# Arquitetura de SGBD Relacionais
## — Indexação —

Caetano Traina Jr.

Grupo de Bases de Dados e Imagens
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
caetano@icmc.usp.br

13 de junho de 2013
São Carlos, SP - Brasil

Esta apresentação mostra os principais conceitos do uso da Indexação em SGBD Relacionais. Discute o uso de comandos de indexação em SQL, e como estruturas de indexação são usadas pelo SGBD, bem como as principais combinações são utilizadas.

# Roteiro

# Indexação - Introdução

- A indexação de dados é utilizada como uma maneira de acelerar o processamento de consultas em um SGBD Relacional.
- É usada fundamentalmente para organizar os valores de um atributo.
- Ela é usada como meio de acesso aos dados, e na maioria das vezes é aplicada sobre atributos que tenham uma seletividade muito alta, tal como chaves das relações.

...Chaves! - e la vamos nós outra vez...

# Indexação - Introdução

## Indexação - Terminologia

### Terminologia

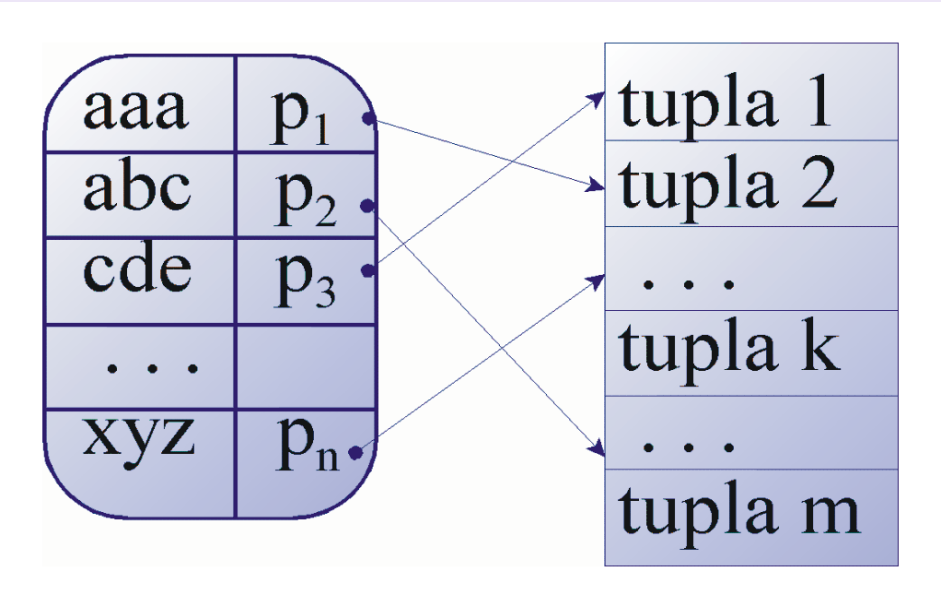
- A indexação em SGBD envolve duas áreas da computação:
  - Bases de Dados
  - Algoritmos e Estruturas de Dados
- Cada área usa um significado para o termo "chave":
  - Em Bases de Dados, chave é um valor que não pode repetir em mais de uma tupla;
  - Em Estruturas de Dados, chave é o valor usado para buscar um elemento armazenado na estrutura, não existe restrição de que ele seja único.
- Nesta apresentação usaremos o termo **chave da relação** ou simplesmente **chave** para indicar chave das relações segundo o uso do termo chave em Bases de Dados,
- e o termo **chave de busca** para indicar o valor de busca em Estruturas de Dados.

# Indexação - Introdução

- Índices são criados sobre um ou mais atributos.
  Quando um índice é criado sobre dois ou mais atributos, utiliza-se como chave de busca a concatenação dos valores dos atributos envolvidos.
- ⋆ SGBDs em geral aceitam até 32 atributos concatenados em um índice.
- Pode-se imaginar um índice como uma coleção de pares `<Valor, RowId>`, onde `Valor` é a chave de busca do índice, e `RowId` e o endereço físico de onde a tupla indexada por aquele `Valor` está armazenada.

# Indexação - Conceitos

Um índice é uma coleção de pares <Valor, RowId>:

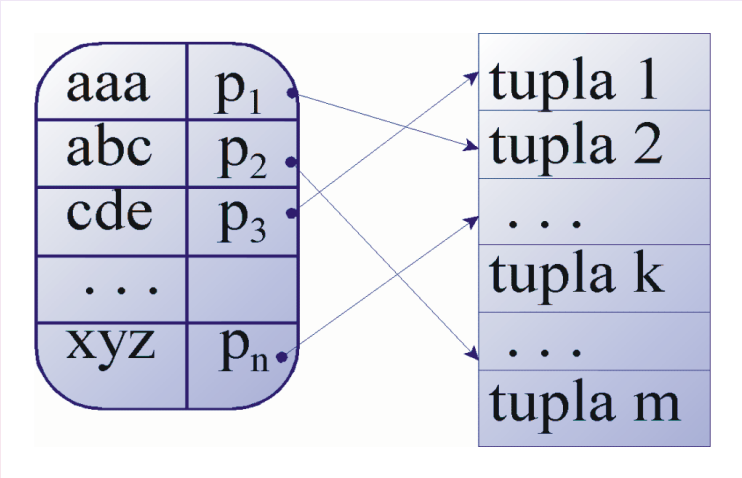



Essa é só uma forma conceitual de imaginar índices: deve-se lembrar que a organização das chaves é feita por uma estrutura de dados que visa agilizar a busca.

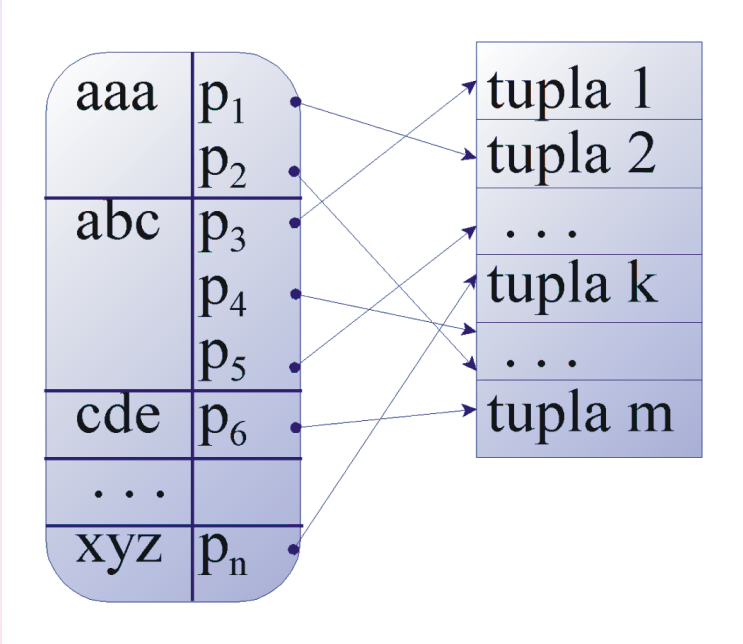
# Indexação - Conceitos

De maneira distinta do que se imagina em uma relação, uma chave de busca não precisa ser única:

Chave de busca única:




Chave de busca múltipla:




São comuns índices ISAM (*Indexed-Sequential Access Method*), índices invertidos, índices BitMap e índices *hash*.

# Indexação - Conceitos

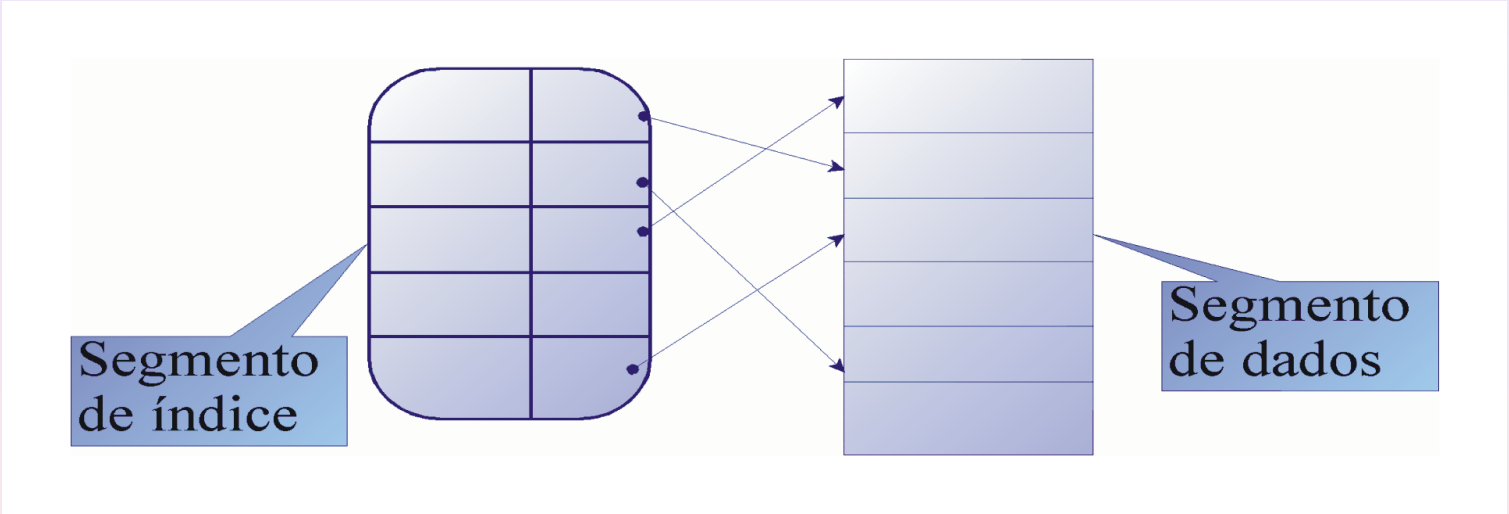
- Uma relação é um dado **Dado primário**, que não pode ser alterado a não ser por solicitação expressa do usuário.
- Um índice é chamado **Dado secundário**, e pode ser apagado e recriado a partir da relação.
- Assim, índices pode ser criados e apagados a qualquer instante.
- Se uma tupla é inserida, 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 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.

# Indexação - Introdução

- 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 chamado **Segmento**.
- Existem diversos tipos de segmentos em um SGBD, sendo que cada segmento de dados armazena uma relação, e cada 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*).

# Indexação - Introdução

Além disso, segmentos de dados e de índices têm estruturas de acesso e de armazenagem diferentes.




Usualmente, o segmento de dados de uma relação e os segmentos de seus índices são colocados na mesma unidade de armazenagem, chamada **tablespace**, mas o usuário pode especificar livremente onde colocar cada um.

# Declaração de Índices em SQL

- Existem duas maneiras de se declarar índices em SQL:
  - **Implicitamente:** isso ocorre quando se declara uma chave primária ou candidata;
  - **Explicitamente:** quando se usa o comando `CREATE INDEX`.

# Declaração Implícita de Índices como chaves

- A restrição de integridade `PRIMARY KEY` num comando `CREATE TABLE` ou `ALTER TABLE` corresponde à definição de uma chave primaria em SQL.
- A restrição de integridade `UNIQUE` corresponde à definição de uma chave candidata em SQL.
- Qualquer restrição pode ser indicada como
  - **Uma restrição de atributo**
  - **Uma restrição de relação**
- **Nota:**
  É sempre uma boa idéia dar nome para as restrições que criam índices, pois assim podem ser criadas e extintas quando necessário.

# Declaração Implícita de Índices como chaves

## Restrição de atributo

- Uma restrição de atributo (também chamada restrição de coluna) ocorre quando se declara a restrição junto com a declaração do atributo.
- Por exemplo:

```
NUSP DECIMAL(10) NOT NULL PRIMARY KEY,
                                -- restrição sem nome
NUSP DECIMAL(10) NOT NULL
                 CONSTRAINT ChaveDaRelacao PRIMARY KEY,
                                -- restrição com nome
```

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

# Declaração Implícita de Índices como chaves

## 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),
                              -- restrição sem nome
CONSTRAINT NaoRepeteNome
    UNIQUE(Nome, NomeDaMae, DataNascimento),
                                -- restrição com nome
```

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

# Declaração Explícita de Índices

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

## CREATE INDEX

