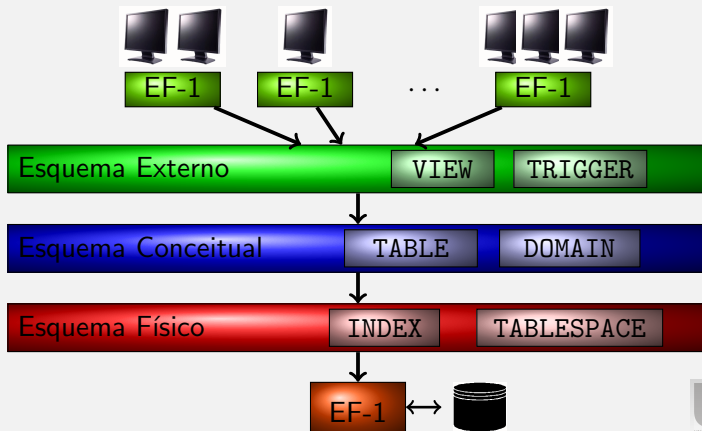
# Triggers

Existem diversas opções para definir um *Trigger*:

- A Ação pode ser executada antes ou depois (BEFORE ou AFTER) do evento;
- O corpo da Ação pode se referir ao valor das tuplas antes e depois da operação que dispara o *Trigger* (NEW e OLD);
- O corpo da Ação pode ser executado ao invés da operação solicitada (INSTEAD OF).
- Eventos disparados por UPDATE podem levar em conta um subconjunto específico de atributos da relação;
- Para comandos que atualizam mais de uma tupla, é possível especificar se a ação é executada uma vez para cada tupla, ou uma vez só para todas as tuplas modificadas pelo comando.

# Conclusão

- Aplicado ao Relacional, o modelo de representação de dados em três níveis garante a **Integração** e a **Independência de Dados** da aplicação.



# Roteiro

- 1 Conceitos e Motivação
- 2 O Esquema Físico em SQL
- 3 Esquemas Externos no Modelo Relacional
- 4 Visões
- 5 Triggers
- 6 Conclusão

# SGBDs Relacionais

## – Representação de Dados em Três Níveis Conceituais

**Prof. Dr. Ives Renê V. Pola**

[ivesr@utfpr.edu.br](mailto:ivesr@utfpr.edu.br)

Departamento Acadêmico de Informática – DAINF

UTFPR – Pato Branco DAINF

UTFPR

Pato Branco - PR

FIM