```
CREATE [UNIQUE | CLUSTERED] INDEX idx-name
     ON table [USING method]
    ({column | (expression)} [ASC | DESC][NULLS {FIRST | LAST}
    [WITH (storage_parameter = value [, ... ])]
    [TABLESPACE tablespace]
    [WHERE predicate]
```

- Cada fabricante de SGBDs varia bastante esse comando. A sintaxe mostrada é um resumo do que a maioria disponibiliza.
- Por exemplo:
  - Em POSTGRES, `CLUSTER` é um comando da DML, e executa a 'clusterização' quando o comando é solicitado;

# Declaração Explícita de Índices

É importante destacar alguns pontos sobre o comando `CREATE INDEX`:

- `USING method` indica a estrutura de dados a ser usada para o índice.

| SGBD | ISAM | invertido | BitMap | *hash.* |
|---|---|---|---|---|
| POSTGRES | B-tree | GIN | | Hash |
| ORACLE | B-tree<br>B-tree cluster | | Bitmap<br>Bitmap join | Hash cluster |

(ORACLE não usa a construção `USING`, mas indica o tipo de índice pela estrutura sintática da especificação das colunas)

- Índices `UNIQUE` indicam que a chave de busca não pode ser repetida. Somente índices *B-tree* suportam `UNIQUE`.

# Declaração Explícita de Índices

Índices não precisam ser criados sobre atributos: expressões podem ser indexadas.

- Indexar expressões é um recurso para agilizar buscas que se fazem com frequência sobre as expressões.
  Por exemplo:
  - Suponha que foi indicada uma chave da relação `Alunos` como:
    ```
    Nome VARCHAR(60) NOT NULL UNIQUE,
    ```
  - então a consulta:
    ```
    SELECT * FROM Alunos WHERE lower(Nome)='maria';
    ```
    não usará o índice.
  - No entanto, pode ser criado o índice
    ```
    CREATE INDEX NomeMinusculas ON Alunos
    (lower(Nome));
    ```
    e nesse caso aquela busca usará esse índice.

# Declaração de Índices em SQL

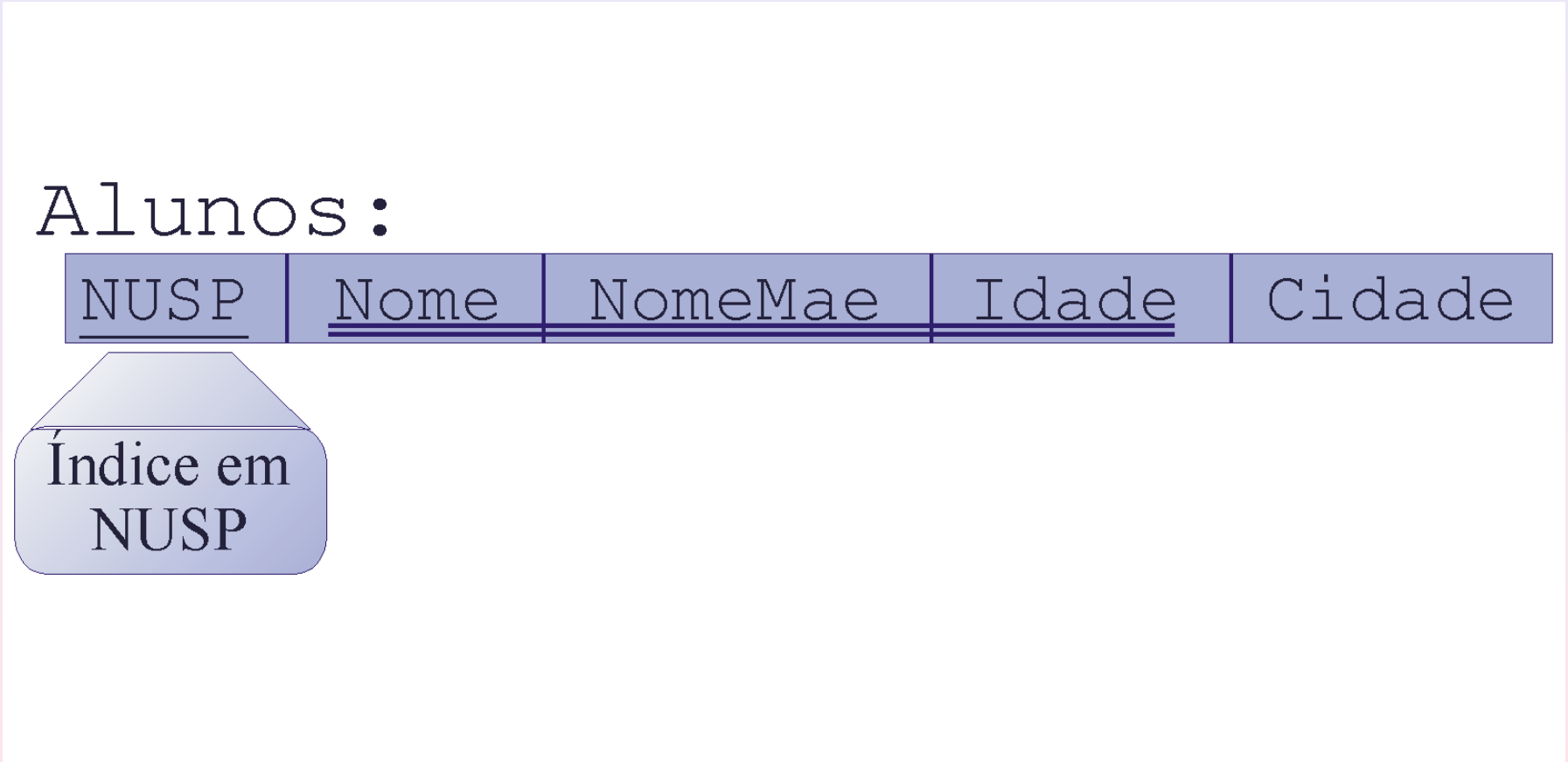
Podem ser criados muitos índices sobre uma mesma relação:

Alunos:

| NUSP | Nome | NomeMae | Idade | Cidade |
|---|---|---|---|---|

# Declaração de Índices em SQL

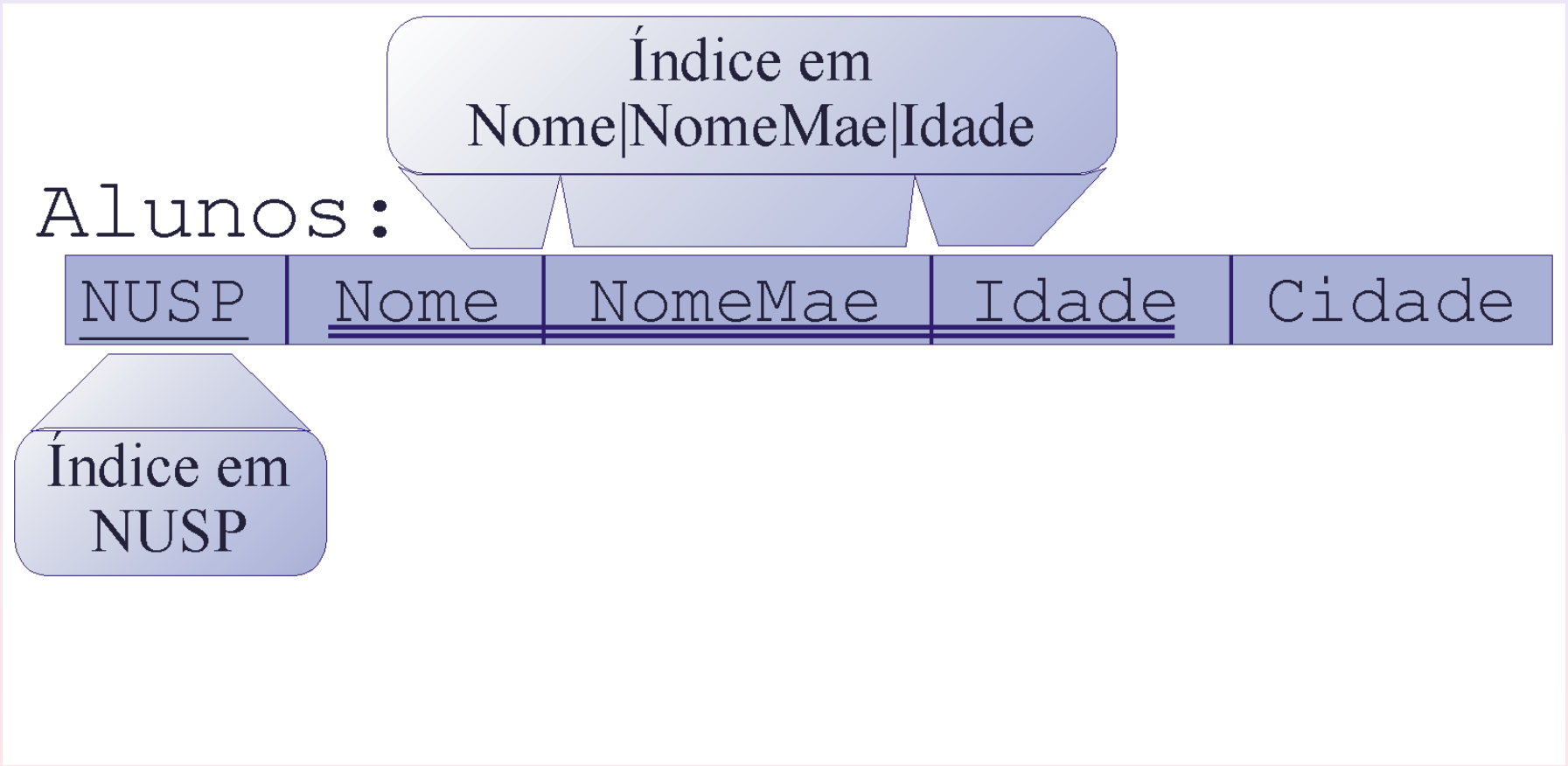
Podem ser criados muitos índices sobre uma mesma relação:




Índice para chave primária

# Declaração de Índices em SQL

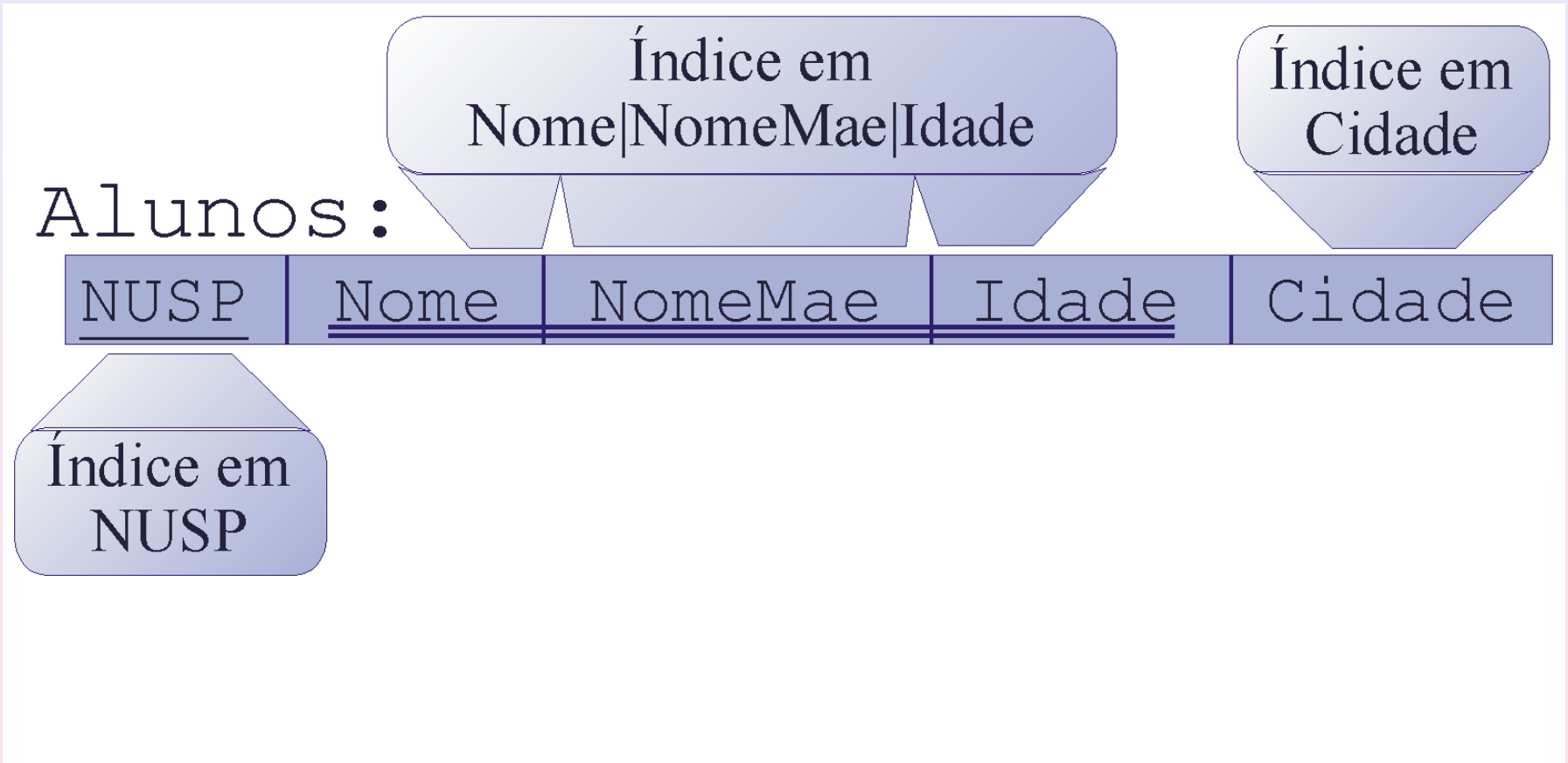
Podem ser criados muitos índices sobre uma mesma relação:




Índice para chaves candidatas

# Declaração de Índices em SQL

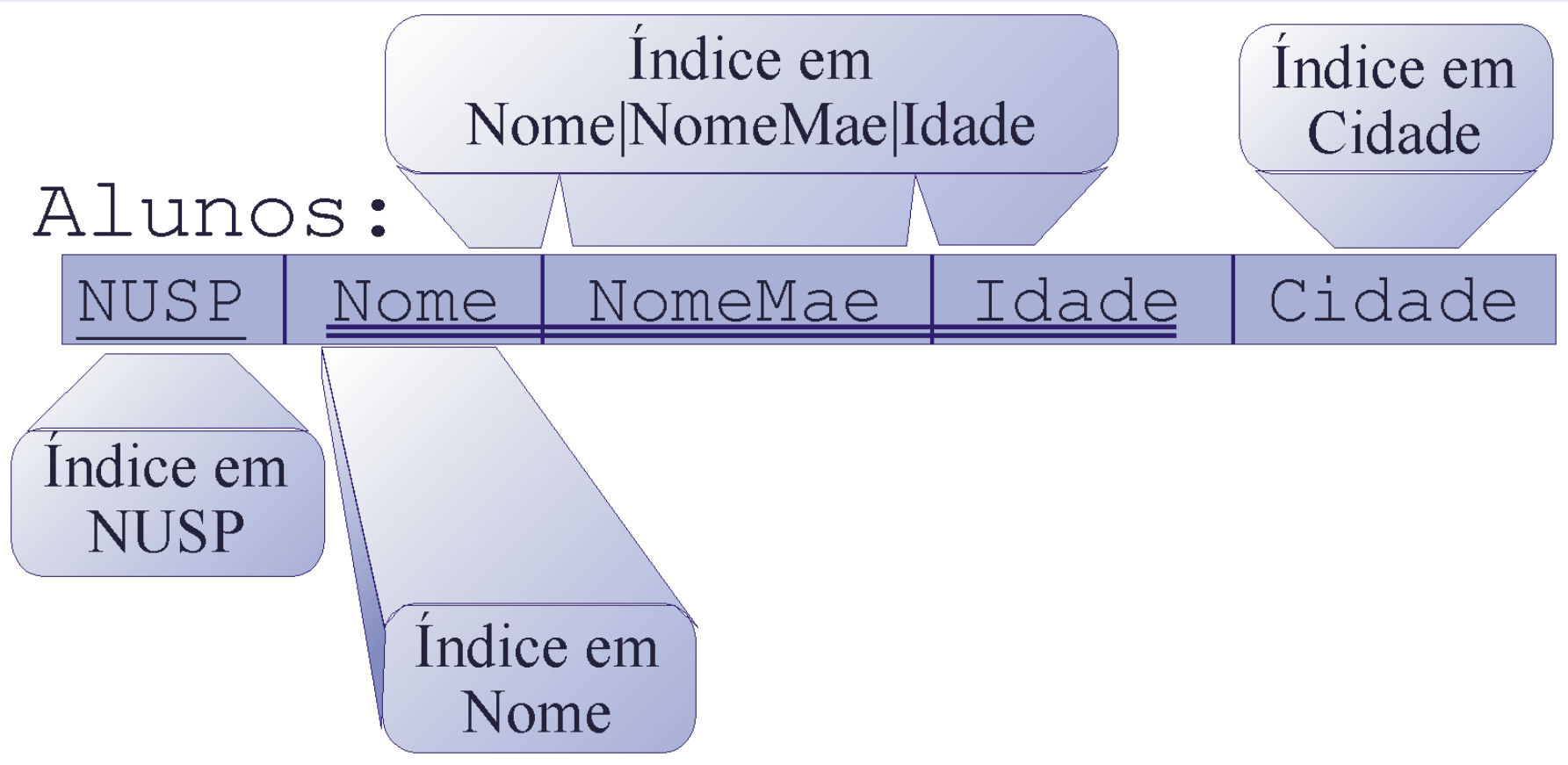
Podem ser criados muitos índices sobre uma mesma relação:




E para qualquer conjunto de atributos que tenha acesso frequente

# Declaração de Índices em SQL

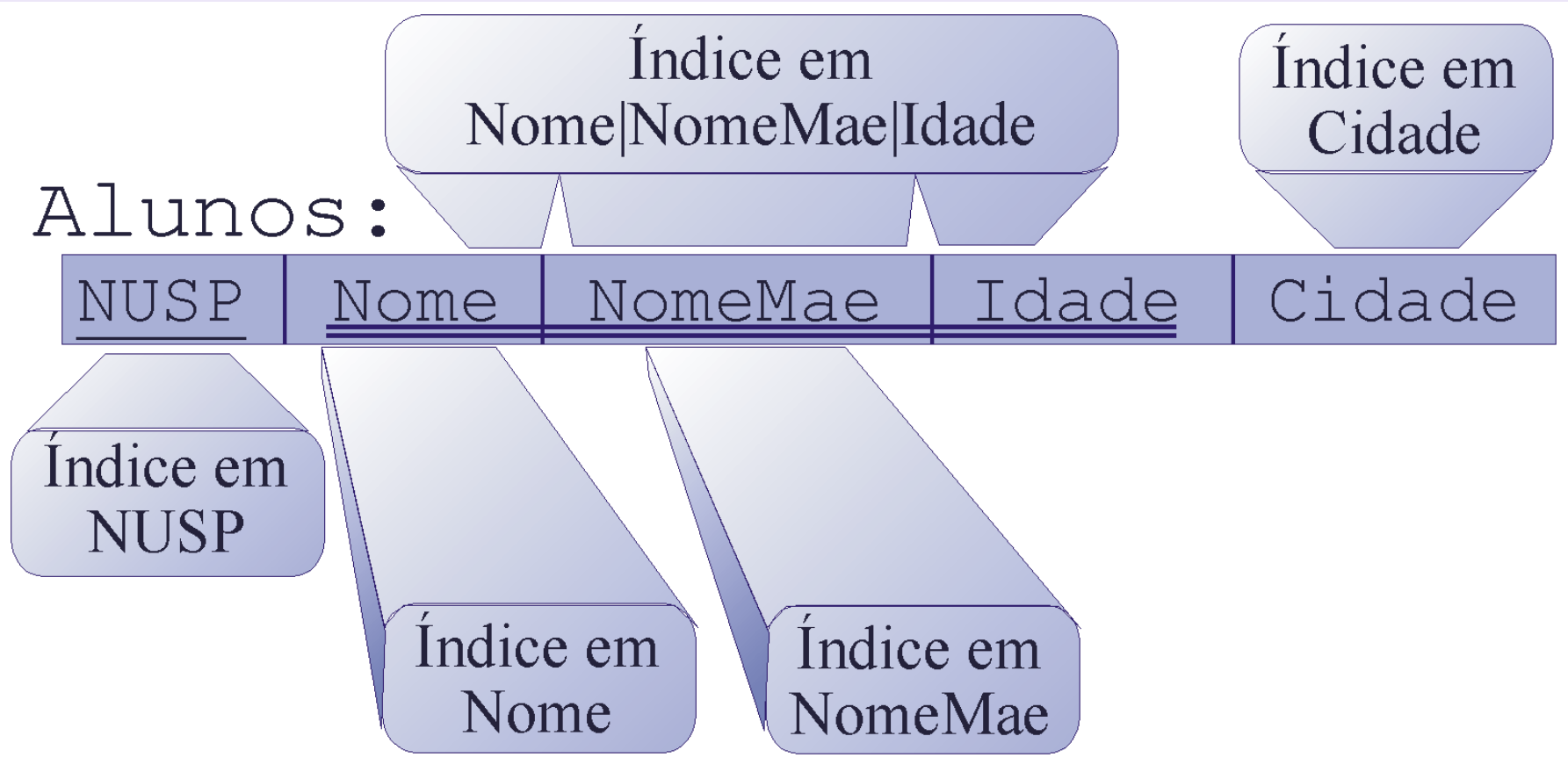
Podem ser criados muitos índices sobre uma mesma relação:




E para qualquer conjunto de atributos que tenha acesso frequente

# Declaração de Índices em SQL

Podem ser criados muitos índices sobre uma mesma relação:




E para qualquer conjunto de atributos que tenha acesso frequente.

# Índices parciais

Um índice não precisa indexar todos os dados de uma relação: ☞ **Índices parciais**

## CREATE INDEX

```
CREATE [UNIQUE | CLUSTERED] INDEX idx-name
    ON table [USING method]
    ({column | (expression)}
        [ASC|DESC][NULLS {FIRST|LAST}][, ...])
    [WITH (storage_parameter = value [, ... ])]
    [TABLESPACE tablespace]
    [WHERE predicate]
```

Índices com a cláusula `WHERE` indexam apenas as tuplas que atendem ao termo `predicate` indicado.

# Índices parciais – Exemplo

- Por exemplo, vamos assumir que a relação `Alunos` armazena as cidades de origem de cada aluno, e que a maioria é de São Paulo.
- Queremos selecionar todos os alunos de São Paulo cujo nome começa com `'Jose'`:

```
SELECT Nome
   FROM Alunos
   WHERE Nome LIKE '%Jose' AND
      Cidade = 'Sao Paulo'
```

- Se for criado o índice

```
CREATE INDEX NomeAluno ON Alunos(nome);
    WHERE Cidade = 'Sao Paulo'
```

Essa consulta irá utilizar o índice, com a vantagem que o índice será menor.

# Índices parciais – Exemplo

- Agora queremos selecionar todos os alunos que não são de São Paulo cujo nome começa com `'Jose'`:

```
SELECT Nome
   FROM Alunos
   WHERE Nome LIKE '%Jose' AND
      Cidade <> 'Sao Paulo'
```

- Essa consulta não pode utilizar o índice, pois ele não indexa nomes que não são de São Paulo.

# Índices parciais – Exemplo

- Veja que a seguinte consulta não pode usar esse índice parcial:

```
SELECT Nome
    FROM Alunos
    WHERE Nome LIKE '%Jose'
```

- Isso porque a consulta sobre os alunos '`Jose`' em geral não permite saber se algum aluno `Nome LIKE '%Jose'` é de São Paulo ou não, portanto alguns podem estar no índice, mas outros não.
- Ou seja, para que um índice parcial seja usado, é necessário que pelo menos os primeiros atributos indexados e os atributos da cláusula `WHERE` estejam em predicados da consulta.
- Nesse caso, somente um índice integral poderá ser usado.

# Conclusão: Índices em SGBDRs

- A indexação de dados é utilizada como uma maneira de acelerar o processamento de consultas em um SGBD Relacional;
- A idéia geral é organizar os dados numa estrutura de dados de maneira a não precisar acessar todos os dados, se parte deles puder ser descartada (podada) com certeza, sem precisar ser lida;
- Outro ponto importante é reduzir, e idealmente eliminar, a necessidade de re-ler dados que já tenham sido lidos para responder à mesma consulta;
- E finalmente, é importante levar em conta as características físicas do meio físico de armazenagem dos dados que possam afetar o desempenho das consultas.

O meio físico mais usado hoje são os DISCOS!

# Estrutura das páginas de dados

- Existem várias Estruturas de dados usadas para armazenar os dados nos segmentos de dados. Os mais utilizados são:
  - Como uma `HEAP`,
  - Como um `Índice primário`,
  - Como um `Índice secundário`.
- A estrutura mais utilizada, em ampla medida, é a estrutura em `HEAP`.

# Estrutura de páginas de dados em HEAP

- Numa estrutura HEAP, as tuplas são inseridas uma a uma nas páginas, sem qualquer ordem ou outra forma de organização.
- As tuplas vão sendo inseridas na página inicial, e quando ela atinge a capacidade especificada em `fillfactor`, outra página é inserida no mesmo *extent* (ou cria-se um *extent* de extensão no segmento da relação se o corrente já estiver todo preenchido), e a inserção continua da mesma maneira.
- Uma tupla é uma sequência contínua de bytes que concatena os valores dos atributos nessa tupla.

# Estrutura de páginas de dados em HEAP

- A estrutura típica de uma página de dados é a seguinte:


onde:

- **Header** identifica e indica a estrutura dessa página.
- **Data row** são as tuplas, armazenadas do final para o inicio da página.
- **Row Directory** indica o *offset* de cada tupla na página.
- O espaço livre fica no meio da página, pois não se sabe a priori quantas tuplas cabem por página.
- Se tuplas são removidas, o movimento de acomodar as tuplas é postergado para quando o espaço é necessário, e sempre fica restrito àquela página. Isso também reduz a necessidade de atualizar o **identificador das tuplas** em outras estruturas que apontam para ele.

# "Endereço" de uma tupla

- Um ponto importante a ser considerado é como localizar uma tupla na base de dados.
- Do ponto de vista do Modelo Relacional, uma tupla somente pode ser localizada a partir do valor de uma chave.
- Portanto, um usuário não deve ter acesso direto a nenhuma tupla da base a partir de seu "**Endereço**",
  - e SQL não tem nenhuma estrutura sintática para expressar ou obter esse endereço.
- No entanto, internamente, o SGBD pode acessar uma tupla por seu endereço...
  - $\cdots$ e em termos de programação, existem várias situações em que usar esse endereço é muito eficiente!
  - $\cdots$ e vários produtos permitem usar esse conceito!
- Oracle chama o "Endereço" de uma tupla de *Row Identification* – `RowId`. Esse termo é usado pela maioria dos fabricantes.

# "Endereço" de uma tupla

O que é um `RowId`?

- Recordando: Componentes da estrutura lógica de uma base de dados em disco:

Tupla $\xrightarrow{\texttt{HEAP}}$ Página $\xrightarrow{\text{Inteiro}}$ Relação $\xrightarrow{\text{Partição}}$ *Tablespace* $\rightarrow$ Base de Dados

- Então um `RowId` é uma expressão `ts#.seg#.pg#.t#`
- Em oracle, o `RowId` é uma string com esses 4 valores concatenados expressos em base 64 (versão *extended*).
- Postgres não permite expressar `RowId`.

# "Endereço" de uma tupla

Mas atenção:

- O gerenciador tem toda a liberdade de mudar o `RowId` quando ele precisar,
- portanto usar `RowId` é uma operação de risco!
- Ele somente é seguro se for usado em uma transação única `SERIALIZABLE`.

# "Endereço" de uma tupla

Quando é proveitoso usar RowId?

- Considere o seguinte programa:

```
exec sql SELECT Atr1 into :var
         FROM Tabela WHERE Chave=:ch;
result=processa(var)
exec sql UPDATE Tabela SET Atr2=:result
         WHERE Chave=:ch;
```

- Note-se que a tupla já foi localizada no comando SELECT, mas ela terá que ser localizada de novo, pelo valor de Chave, no comando UPDATE.

# "Endereço" de uma tupla

- Assumindo que o gerenciador ainda estará com a tupla em questão no *buffer* quando o comando UPDATE for emitido, o seguinte programa evita a segunda busca, e faz a atualização muito mais rapidamente:

```
exec sql SELECT Atr1, Tabela.ROWID into :var, :id
         FROM Tabela WHERE Chave=:ch;
result=processa(var)
exec sql UPDATE Tabela SET Atr2=:result
         WHERE Tabela.ROWID=:id;
```

# "Ponteiro" para uma tupla

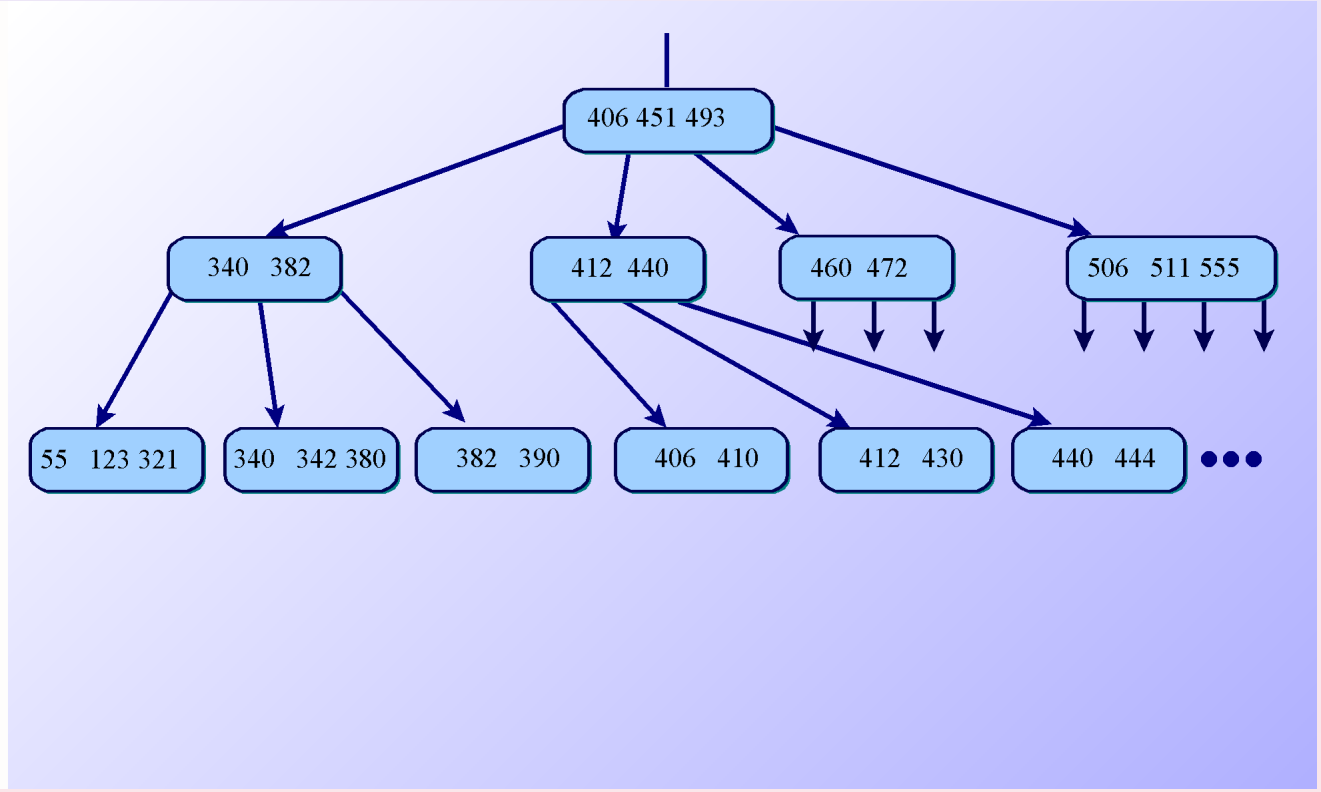
- O conceito de "endereço de tupla" será importante na discussão que se segue, mesmo que você não o use.
- O termo `RowID` é usado com frequência, sempre com a idéia de "apontar" para a posição no disco onde a tupla está armazenada fisicamente.
- Sempre que o `RowID` é usado fora de um contexto, deve ser indicada a expressão `ts#.seg#.pg#.t#`, por exemplo quando ele é passado para o usuário.
- Internamente, em geral o SGBD sabe o *tablespace* e o segmento a que o `RowID` se refere. Portanto, ele pode armazenar apenas a expressão `pg#.t#`, e isso ele o faz usando um único número.
- Nesta apresentação, sempre que o `RowID` é usado como um **Ponteiro para tupla** interno ao SGBD, assumimos que a expressão `pg#.t#` pode ser armazenada em 4 bytes.
- SGBD em geral usam 4 ou 6 bytes (em Oracle são chamados índices esparsos ou densos, respectivamente).

# Tipos de Índices

- Existem vários Tipos de Índices utilizados em SGBDs em geral:
  - Índices ISAM,
  - Índices de Arquivos invertidos,
  - Índices BitMap,
  - Índices *hash*,
  - Índices Multidimensionais.
- O índice mais utilizado é o índice ISAM, que na imensa maioria das vezes é implementado como uma B-tree.

# O Índice ISAM

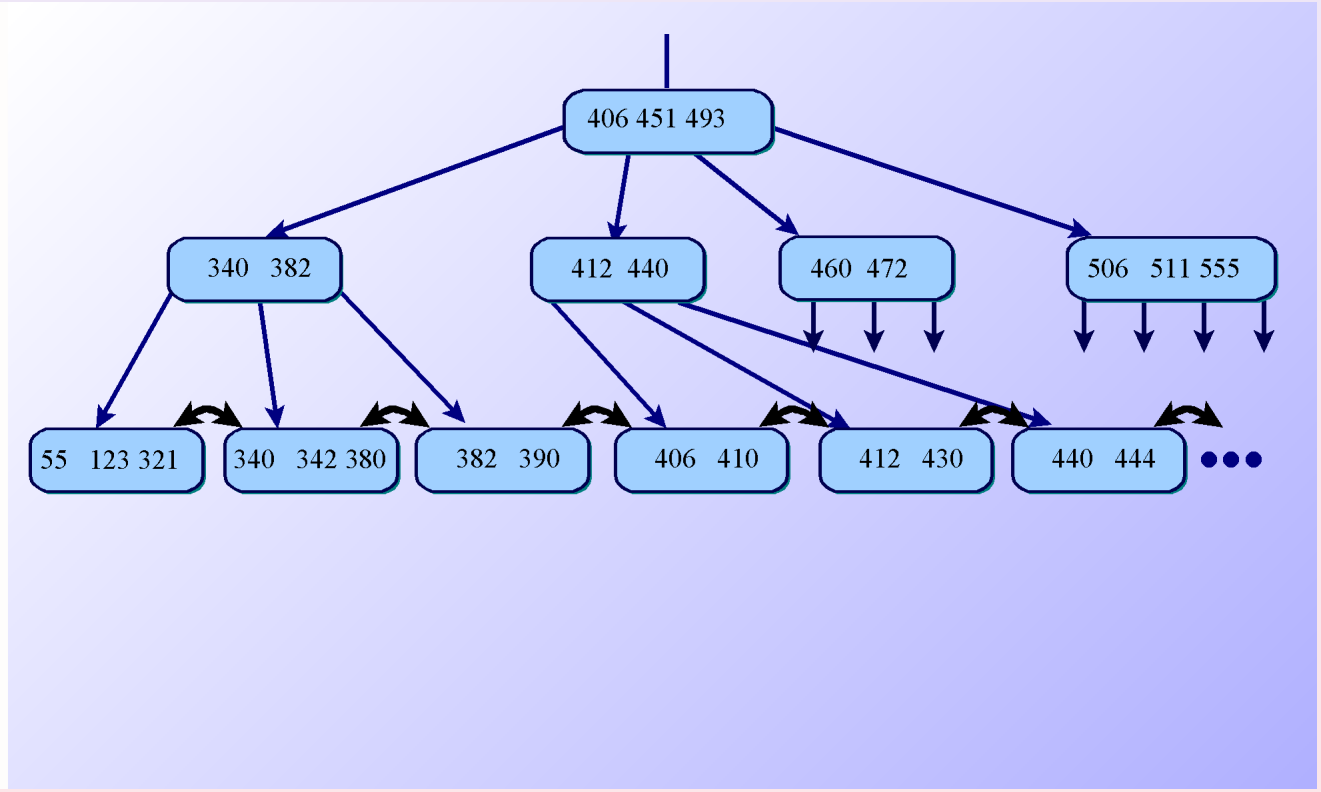
Como o nome indica, um índice B-tree usa uma estrutura de dados ···
$B^+$-tree ...




Uma $B^+$-tree é uma variação da B-tree, em que todas as chaves de acesso são colocadas nas folhas.

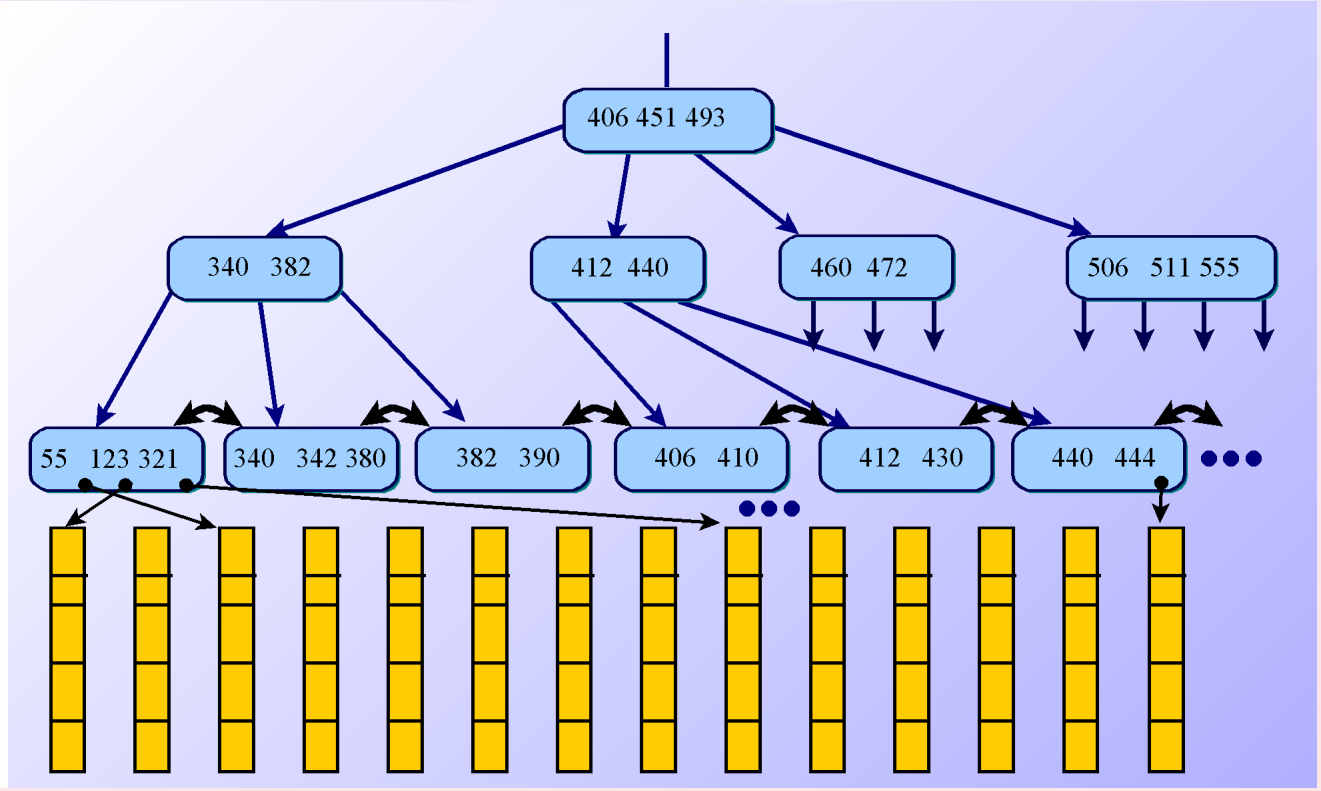
# O Índice ISAM

Como o nome indica, um índice B-tree usa uma estrutura de dados B$^+$-tree com folhas ligadas ☞ **Ponteiros para Irmãos**.



# O Índice ISAM

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

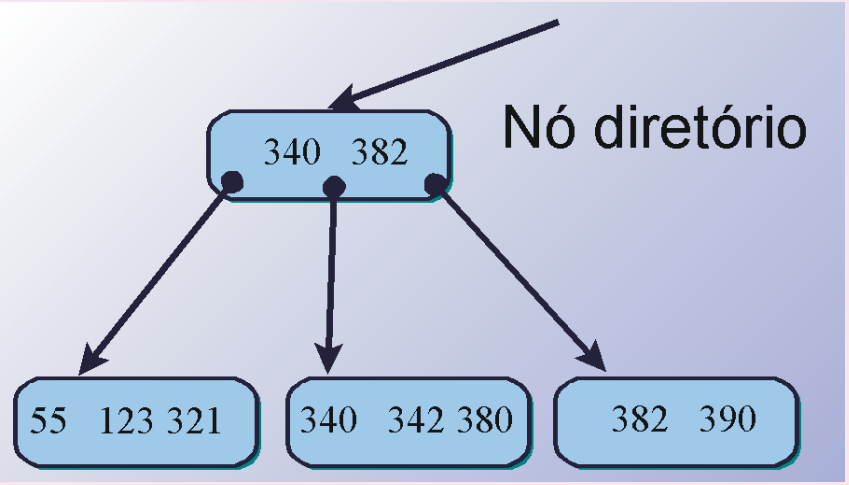


# O Índice ISAM

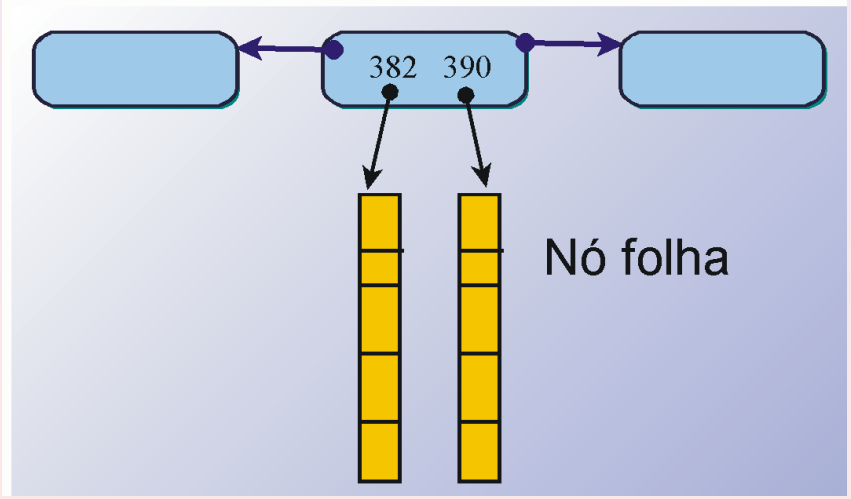
## Propriedades do índice B-tree

- 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.



# O Índice ISAM

## Propriedades do índice B-tree

- Uma B-tree usada na estrutura de memória física de um SGBD tem várias propriedades específicas:
  - Um nó é limitado pelo tamanho da página em disco: não existe um limite para o número de chaves de acesso que ele pode conter;
  - Como o tamanho das chaves de acesso varia (por exemplo, ela pode ter atributos `varchar`), cada nó pode ter um limite diferente;
  - Novas chaves vão sendo inseridas até que o nó fique `fillfactor` cheio, então ele quebra;
  - Portanto, cada nó fica sempre pelo menos 50% cheio;
  - Na realidade, um índice B-tree tem taxa média de ocupação de ≈71%;
  - O nó raiz sempre tem pelo menos duas chaves, caso contrário a árvore reduz de 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.

# O Índice ISAM

## Propriedades do índice B-tree

- Deve-se notar que as chaves de acesso em um nó são mantidas ordenadas também;
- Isso permite que a busca entre as chaves armazenadas em um nó possa ser feita usando o método da busca binária.

# O Índice ISAM

## Tamanho das B-trees

- Para ter uma idéia dos tamanhos envolvidos num índice B-tree, considere-se o seguinte exemplo:
- Seja um índice criado sobre a chave primária de uma relação com $N = 1.000.000$ de tuplas, em que a chave é um atributo `char(30)`, em uma base de dados com tamanho de página de 2KBytes.
  - Para simplificar as contas, considere que a página tenha 2.000 bytes livres para dados (48 bytes dos 2048 ficam para o *header*).
  - Cada chave de acesso armazenada num nó diretório gasta 30 bytes, mais 2 bytes de *offset* no diretório da página, mais 4 bytes para o ponteiro para o próximo nível: $30 + 2 + 4$=36 bytes;
  - então em um nó com `fillfactor` de 100% cabem $\lfloor (2000 - 4)/36 \rfloor = \lfloor 55,44 \rfloor = 55$ chaves de acesso;
  - Considerando a taxa média de ocupação de 71%, temos em média $\lceil 0,71 * 55 \rceil = \lceil 39,05 \rceil = 40$ chaves de acesso/nó diretório.

# O Índice ISAM

## Tamanho das B-trees

- Para ter uma idéia dos tamanhos envolvidos num índice B-tree, considere-se o seguinte exemplo:
- Seja um índice criado sobre a chave primária de uma relação com $N = 1.000.000$ de tuplas, em que a chave é um atributo `char(30)`, em uma base de dados com tamanho de página de 2KBytes.
  - Para simplificar as contas, considere que a página tenha 2.000 bytes livres para dados (48 bytes dos 2048 ficam para o *header* e o ponteiro extra das $n + 1$ chaves de acesso).
  - Cada chave de acesso armazenada num nó folha gasta 30 bytes, mais 2 bytes de *offset* no diretório da página, mais 6 bytes para o ponteiro para a tupla no segmento de dados: $30 + 2 + 6 = 38$ bytes;
  - então em um nó diretório com `fillfactor` de 100% cabem $\lfloor \frac{2000-2*4}{38} \rfloor = \lfloor 52,42 \rfloor = 52$ chaves de acesso (lembrar que cada nó folha tem dois ponteiros para os irmãos);
  - Considerando a taxa média de ocupação de 71%, temos em média $\lceil 0,71 * 52 \rceil = \lceil 36.92 \rceil = 37$ chaves de acesso/nó folha.

# O Índice ISAM

## Tamanho das B-trees

- 40 chaves de acesso/nó diretório, 37 chaves de acesso/nó folha e N=1.000.000 de tuplas:
  - Esse índice precisa de $\lceil 1.000.000/37 = \rceil = \lceil 27.027,02 \rceil = 27.028$ nós folha;
  - O último nível de nós diretório precisa de $\lceil 27.028/40 \rceil = \lceil 675,70 \rceil = 676$ nós diretórios;
  - O próximo nível de nós diretório precisa de $\lceil 676/40 \rceil = \lceil 16,89 \rceil = 17$ nós diretórios;
  - Dezessete nós são indexados por um único nó, a raiz.
  - portanto esse índice terá altura $H = 4$ e $27.028 + 676 + 17 + 1 = 27.722$ páginas em disco, correspondendo a 55.444 KBytes ou $\approx$ 56MBytes de espaço em disco.

# O Índice ISAM

## Acesso a um índice B-tree

- **ISAM** significa acesso **I**ndexado ou **S**equencial.
  - O acesso **I**ndexado requer navegar a árvore desde a raiz até a folha, lendo os $H$ 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 **S**equencial 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;
  - Portanto até faz sentido ler vários nós folha numa única operação de disco: pode ser interessante armazenar os nós folha em `extents` de tamanho maior do que um.
  - Mas isso não é implementado nos SGBDS atuais, pois um índice usa um único segmento, e ele é mantido com a propriedade de ter um nó por *extent*, o que é o adequado para os nós diretório e para a maioria dos acessos às folhas.

# O Índice ISAM

## Criando um índice B-tree

O Índice B-tree é o mais usado em SGBDs, e sempre é o default quando não especificado:

### CREATE INDEX – Postgres

```
CREATE [UNIQUE] INDEX idx-name ON table
    [USING {btree | hash | gist | gin}]
    ({column | (expression)} [ASC | DESC]
        [, [WITH (FILLFACTOR=value)]])
    [TABLESPACE tablespace]
    [WHERE predicate]
```

# Comando CREATE INDEX

## Oracle

### CREATE INDEX – Oracle

```
CREATE [UNIQUE] [BITMAP] INDEX idx-name ON table
    ({column | (expression)} [ASC | DESC][, ...])
    [TABLESPACE tablespace]
    [STORAGE ([INITIAL init-size] ...)]
       <outras...>
```

# O Índice ISAM

## Criação de um índice B-tree

- Um índice pode ser criado sobre uma relação vazia ou já alimentada.
- Inserção (ou substituição) de chaves de acesso em índices já existentes requerem a navegação da árvore, lendo uma página em cada um dos $H$ níveis.
  - Portanto, cada operação de `INSERT` ou `UPDATE` requer o acesso a pelo menos $H$ páginas do índice e pelo menos uma escrita.
- Inserir muitas tuplas num índice já existente pode ser uma operação demorada.
- Existe uma operação mais eficiente: a **carga rápida** (*bulk loading*).

# O Índice ISAM

## Criação de um índice B-tree

- Numa operação de **carga rápida**, o SGBD:
  - Armazena todas as tuplas a serem inseridas nas páginas de dados, e vai extraindo as chaves de acesso num *buffer* de trabalho, de preferência mantido inteiro na memória;
  - No exemplo anterior, 1.000.000 de chaves de acesso de 30 bytes cada uma requer 30MBytes de memória
  - A ordenação em memória pode ser feita em $O(N \cdot \log(N))$, portanto bastante rápida;
  - Tendo todas as chaves de acesso ordenadas, a B-tree é criada 'montando' cada nível da árvore: é feita uma varredura única da sequência de chaves de acesso ordenadas, escrevendo em sequência cada bloco de chaves de acesso que formam os nós-folha (no exemplo, blocos de 37 chaves de acesso por folha), e também em sequência cada nível da árvore;
  - Portanto, a árvore é criada com exatamente uma operação de escrita individual em cada nó diretório, e uma operação de escrita em cada *extent* que forma a sequência de nós folha.

# O Índice ISAM

## Criação de um índice B-tree

- Recomendação: quando for necessário carregar ou alterar muitas tuplas em uma relação, é preferível
  - desativar os índices,
  - carregar os dados,
  - reativar os índices.

# O Índice ISAM

## Chaves de acesso duplicadas

Recordando, um comando de criação de índices tem o seguinte formato:

### CREATE TABLESPACE – Postgres

```
CREATE [UNIQUE] INDEX name ON table
    {column | (expression)} [ASC | DESC] [,...]
```

- A cláusula `UNIQUE` indica que a chave de acesso do índice não pode ter valores repetidos. Sem ela, o índice pode ter chaves de acesso repetidas.
- A cláusula `UNIQUE` é usada para impor que os atributos indicados no índice são chave candidata ou primária para a relação.
- A discussão levada até aqui considera que o índice não pode ter chaves de acesso repetidas.
- Para permitir que chaves repetidas possam ser indexadas numa B-tree, devem ser tomados alguns cuidados.

# O Índice ISAM

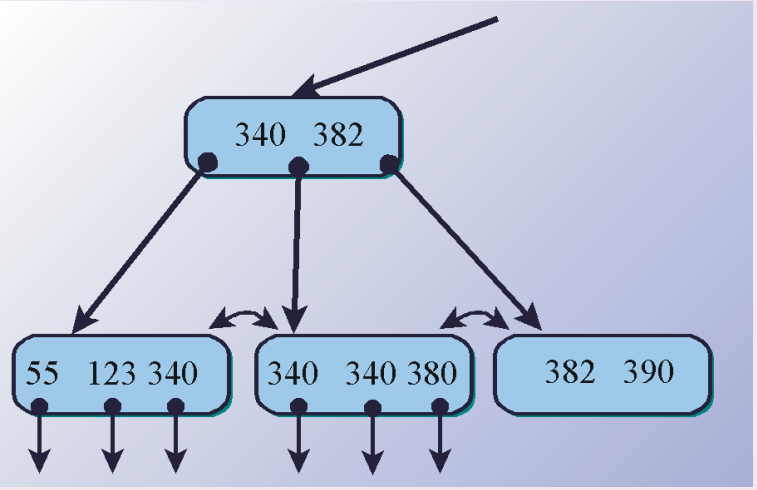
## Chaves de acesso duplicadas

- A princípio, para aceitar chaves duplicadas basta permitir armazenar mais de um par `<chave, ponteiro>` que tenha o mesmo valor da chave.
- A busca de qual sub-árvore descer para encontrar uma chave de acesso numa B-tree procura o primeiro valor de roteamento maior ($>$) do que a chave procurada e desce pelo ponteiro anterior a esse valor;
- em um índice `UNIQUE`, deve-se procurar por maior ou igual ($\geq$).

# O Índice ISAM

## Chaves de acesso duplicadas

- Mas um cuidado adicional deve ser tomado: pode acontecer que uma sequência de chaves repetidas comece num nó folha e termine em outro.




- Para cuidar disso, é necessário que, quando um índice não for `UNIQUE` e a chave for a primeira de um nó, deve-se fazer um *backtracking* usando o ponteiro para o irmão anterior, até achar a primeira chave menor do que a que está sendo buscada.

# O Índice ISAM

## Chaves de acesso duplicadas

- Note-se que, a menos do *backtraking* para localizar o início de um bloco de chaves repetidas, a navegação sequencial a partir de um valor localizado por um acesso indexado é uma operação regular que o SGBD deve realizar para acessar uma faixa de valores, por exemplo a partir de um critério `BETWEEN x AND y`.

# O Índice ISAM

## Remoção de chaves de acesso

- Chaves duplicadas também precisam de cuidado especial nas operações de remoção de chaves de acesso:
  - Lembrar que uma entrada em um nó índice tem o formato `<valor, rowid>`.
  - Chaves `UNIQUE` precisam apenas do `valor` para que sejam identificadas.
  - Chaves duplicadas precisam do par `<valor, rowid>` completo para que a chave e o ponteiro corretos sejam removidos.
  - A tática para facilitar a busca de um par `<valor, rowid>` é manter a coleção de entradas com mesmo valor da `chave` ordenadas pelo valor do `rowid`.
- ☞ Mantendo o par `<valor, rowid>` ordenado tanto nos nós folha quanto nos nós diretório, o índice volta ser `UNIQUE`!
- Chaves que repetem não precisam ser armazenadas mais de uma vez. Assim, uma sequência de pares `<valor, rowid1>...<valor, rowidn>` pode ser reduzido a `<valor, <rowid1 ... rowidn> >`, em que a sequência `<rowid1 ... rowidn>` é comumente chamada *RID List*.

# Índices *Prefix-B-tree*

- Frequentemente, em atributos de tipo `char` (ou `varchar`), os caracteres iniciais de uma chave tendem a se repetir bastante, mesmo em índices `UNIQUE`.
- O mesmo efeito ocorre em chaves compostas por mais de um atributo, onde o primeiro atributo se mantém constante enquanto os demais tipicamente variam em diversas tuplas (lembre-se que chaves de relações compostas geram valores concatenados para as chaves de busca).
- Para poupar espaço e aumentar a quantidade de chaves que cabem em um nó, uma técnica é não armazenar os caracteres iniciais que se repetem no inicio de chaves subsequentes.
- Armazena-se apenas um contador de quantos caracteres se repetem, e o sufixo que muda.
- Essa estrutura é chamada *Prefix-B-tree*.

# Índices *Prefix-B-tree*

- A manutenção de chaves de acesso numa *Prefix-B-tree* traz problemas equivalentes à manutenção de índices com chaves repetidas, mas as soluções são simples.
- Sequências em que a chave inteira está em um nó anterior são evitadas colocando-se a chave inteira na primeira entrada do nó.
- A remoção da primeira chave de uma sequência deve restaurar a chave inteira na entrada seguinte.
- Tratamento equivalente é feito para a operação de inserção.

# Índices *Prefix-B-tree*

## Chaves de acesso em índices compostos

- A *Prefix-B-tree* também propicia ganhos de espaço significativos quando ela é usada em índices compostos por mais de um atributo.
- Índices com mais de um atributo têm como chave de acesso a concatenação dos valores de todos os atributos.

# Índices *Prefix-B-tree*

## Chaves de acesso em índices compostos – Exemplo

- Por exemplo, considere-se a relação de matrículas de alunos do ICMC em disciplinas:

```
CREATE TABLE Matriculas (
    Disciplina CHAR(8) NOT NULL,
    NUSP DECIMAL(10) NOT NULL,
    Nota DECIMAL(4,2),
    PRIMARY KEY (Disciplina, NUSP) )
```

- Vamos considerar que existam 1.000 alunos e 160 disciplinas, cada aluno se matriculando numa média de 8 disciplinas. Então haverá 8.000 tuplas na relação `Matrícula`.
- Vamos considerar também que o tamanho da página seja de 2KBytes, assumindo 2.000 bytes para dados.

# Índices *Prefix-B-tree*

## Chaves de acesso em índices compostos – Exemplo

- A indicação de `PRIMARY KEY` induz a criação de um índice B-tree `UNIQUE`.
- Cada chave é a concatenação de uma `Disciplina` e um `NUSP`, totalizando $8 + 10/2 = 13$ bytes.
- Cada entrada de nó folha totaliza (chave + diretório + RowId) $13 + 2 + 6 = 21$bytes.
- Então cabem $\lfloor 0.71 \cdot \frac{2.000-2*4}{21} \rfloor = \lfloor 65,35 \rfloor = 65$ entradas por nó (cada nó folha tem dois ponteiros irmãos).
- A B-tree terá $\lceil 8.000/65 \rceil = \lceil 123,08 \rceil = 124$ nós folha.

# Índices *Prefix-B-tree*

## Chaves de acesso em índices compostos – Exemplo

- Considerando 8.000 matrículas em 160 disciplinas, temos uma média de $8.000/160 = 50$ alunos matriculados em cada disciplina.
- Portanto, pelo menos os 8 primeiros caracteres de cada chave se repetem em média 50 vezes por disciplina.
- Se for usada uma *Prefix-B-tree*, tuplas que sucedem outras matrículas na mesma disciplina não irão repetir o valor do atributo `Disciplina`, e terão uma chave com os 5 bytes do atributo `NUSP` mais o byte contador de repetições, num total de 6 bytes.
- Cada entrada repetida totaliza(chave + diretório + rowid) $6 + 2 + 6 = 14$ bytes.
- Então cabem $\lfloor 1.992/14 \rfloor = \lfloor 142, 29 \rfloor = 142$ entradas repetidas por nó.
- Mas serão 160 entradas sem repetição de 21 bytes e $8.000 - 160 = 7.840$ entradas com repetição de 14 bytes.
- Ou seja, em média serão $\left\lceil \frac{(160*21+7.840*14)}{8.000} \right\rceil = \lceil 14, 14 \rceil = 15$ bytes em média por chave.
- Então cabem $\lfloor 0,71 \cdot 1.992/15 \rfloor = \lfloor 94, 29 = 94 \rfloor = 94$ entradas em média por nó.
- A *Prefix B-tree* terá $\lceil 8.000/94 \rceil = \lceil 85, 11 \rceil = 86$ nós folha.

# Índices *Prefix-B-tree*

## Chaves de acesso em índices compostos – Exemplo

- Portanto, nesse exemplo, uma B-tree usaria 124 nós folha, e uma *Prefix-B-tree* usaria apenas 86 nós folha.
- Essa diferença seria maior se o tamanho do primeiro atributo fosse maior,
- e maior ainda se fossem mais do que dois atributos concatenados na chave.
- Normalmente, o usuário não precisa (nem tem como) indicar um índice *Prefix-B-tree*:
  o SGBD assume essa variante do índice B-tree automaticamente.

# O Índice de Arquivos invertidos

- Um dado que possa ser ordenado e seja chave usa um índice B-tree `UNIQUE`.
- Um dado que possa ser ordenado e tenha poucos valores repetidos usa um índice B-tree não `UNIQUE`.
- Mas um dado que possa ser ordenado e tenha muitos valores repetidos requer uma outra variação da B-tree: o **Arquivo invertido**.
- Esse índice é usado, em geral, também de maneira automática pelo SGBD, quando a cardinalidade do índice é pequena, e o conteúdo da relação é bastante estável.

# O Índice de Arquivos invertidos

- Um índice *prefix-B-tree* não `UNIQUE` terá as chaves repetidas reduzidas ao ponteiro `RowID`, já que o contador de repetição degenera para "Tudo repetido", e portanto pode ser eliminado.
- Nesse caso, uma *prefix-B-tree* indexando um conjunto de chaves de baixa cardinalidade mas com grande quantidade de tuplas acaba se tornando uma sequência de poucas chaves de acesso, cada uma seguida de uma longa lista de `RowID`, chamada **RID List**.
- Isso significa também que nos registros de dados da relação, muitas tuplas terão valores repetidos.
- Para poupar espaço também na armazenagem das tuplas, quando o valor de um atributo ocupa muitos bytes, uma técnica é trocar o valor pelo índice da chave no índice: como a cardinalidade é pequena, um ou no máximo dois bytes indexam o valor do atributo no índice.
- Esse "ponteiro ao contrário" cria o que é chamado **Arquivo Invertido**.

# O Índice de Arquivos invertidos

## A Estrutura

- Portanto, um Arquivo invertido é:
  - ☞ Uma lista de valores de chaves de acesso,
  - ☞ com cardinalidade reduzida (tipicamente $< 1.000$);
  - ☞ A lista de chaves de acesso é mantida ordenada por uma B-tree e é associada a um **"número da chave"** sequencial;
  - ☞ Cada chave de acesso tem uma *RID list*, usualmente longa;
  - ☞ O valor do atributo nas tuplas é trocado pelo número da chave correspondente;
  - ☞ Cria-se um segundo índice, um *Hash*, para associar o número da chave com o valor da chave.

# O Índice de Arquivos invertidos

## A Estrutura

- Quando um atributo é associado a um arquivo invertido, o acesso à tupla não trás o valor do atributo, o qual precisa ser então obtido no arquivo invertido.
- Isso reduz o tamanho da tupla, permitindo armazenar mais tuplas por registro, o que reduz a necessidade de memória para acessar essa relação.
- Sempre que essa relação for acessada, o índice de Arquivo Invertido tem que ficar na memória.
- Arquivos invertidos permitem manter diversas estatísticas interessantes (por exemplo: histogramas), como um benefício adicional, sendo usados em atributos com valores que ocupam muitos bytes em domínios de pequena cardinalidade.

# O Índice de Arquivos invertidos

## A Estrutura

- Um índice invertido é criado como usando uma estrutura B-tree ou *hash*, e o arquivo invertido é gerado quando se solicita que a estimativa de seletividade do atributo opere em nível '3 – Histograma'.

```
CREATE TABLE Alunos (
    NUSP CHAR(10) PRIMARY KEY,
    Nome CHAR(50),
    Idade DECIMAL(3),
    Cidade CHAR(40) );
CREATE INDEX IdxCidade on Alunos USING BTREE
    (Cidade);
 . . .
ANALIZE TABLE Alunos
    ESTIMATE SYSTEM STATISTICS
    COLUMNS (Cidade);
```

# O Índice de Arquivos invertidos

## Exemplo

- Por exemplo, considere uma relação de alunos da USP, que tem $\approx 80.000$ alunos em uma base de dados com página de 2KB (2.000 bytes para dados):
  `Alunos={NUSP, Nome, Idade, Cidade}`
- Vamos assumir que o tamanho da tupla (incluindo o ponteiro no diretório) seja em média 40 bytes: ☞ cada página armazena $2000/40 = 50$ tuplas,
- e a relação ocupa $80.000/50 = 1600$ páginas.
- Vamos assumir que o atributo `Cidade`, que indica a cidade de origem do aluno terá poucos valores, tenha tamanho médio de 15 bytes.
- Se o atributo `Cidade` for colocado em um arquivo invertido, seus 15 bytes em média serão substituídos pelos dois bytes do número da Cidade.
- A tupla ocupará em média $40 - 15 + 2 = 27$ bytes e cada página armazenará $\lfloor 2000/27 \rfloor = \lfloor 74,07 \rfloor = 74$ tuplas em média.
- A relação ocupará $\lceil 80.000/74 \rceil = \lceil 1081,08 \rceil = 1082$ páginas – um ganho de 518 páginas, ou quase um terço.
- O índice armazenará em média $\left\lfloor \frac{0,71 \cdot 1996}{15+2+6} \right\rfloor = \lfloor 61,62 \rfloor = 61$ chaves de acesso por página: Possivelmente terá apenas o nó raíz, ou no máximo 4 ou 5 nós folha mais a raiz (sem contar a *RID list*, que não precisa ficar na memória).

# O Índice BitMap

- Uma alternativa para o arquivo invertido quando existem muitos valores repetidos e cardinalidade bem pequena é o índice BitMap.
- O índice BitMap requer a existência de um par de funções que possa associar o *RowID* de cada tupla com a posição ordinal da tupla na relação armazenada e vice versa.
- A idéia do índice BitMap é criar um "*bit map*" para cada valor de chave de acesso onde o bit tem o valor '1' para a tupla que tem aquele valor no atributo (ou atributos) do índice e '0' se não tem aquele valor.

# O Índice BitMap

- Portanto um BitMap terá a estrutura:

```
chave1: 00011000000100000100010...
chave2: 00100000010000010010000...
    demais chaves.
```

- As chaves são indexadas seguindo uma B-tree, mas nas folhas, ao invés de haver o ponteiro `RowID`, estará armazenado o *bit map*.
- Note-se que uma estrutura *bit map* pode ser encarada como uma variante da B-tree, tal como um arquivo invertido.

# O Índice BitMap

## Exemplo

- Por exemplo, considere uma relação de Professores da USP, que tem $\approx 8.000$ professores em uma base de dados com página de 2KB (2.000 bytes para dados):
  `Professores={NUSP, Nome, Idade, Nivel, Ativo}`
- O atributo `Nivel` indica se o professor é nível `MS-1`, `MS-2`, `MS-3`, `MS-5` ou `MS-6`, portanto tem cardinalidade 5.
- O atributo `Ativo` indica se o professor está na ativa (`Ativo=S`) ou aposentado (`Ativo=N`).
- Criando um índice BitMap sobre atributo `Nivel`:
  ```
  CREATE BITMAP INDEX IdxNivel ON Professores (Nivel)
  ```
- cada chave ocupa 4 bytes, e cada *bit map* ocupa $8.000/8 = 1.000$ bytes.
- O índice ocupa 5*(chave+*bitmap*+diretório)$=5 * (4 + 1.000 + 4 = 5040)$ bytes nas folhas, e portanto usa 3 páginas nas folhas mais um nó raiz.

# O Índice BitMap

## Uso do índice

- A grande vantagem do índice BitMap é quando ele pode ser usado para resolver quais tuplas atendem aos critérios que envolvem atributos indexados por BitMap sem acessar as tuplas propriamente ditas.
- Por exemplo, considere a consulta "Quantos professores `MS-3` e `MS-5` existem?". Ela pode ser respondida com o comando:

```
SELECT Count(*)
    FROM Professores
    WHERE Nível='MS-3' OR Nível='MS-5'
```

- Para responder a essa consulta, basta acessar o índice `IdxNivel` e contar quantos bits estão ligados no *bit map* correspondente aos valores `'MS-3'` e `'MS-5'`.

# O Índice BitMap

## Uso do índice

- A comparação pode incluir *bit maps* de qualquer atributo da relação.
- Por exemplo, se for criado outro índice BitMap para o atributo `Ativo`:

```
CREATE BITMAP INDEX IdxAtivo ON Professores (Ativo)
```

- então uma consulta que pergunta quantos professores 'Auxiliar de Ensino' ou 'Mestre' 'Aposentados' existem:

```
SELECT Count(*)
   FROM Professores
   WHERE (Nível='MS-1' OR Nível='MS-2') AND Ativo='N';
```

- Será respondida comparando com OR e AND os bits correspondentes nos BitMaps `IdxNivel` e `IdxAtivo`.

# O Índice BitMap

## Densidade do BitMap

- Assumindo que todos os valores do atributo *Atrib* de um BitMap estão presentes em aproximadamente a mesma quantidade nas tuplas, então cada *bit map* terá em média $\frac{1}{|Atrib|}$ bits ligados, e todos os demais desligados.
- A proporção de bits ligados é chamada a **densidade** do BitMap.
- BitMaps que tenham uma densidade pequena são chamados **índices esparsos**, e os que têm densidade grande são chamados **índices densos**.
- Mas o que significa ser **denso** ou **esparso**?

# O Índice BitMap

## Densidade do BitMap

- Vamos considerar três índices de arquivo invertido: `IdxAIA`, `IdxAIB` e `IdxAIC`, cada um indexando um atributo de tipo `CHAR(10)` com cardinalidade 32, 48 e 64 respectivamente, sobre uma relação com 10.000 tuplas.
- Cada índice ocupa o espaço para as chaves e mais 6 bytes para o *RowID* por tupla.
- O número de *RowID*s é constante para os três índices: cada um deles estará associado a alguma das chaves exatamente uma vez.
- Portanto, cada índice ocupará 60.000 Bytes mais o espaço para armazenar as chaves:
  `IdxAIA` ocupará $60.000 + 320 = 60.320$ bytes,
  `IdxAIB` ocupará $60.000 + 480 = 60.480$ bytes e
  `IdxAIC` ocupará $60.000 + 640 = 60.640$ bytes,
  ou seja, todos terão $\approx 60$ KBytes.

# O Índice BitMap

## Densidade do BitMap

- Ao invés de arquivos invertidos, vamos considerar agora três índices BitMap: `IdxBMA`, `IdxBMB` e `IdxBMC`, indexando os mesmos atributos de cardinalidade 32, 48 e 64 respectivamente, sobre a relação com 10.000 tuplas.
- Cada índice ocupa o espaço para as chaves e mais um *bit map* de $\frac{10.000}{8} = 1.250$ bytes por chave. (8 bits por byte)
- O tamanho de cada *bit map* é constante para cada chave, assim o tamanho de cada índice é proporcional ao número de chaves.
- Portanto, para cada chave o índice ocupa 1.250 bytes mais os 10 bytes da chave.

  `IdxBMA` ocupará $32 * (1.250 + 10) = 40.320$ bytes,
  `IdxAIB` ocupará $48 * (1.250 + 10) = 60.480$ bytes e
  `IdxAIC` ocupará $64 * (1.250 + 10) = 80.640$ bytes,

  ou seja, o indice aumenta com a cardinalidade da chave.

# O Índice BitMap

## Densidade do BitMap

A tabela seguinte compara o tamanho dos Índices:

| | Arq.Invertido | Bit Map |
|---|---|---|
| $\lvert\texttt{AtrA}\rvert = 32$ | 60.320 | 40.320 |
| $\lvert\texttt{AtrB}\rvert = 48$ | 60.480 | 60.480 |
| $\lvert\texttt{AtrC}\rvert = 64$ | 60.640 | 80.640 |

☜ =

- Note-se que o total de bits ligados em todos os *bit maps* de um índice BitMap é igual ao número de tuplas da relação.
- Portanto, quanto maior a cardinalidade do índice, menor a densidade.
- Assume-se que quando o tamanho do índice BitMap é maior do que o índice em Arquivo invertido correspondente, então o índice é esparso, senão é denso.
- O tamanho dos dois índices igualam-se quando o número de chaves é igual ao número de bits de um *RowID*.

# O Índice BitMap

## Índices BitMap Esparsos

- Índices esparsos podem ser **comprimidos**.
- Técnicas de compressão permitem comparar os bits e aplicar os operadores `AND`, `OR` e `NOT` sobre os *bit maps* comprimidos (evitando ter que descomprimir para comparar).
- Índices comprimidos sobre atributos que se repetem tão pouco quanto duas a três vezes em média podem ser comprimidos para índices de tamanho equivalentes ao de um índice em arquivo invertido correspondente.
- Portanto Índices BitMap densos não são comprimidos e ocupam menos espaço do que um arquivo invertido;
- índices BitMap esparsos são comprimidos e ocupam aproximadamente o mesmo espaço de um arquivo invertido.

# O Índice BitMap

## Vantagens/Desvantagens de usar BitMap

- Existem quatro vantagens em usar índices BitMap:
  - Ele usa pouco espaço.
  - Ele responde com facilidade consultas que envolvem conjunções e disjunções de comparações por igualdade.
  - Ele auxilia a filtrar dados em índices *clustered* (a ser visto em breve).
  - Um índice BitMap é muito útil para responder a operações de contagem (`count()`).
- e existe uma desvantagem em usar índices BitMap esparsos:
  - A atualização de um *bit map* comprimido é trabalhosa.

# O Índice *Hash*

## Conceito

- Um índice *Hash* usa uma função de distribuição (função *Hash*) $Hash(Ch_i)$ que atribui um número $0 < i \leq N, i \in \mathbb{N}$ a cada valor $Ch_i$ da chave de acesso, sendo que $N$ é o número máximo de chaves de acesso esperadas.
- Idealmente, havendo $N$ chaves diferentes, a função *Hash* deveria estabelecer um número $i$ para a chave $Ch_i$ que não é compartilhado por nenhuma outra chave $Ch_j, j \neq i$.
- Dessa maneira, dada uma chave $Ch_i$, calculando $Hash(Ch_i)$ indica diretamente a posição ordinal onde uma tupla que contém esse valor de chave está armazenada.
- Cada posição ordinal corresponde a uma célula na estrutura de dados, que armazena a chave e o par `<valor, rowid>`,
- ou no caso de uma estrutura em disco, a uma coleção de células, usualmente chamado um **bucket** que cabe em uma página de dados.

# O Índice *Hash*

## Usando um Índice *Hash* num SGBD

- Para ser usado em um SGBD, um índice *Hash* requer que sejam resolvidas algumas questões:
  - Como o SGBD sabe qual o valor de $N$?
    ☞ Não tem como saber. O analista tem que especificar quando criar o índice.
  - Qual a função $Hash(Ch_i)$ a ser usada em cada tipo de dados da chave $Ch_i$?
    ☞ O SGBD tem uma função default, que calcula o *checksum* dos bytes que compõem a chave e aplica a função módulo da divisão do *checksum* com o primeiro número primo maior ou igual a $N$.
  - Como tratar funções imperfeitas que associem o mesmo valor $Hash(Ch_i) = Hash(Ch_j)$ para duas chaves distintas (colisão)?
  - Como tratar índices não `UNIQUE`?

# O Índice *Hash*

## Como tratar colisões e chaves repetidas

- As duas últimas questões são resolvidas pelo mesmo mecanismo:
  - Como tratar colisões?
  - Como tratar chaves repetidas?
- Sempre que duas chaves forem apontadas para uma mesma célula (ou `bucket`), deve haver uma área de memória de **inundação**, capaz de armazenar todas as chaves mapeadas para essa célula.
- Existem 2 maneiras de fazer isso, que são usadas juntas:
- Primeiro reserva-se espaço nas páginas usando `FillFactor` (ou `PCTFREE/PCTUSED`) para assimilar a maioria das colisões e chaves repetidas previsíveis;
- Segundo, cria-se uma lista de páginas de *overflow*, chamada **String de Páginas** com a coleção de todos os *buckets* associados a essa célula.

# O Índice *Hash*

## Como tratar colisões e chaves repetidas

- A reserva de espaço não deve ser exagerada, sob pena de se gastar muito espaço em disco que não é utilizado.
- Por outro lado, se houver muito pouco espaço e muitas das chaves precisarem de páginas extras, a *string* de páginas fica longa e a recuperação de dados pode requerer várias operações de acesso ao disco.

# O Índice *Hash*

## Quando usar índices *Hash*

- Um índice Btree bem "afinado" permite acessar uma página de diretório por nível da árvore (usualmente 3 ou 4) e uma página de dados.
- Um índice BitMap ou Arquivo invertido bem "afinado" permite acessar apenas uma ou duas páginas de diretório e uma página de dados.
- Um índice *Hash* bem "afinado" permite acessar apenas uma página de diretório e uma página de dados.
- Note-se que o acesso a páginas de diretório em qualquer índice é sempre aleatório: quase não é possível tirar proveito do esquemas de *extents*.
- Portanto, em situações críticas em que a velocidade de acesso é fundamental, é importante reduzir ao máximo o número de acessos às páginas de diretório.
- Mas índices *Hash* não permitem ordenações.

# O Índice *Hash*

## Quando usar índices *Hash*

- Quando a estrutura usada nos segmentos de dados é uma `HEAP`, pelo menos uma página de diretório sempre tem que ser acessada.
- Por isso, índices *Hash* são usados preferencialmente em estruturas que exploram um índice primário.

# Índices Primários e Secundários

- Conceitualmente, um índice serve para organizar um conjunto de dados, e os dados deveriam ser encontrados ao final da navegação nos índices
- Mas em SGBD, os índices são usados não para organizar os dados, que são as tuplas, mas para organizar chaves de acesso, que em geral são valores de um sub-conjunto dos atributos que forma cada tupla.
- Índices que organizam os dados diretamente são chamados **índices primários**;
- Índices que organizam ponteiros para os dados são chamados **índices secundários**.
- Quando as tuplas são armazenadas em qualquer ordem, diz-se que elas são organizadas em **Heap**.
- Se elas forem organizadas para ficarem fisicamente na mesma ordem do diretório, diz-se que elas são organizadas em índice secundário.
- Se elas forem armazenadas junto com o índice, diz-se que elas são organizadas em índice primário.

# Índices Secundários

## Conceitos

- Um índice secundário é chamado no jargão dos fabricantes de SGBD de **Índice de Blocos** ou **Índice Clusterizado**(!) (*cluster index*).
- Somente índices B-tree e índices Hash podem ser clusterizados nos SGBDs disponíveis.
- Quando um índice é clusterizado, as tuplas são armazenadas nas páginas de dados na mesma ordem das páginas folhas da B-tree ou da sequência Hash.
- Uma limitação é que a disposição física é única, e portanto cada relação pode ter no máximo um índice clusterizado, e nos SGBDs disponíveis, ele tem que ser a chave primária da relação.

# Índices Secundários

## Conceitos

- Note-se que por ser um índice secundário, as folhas da B-tree (ou o diretório do hash) armazenam os RowID das tuplas.
- Assim, o número de chaves armazenadas por página da folha ou do diretório é independente do número de tuplas armazenadas por página de dados.
- O objetivo do índice clusterizado é facilitar o acesso a tuplas que usualmente são acessadas juntas:
- com isso, depois de lida a primeira tupla, a chance da próxima tupla necessária já estar no mesmo *extent* da primeira é muito alta.

# Índices Secundários

## Chuva de ponteiros

- Note-se que, se uma relação estiver organizada em heap, qualquer que seja a ordem de acesso ditada por um índice vai gerar uma **chuva de ponteiros** nas páginas de dados.
- Chuva de ponteiros é o termo usado quando uma lista de ponteiros aponta para os dados em qualquer ordem. A lista de ponteiros, chamada **Rid-List** é uma sequência de RowID tipicamente armazenados em alguns nós folha de B-tree, em um diretório de hash ou numa lista de ponteiros de um arquivo invertido.
- Quando isso ocorre, percorrer sequencialmente a lista de ponteiros faz com que uma página de dados seja acessada múltiplas vezes, com acessos intercalados a outras páginas: quando a mesma página é necessária de novo, ela provavelmente já não estará mais no *buffer*, pois o acesso a outras páginas já a terá desalojado.

# Índices Clusterizados

## Criando um índice clusterizado - Oracle

- A criação de um índice clusterizado é especificado no comando `CREATE TABLE`. A sintaxe em Oracle é a seguinte:

**`CREATE TABLE` – Oracle**

```
CREATE TABLE tbl-name (definição de atributos e restrições)
    [TABLESPACE tablespace]
    [STORAGE ...]
    [PCTFREE n4] [PCTUSED n5]
    [ORGANIZATION {HEAP | INDEX}]
```

- onde a cláusula `ORGANIZATION`, se não for usada, recai para organização em `HEAP`.
- Quando a cláusula `ORGANIZATION INDEX` é colocada, a relação se torna um índice secundário organizado pela `PRIMARY KEY`.

# Índices Clusterizados

## Exemplo

- Vamos assumir que a relação `Matriculas` na USP seja criada *Clusterizada*:

```
CREATE TABLE Matriculas (
    Disciplina CHAR(8) NOT NULL,
    NUSP DECIMAL(10) NOT NULL,
    Nota DECIMAL(4,2),
    PRIMARY KEY (Disciplina, NUSP))
    ORGANIZATION INDEX
```

- Vamos considerar que existam 80.000 alunos e 12.000 disciplinas, cada aluno se matriculando numa média de 8 disciplinas. Então haverá $80.000 * 8 = 640.000$ tuplas na relação `Matrículas`.
- Vamos considerar também que o tamanho da página seja de 2KBytes, assumindo 2.000 bytes para dados.

# Índices Clusterizados

## Exemplo

- Cada tupla da relação ocupa $8 + \frac{10}{2} + \frac{4}{2} = 15$ bytes. Contando o ponteiro no diretório, cada página armazena $\lfloor 2.000/17 \rfloor = \lfloor 117,65 \rfloor = 117$ tuplas.
- A relação `Matriculas` ocupa $\lceil 640.000/117 \rceil = \lceil 5470,09 \rceil = 5470$ páginas de dados.
- Cada chave primária é composta pelos atributos `Disciplina` e `NUSP`, que ocupa $8 + \frac{10}{2} = 13$ bytes. O índice da chave primária terá $\left\lfloor 0,71 \cdot \frac{2.000-2*4}{2+13+6} \right\rfloor = \lfloor 67,35 \rfloor = 67$ chaves por nó folha e $\left\lfloor 0,71 \cdot \frac{2.000-4}{2+13+4} \right\rfloor = \lfloor 74,59 \rfloor = 74$ chaves por nó diretório.
- Então o índice terá $\lceil 640.000/67 \rceil = \lceil 9552,24 \rceil = 9553$ nós folha, $\lceil 9553/74 \rceil = \lceil 129,09 \rceil = 130$ nós no terceiro nível da árvore, portanto $\lceil 130/74 \rceil = 2$ nós no segundo nível e uma folha com 2 entradas.

# Índices Clusterizados

## Exemplo

- Suponha-se que é emitida a seguinte consulta:

```
SELECT NUSP, Nota
    FROM  Matriculas
    WHERE Disciplina='SCC-260'
```

- Como a relação está clusterizada, uma vez que a primeira chave com o valor `Disciplina='SCC-260'` é recuperado, todas as seguintes estarão no mesmo *extent*. Portanto será feita a leitura de 4 páginas de índice (raiz, segundo e terceiro nível e primeira folha), mais o *extent* de dados onde estão as tuplas em sequência.
- Se a relação não estivesse clusterizada, além das 4 páginas, os RowID indicados pela folha da árvore causariam uma chuva de ponteiros pelos *extents* onde as 5470 páginas de dados da relação estariam espalhadas, acessando em média até $\lceil 640.000/12.000 \rceil = \lceil 53,33 \rceil = 54$ páginas de dados.

# Índices Primários

## Conceitos

- Um índice Primário ocorre quando a estrutura de índices é criada junto com os dados, compartilhando o mesmo espaço de memória, e portanto, em um SGBD compartilhando o mesmo segmento de memória.
- A desvantagem disso é que o índice e os dados ficam juntos, portanto não é possível desativar o índice e continuar acessando os dados: o acesso aos dados somente pode ser feito pelo índice.
- Além disso, o nível das folhas da B-tree tem que armazenar não somente a chave de acesso, mas toda a tupla, reduzindo a quantidade de entradas por nó folha e com isso em geral aumentando a altura da árvore (mesmo que o número de entradas por nó diretório não se altere).
- A grande vantagem é que uma vez que se chegou no nó folha, não é necessário seguir um RowID para obter a tupla, ela já está na folha.
- Por isso, a organização do índice primário é sempre feito por *extents* do tamanho que seria o *extent* de dados correspondente.

# Índices Primários

## Cluster de relações - Oracle

- O recurso de *Cluster* de Relações (*Table Clusters*) em Oracle incorpora o conceito de um Índice Primário.
- Um *Cluster* de Relações é uma estrutura que permite armazenar uma ou mais relações em um único segmento de dados.
- As relações precisam ter um conjunto de atributos formando uma chave primária numa relação e chaves estrangeiras nas demais.
- A chave primária é chamada de chave do *cluster*.
- As tuplas de todas as relação que têm o mesmo valor na chave primária e nas chaves estrangeiras são colocadas num mesmo *extent* do segmento do *cluster*.
- Um *Cluster* de Relações sempre usa uma estrutura B-tree ou Hash para organizar o índice primário.
- Outros índices pode ser criados sobre quaisquer das relações.

# Índices Primários

## Cluster de relações - Oracle

- Definição de um *Cluster* de relações em **Oracle**:

**CREATE CLUSTER – Oracle**

```
CREATE CLUSTER Cluster-name (definicoes de atributos [,])
    [TABLESPACE tablespace]
    [STORAGE ...]
    [PCTFREE n4] [PCTUSED n5]
    [SIZE n6]
    [INDEX | [SINGLE TABLE] HASHKEYS n7 [HASH IS expression]]
```

- onde a cláusula `INDEX` assume por default que o *cluster* é baseado em um índice B-tree.

# Índices Primários

## Cluster de relações - Oracle

- Depois de criado o *cluster*, as relações são colocadas nele pelo próprio comando `CREATE TABLE`:

**`CREATE TABLE` – Oracle**

```
CREATE TABLE tbl-name
    (definicoes de atributos e restricoes [,])
    CLUSTER Cluster-name (atr-name [,atr-name ...])
```

- A criação do índice do cluster segue uma sintaxe reduzida:

**`CREATE TABLE` – Oracle**

```
CREATE INDEX cluster-index-name ON CLUSTER Cluster-name
```

# Índices Primários

## Cluster de relações - Oracle

- O comando `CREATE CLUSTER` inclui as cláusulas usuais para definição da organização de memória, semelhantes às do comando `CREATE TABLE`.
- O comando `CREATE INDEX` cria o índice do *cluster*: ele não interfere nos índices das relações, os quais devem ser criados da maneira usual.
- Cada relação a ser colocada no *cluster* também é criada da maneira usual por comandos `CREATE TABLE`, mas eles não podem ter as cláusulas de organização de memória nem a indicação da `TABLESPACE` a ser usada:
- no lugar é colocada a cláusula `CLUSTER`.

# Índices Primários

## Cluster de relações - Oracle

- Os atributos que compõem o *cluster* e as relações devem ser indicados nas respectivas cláusulas `(definições de atributos)` assim:
  - No comando `CREATE CLUSTER`, indicam-se os atributos que são a chave do *cluster* (a chave primária da "relação principal")
  - Na cláusula `(definicoes de atributos e restricoes [,])` do comando `CREATE TABLE` indicam-se todos os atributos e restrições da relação, da maneira usual, incluindo todas as chaves: a chave primária e as chaves estrangeiras.
  - Na cláusula `CLUSTER Cluster-name (atr-name [,atr-name ...])` do comando `CREATE TABLE` indicam-se os atributos que são associados ao *cluster* de relações.
  - O Comando `CREATE INDEX` não recebe a indicação de atributos, pois isso é feito automaticamente usando a definição do *cluster*.

# Índices Primários

## Cluster de relações - Oracle

- A cláusula `SIZE` do comando `CREATE CLUSTER` deve prever uma estimativa do quanto de memória deve ser reservado para todas as tuplas, de todas as relações, que compartilham um mesmo valor para a chave de acesso:
- para cada valor da chave primária da "relação principal", juntam-se todas as tuplas de todas as relações com esse valor, tanto na relação que tem a chave primária, quanto em todas as relações em que essa chave é estrangeira.
- Esse espaço de memória é chamado **bloco do cluster**.
- A partir do valor `n6` dado para o `SIZE`, o Oracle calcula quantos blocos cabem em uma página de dados:
  Blocos por página: $BPP = \left\lceil \frac{\text{Bytes uteis por página}}{\text{SIZE n6}} \right\rceil$.

# Índices Primários

## Cluster de relações - Oracle

- Quando um bloco precisa de mais espaço na pagina, vai usando o espaço até encher a página.
- Quando uma página enche, outra é alocada, de preferência no mesmo *extent*, mas isso não é obrigatório.
- Para isso cria-se uma lista de estouro, equivalente àquela que cuida de colisão num índice Hash, embora o índice possa tanto ser Hash quanto B-tree.
- Portanto é importante estimar corretamente o valor de `SIZE n6`:
  - estimar de menos causa listas de estouros muito grandes;
  - estimar de mais causa desperdício de memória (e aumenta o número de acessos a disco).

# Índices Primários

## Cluster de relações - Oracle

- A cláusula `INDEX | [SINGLE TABLE] HASHKEYS n7 [HASH IS expression]` do comando `CREATE CLUSTER` indica:
  - Se for usado `INDEX`, que é o default, o índice do cluster é uma B-tree.
  - Quando o índice B-tree é usado, ele precisa ser criado explicitamente. Ele pode ser desativado, mas nesse caso as relações do *cluster* ficam inacessíveis.
  - Se for usado `[SINGLE TABLE] HASHKEYS n7 [HASH IS expression]`, o índice do *cluster* é Hash.
  - Índices Hash não são criados explicitamente e não podem ser desativados.

# Índices Primários

## Exemplo de Cluster de relações com B-tree

- Vamos criar um "*cluster* de relações" sobre a relação `Alunos`, juntando os alunos e suas matrículas em um *cluster* só.
- A chave primária é a chave de `Alunos`, um único atributo que é usado para definir o *cluster*:

```
CREATE CLUSTER Cluster_Alunos
    (NUSP CHAR(10))
    TABLESPACE Graduacao
    SIZE 250;
```

- Vamos considerar que existam 80.000 alunos e 12.000 disciplinas, cada aluno se matriculando numa média de 8 disciplinas. Então haverá $80.000 * 8 = 640.000$ tuplas na relação `Matrículas`.
- Vamos considerar também que o tamanho da página seja de 2KBytes, assumindo 2.000 bytes para dados.

# Índices Primários

## Exemplo de Cluster de relações com B-tree

- A relação `Alunos` é criada como:

```
CREATE TABLE Alunos (
    NUSP CHAR(10) PRIMARY KEY,
    Nome VARCHAR(40),
    Idade DECIMAL(3),
    Cidade CHAR(30) )
    CLUSTER Cluster_Alunos(NUSP);
```

- Cada tupla dessa relação ocupa $10 + 40 + \lceil \frac{3}{2} \rceil + 30 = 82$. Com mais 2 bytes do diretório da página, temos 84 bytes por tupla.

# Índices Primários

## Exemplo de Cluster de relações com B-tree

- A relação `Matrículas` é criada como:

```
CREATE TABLE Matriculas (
    Disciplina CHAR(7) NOT NULL,
    NUSP DECIMAL(10) NOT NULL,
    Nota DECIMAL(4,2),
    PRIMARY KEY (Disciplina, NUSP) )
    CLUSTER Cluster_Alunos(NUSP);
```

- Cada tupla dessa relação ocupa $7 + \frac{10}{2} + \frac{4}{2} = 14$. Com mais 2 bytes do diretório da página, existem 16 bytes por tupla.

# Índices Primários

## Exemplo de Cluster de relações com B-tree

- Considerando 640.000 matrículas que os 80.000 alunos fazem, temos uma média de $640.000/80.000 \approx 8$ alunos matriculados em cada disciplina.
- Então, cada bloco do *cluster* irá ter em média 1 tupla da relação `Alunos` e 8 tuplas da relação `Matriculas`, o que resulta em $1 * 84 + 8 * 16 = 212$ bytes em média.
- Como essa é a média, majora-se um pouco a estimativa, para atender a maioria das situações. Assim, usamos foi usado o valor de 250 bytes para `SIZE`.
- O Oracle irá calcular $BPP = \left\lceil \frac{2.000}{2500} \right\rceil = 8$ blocos do *cluster* por página.
- Portanto, a relação alimentada irá utilizar $80.000/8 = 10.000$ páginas para o *cluster*. Por ser um índice primário B-tree, essas páginas são criadas sob demanda.
- Se houver previsão de que ocorrerão muitos acessos sequenciais a diversos alunos na mesma consulta, então os *extents* deverão ser grandes, caso contrário é possível usar o valor default – no exemplo está sendo usado `INITIAL=2K` e `NEXT=2K`.

# Índices Primários

## Cluster de relações com HASH –Oracle

- A cláusula `[SINGLE TABLE] HASHKEYS n7 [HASH IS expression]` indica que o índice do *cluster* é Hash.
- Nesse caso, é criado o que se chama um índice primário Hash (*Hash Primary Index*)
- O valor `HASHKEYS n7` indica quantas chaves de acesso estão previstas para o índice, ou seja, o qual número previsto de blocos no cluster.
- Por ser um índice Hash, essas páginas são pré-alocadas quando o *cluster* é criado. Páginas de *overflow* podem ser adicionadas depois, sob demanda, se houver colisões ou mais chaves repetidas do que o previsto.
- Dessa maneira, o índice inteiro é criado com tamanho de *extent* igual a `SIZE n6*HASHKEYS n7` (limitado obviamente ao tamanho do cilindro no disco onde ele estiver armazenado).

# Índices Primários

## Cluster de relações com HASH –Oracle

- A indicação de `SINGLE TABLE` permite que o *hash* seja otimizado para ter apenas uma relação.
- Essa é a maneira de se criar índices *hash* em Oracle.
- A indicação de `<HASH IS expression>` permite que o analista defina a função de Hash a ser usada.
- Se não indicada, a função de *hash* calcula o *checksum* da chave de acesso, e aplica o módulo da divisão desse valor pelo menor número primo maior do que `SIZE n7`.

# Índices Primários

## Exemplo de Cluster de relações com HASH

- Vamos assumir que no exemplo anterior deve-se criar um *cluster* de relações usando Hash ao invés de B-tree. O que muda é só o comando de criação do *cluster*:

```
CREATE CLUSTER Cluster_Alunos
    (NUSP CHAR(10))
    TABLESPACE Graduacao
    SIZE 250
    HASHKEYS 80K;
```

- Como antes, a relação irá utilizar 10.000 páginas para o *hash cluster*, em um *extent* criado inteiro quando a primeira tupla for inserida em qualquer das relações.
- Note-se que não pode ser emitido um comando para criar o índice do *cluster*.

# Declaração de Índices

- Em SQL, índices podem ser declarados:
  - Implicitamente, quando se indicam as restrições de integridade usando as palavras-chave `PRIMARY KEY` ou `UNIQUE` na definição das tabelas;
  - Explicitamente, usando o comando `CREATE INDEX`.
- O efeito de ambas as declarações é a mesma em termos de criar estruturas de indexação.
- Em SQL, uma "**tabela**" é composta:
  - Pela "tabela-base",
  - e por todos os seus índices associados.
- Cada índice cria uma estrutura de dados alocada (a princípio – em uma estrutura `heap` ou de índice secundário) em um segmento específico (segmento de índice), sendo a tabela-base alocada em seu próprio segmento.

# Uso de índices

- Quando uma consulta requisita dados, eles podem ser obtidos na tabela-base ou nos índices da tabela.
- Portanto, um índice é usado para:
  - **Busca** (filtrar – *screening*) o acesso aos dados da tabela-base, apontando diretamente para as tuplas que podem conter a resposta; ou
  - **Prover** (identificar – *matching*) os dados necessários.

# Usando índices para filtrar os dados

- Filtrar os dados é a forma mais disseminada para usar índices. Por exemplo:
- Suponha que exista um índice $B^+$-tree sobre o atributo `Nome` da tabela `Alunos`:

```
CREATE INDEX IDXAlunos_Nome ON Alunos(Nome);
```

- Suponha agora que a seguinte consulta é recebida:

```
SELECT Nome, Idade FROM Alunos
    WHERE Nome ='Jose';
```

- Nesse caso:
  - o índice é acessado,
  - identificam-se a(s) tupla(s) que têm `WHERE Nome ='Jose'`,
  - accessa-se essas tuplas na tabela-base para obter a `Idade` de cada aluno.

# Usando índices para filtrar os dados

- O uso do índice para filtrar dados requer que o índice tenha, dentre os seus primeiro atributos indexados, os atributos que apareçam em ao menos um predicado da consulta.
- Nesse exemplo, o índice pode ser usado porque seu (único) atributo indexado é `Nome`, que aparece no predicado `WHERE Nome ='Jose'`.

# Usando índices para prover os dados

- Prover os dados é uma maneira de acessar apenas o índice, sem acessar os dados. Por exemplo:
- Suponha que o seguinte índice foi criado na tabela `Alunos`:

```
CREATE INDEX IDXAlunos_NomeIdade
    ON Alunos(Nome, Idade);
```

- Suponha agora que a seguinte consulta é recebida:

```
SELECT Nome, Idade FROM Alunos
    WHERE Nome ='Jose';
```

- Nesse caso:
  - o índice é acessado,
  - e já se obtém os atributos `Nome` e `Idade` de todas as tuplas necessárias direto no índice, sem necessidade que a tabela-base seja acessada.

# Usando índices para prover os dados

- Note-se que esse uso do índice não requer que a consulta inclua os atributos do índice em seus predicados.
- Na realidade, a consulta:

```
SELECT Nome, Idade FROM Alunos;
```

  pode usar o índice `IDXAlunos_NomeIdade`, mesmo que nem `Nome` nem `Idade` apareçam nos predicados da cláusula `WHERE`.
- Veja que embora nesse caso o índice inteiro precise ser acessado, e possivelmente acessa-lo é mais rápido que acessar a tabela-base inteira.

# Indexação - Guias de Cálculo

## Calculando o tamanho de um conjunto de atributos

- Passo 1: Calcular o tamanho em bytes de cada atributo:

| Tipo | Bytes | Tipo | Bytes |
|---|---|---|---|
| SMALLINT | 2 | CHAR($n$) | $n$ |
| INTEGER | 4 | VARCHAR($n$) | $n$ |
| BIGINT | 8 | DATE | 3 |
| REAL | 4 | TIME | 5 |
| DOUBLE PRECISION | 8 | DATETIME | 8 |
| DECIMAL($n, m$) | $n/2$ | xLOB | 6 (RowID) |
| NUMERIC($n, m$) | $n/2$ | | |

- Atributos VARCHAR($n$) devem ter uma estimativa independente do número médio de bytes ocupados.
- Passo 2: Calcula-se a soma do espaço de cada atributo: $s = \sum atr_i$ .
- Pode ser o tamanho da tupla $tam = s$ toda,
  ou o tamanho da chave $Tchave = s$ .

# Indexação - Guias de Cálculo

## Calculando quantas tuplas cabem por página de dados

- A estrutura típica de um *extent* de dados é a seguinte:


Cada tupla ocupa seu próprio tamanho mais os dois bytes do ponteiro para ela no diretório.

- Devem ser conhecidos
  - o tamanho da tupla: *tam*
  - o tamanho da página: *pg*
  - o tamanho do header: *header*
  - o `FILLFACTOR` da Relação: $FF \quad (0 < FF \leq 1)$
- A área de dados ou é dada, ou é calculada como $pgDados = pg - header$.
- O número de tuplas por página é: $NTuplasPP = \left\lfloor FF \cdot \frac{pgDados}{2+tam} \right\rfloor$.

# Indexação - Guias de Cálculo

## Calculando quantas Páginas de Dados são necessárias para armazenar uma relação

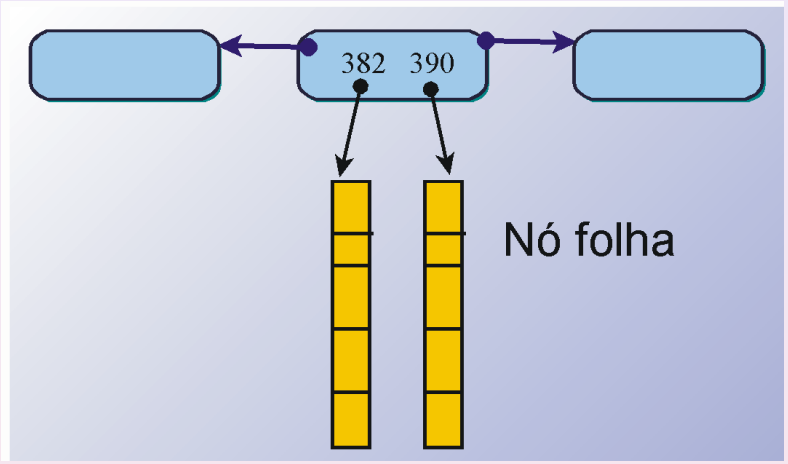
- Como no cálculo do número de tuplas por página já está contado o espaço para o diretório, o cálculo é direto.
  - Devem ser conhecidos
    - o número de tuplas na relação: *NTuplas*
    - o número de tuplas por página: *NTuplasPP*
  - O número de Páginas de Dados da relação é:

$$NPagRel = \left\lceil \frac{NTuplas}{NTuplasPP} \right\rceil .$$

# Indexação - Guias de Cálculo

## Calculando quantas chaves cabem por nó folha em uma B-tree *UNIQUE*

- A estrutura típica de um nó folha é a seguinte:



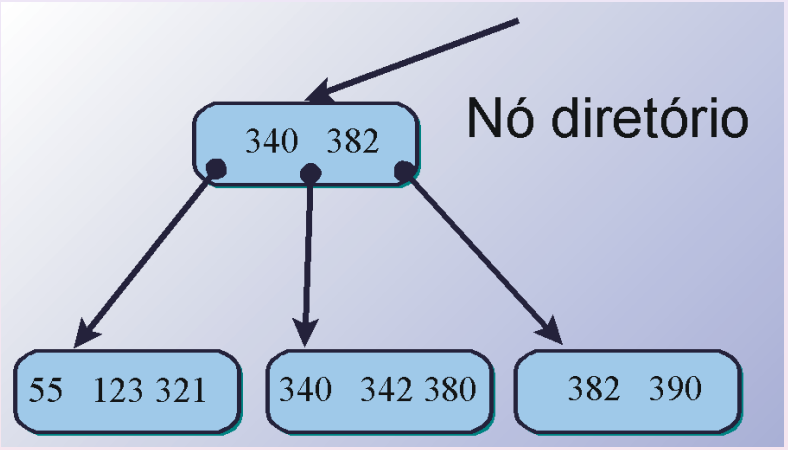

☞ Cada nó folha tem $k$ chaves, $k$ ponteiros RowId e 2 ponteiros para irmãos.
☞ Cada chave ocupa seu tamanho mais dois bytes do seu ponteiro no diretório da página.
☞ Cada ponteiro para irmão é interno ao segmento de índice, então ocupa 4 bytes.
☞ Cada RowId aponta para outro segmento, então ocupa 4 ou 6 bytes. O default é 6.

- Devem ser conhecidos
  - o tamanho da área de dados: *pgDados*
  - o tamanho da chave: *Tchave*
  - o `FILLFACTOR` da Relação: $FF \quad (0 < FF \leq 1)$
- O número de chaves é: $$NChavFolha = \left\lfloor FF \cdot \frac{0{,}71 \cdot (pgDados - 2*4)}{2 + Tchave + 6} \right\rfloor .$$

# Indexação - Guias de Cálculo

## Calculando quantas chaves cabem por nó diretório em uma B-tree *UNIQUE*

- A estrutura típica de um nó diretório é a seguinte:




☞ Cada nó diretório tem $k$ chaves e $k+1$ ponteiros.
☞ Cada chave ocupa seu tamanho mais os dois bytes do ponteiro para ela no diretório da página.

☞ Cada ponteiro é interno ao segmento de índice, então ocupa 4 bytes.

- Devem ser conhecidos
  - o tamanho da área de dados: *pgDados*
  - o tamanho da chave: *Tchave*
  - o `FILLFACTOR` da Relação: $FF \quad (0 < FF \leq 1)$
- O número de chaves é: $$NChavDir = \left\lfloor FF \cdot \frac{0{,}71 \cdot (pgDados - 4)}{2 + Tchave + 4} \right\rfloor .$$

# Indexação - Guias de Cálculo

## Calculando quantas Páginas são necessárias para armazenar um índice B-tree UNIQUE

- Uma B-tree UNIQUE tem o numero de folhas suficiente para armazenar uma chave de busca por tupla da relação
- e o nível de diretórios necessário para ter ponteiros para todas as folhas.
  - Devem ser conhecidos
    - o número de tuplas na relação: *NTuplas*
    - o número de chaves por nó diretório: *NChavDir*
    - o número de chaves por nó folha: *NChavFolha*
  - Passo 1: calcula-se o número de páginas para nós folha:

    $$NPagFolha = \left\lceil \frac{NTuplas}{NChavFolha} \right\rceil .$$

  - Passo 2: calcula-se o número de páginas para o último nível de nós diretório:

    $$NPagFolha = \left\lceil \frac{NTuplas}{NChavDir} \right\rceil .$$

  - Repete-se o segundo passo calculando-se quantos nós diretórios são necessários para apontar para o próximo nível da árvore, recém-calculado, até que o número calculado caiba em um nó só, que é a raiz.
  - A altura da árvore é o

    $$H = \text{número de vezes que o segundo passo foi executado} + 1 \text{ (folhas)} + 1 \text{ (Raiz)} .$$

  - O número de páginas do índice B-tree *UNIQUE* é:

    $$NPagFolha + \sum NpagDir + 1(\text{raiz}) .$$

# Indexação - Guias de Cálculo

## Calculando quantas Páginas são necessárias para armazenar um índice B-tree não-UNIQUE

- Além dos dados necessários para os cálculos em uma B-tree UNIQUE, deve-se ter uma estimativa de quantas repetições existem em média para cada chave:
  - Devem ser conhecidos
    o número médio de tuplas repetidas por chave: *Nrep*
  - O número de chaves numa folha é:
    $$kChavFolha = \left\lfloor FF \cdot \frac{0{,}71 \cdot (pgDados - 2*4)}{2 + Tchave + 6 \cdot Nrep} \right\rfloor$$
  - O número de chaves num nó diretório é:
    $$kChavDir = \left\lfloor FF \cdot \frac{0{,}71 \cdot (pgDados - 4)}{2 + Tchave + 4} \right\rfloor$$
  - O número de páginas para nós folha é: $$NPagFolha = \left\lceil \frac{NTuplas}{Nrep \cdot NChavFolha} \right\rceil$$
  - A partir daí, os passos 2 e 3 do slide anterior se repetem sem alteração.

# Indexação - Guias de Cálculo

## Calculando o número de páginas ocupado por um BitMap

- Um índice Bitmap tem a Chave de acesso e um bit por tupla da relação.

  O diretório do Bitmap é uma B-tree, portanto seu espaço é calculado igual ao do diretório da B-tree.

  Cada página armazena o *bit map* mais um ponteiro de continuação.

  - Devem ser conhecidos
    - o tamanho da área de dados: $pgDados$
    - o tamanho da chave: $Tchave$
    - o número de tuplas na relação: $NTuplas$
  - O número de páginas ocupado é: $k = \left\lceil \frac{Tchave + \frac{NTuplas}{8}}{pgDados - 4} \right\rceil$.

# Arquitetura de SGBD Relacionais
## — Indexação —

Caetano Traina Jr.

Grupo de Bases de Dados e Imagens
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
caetano@icmc.usp.br

13 de junho de 2013
São Carlos, SP - Brasil

FIM