

# INDEXAÇÃO E HASHING

## O QUE É UM ÍNDICE?

- Índices são estruturas de dados auxiliares cujo único propósito é tornar mais rápido o acesso a registros baseado em certos campos, chamados campos de indexação ou chaves de busca
- Um índice permite localizar um registro sem ter que examinar mais que uma pequena fração dos registros possíveis
- Tipos de Índices
  - Ordenados
    - Baseiam-se na ordenação de valores
  - Hash
    - baseiam-se na distribuição uniforme dos valores determinados por uma função (*função de hash*)
- Qual o melhor?
  - Cada uma é mais adequada a determinada aplicação

## CONTEXTUALIZAÇÃO

- Considerando que o arquivo já esteja em alguma organização primária (desordenada/heap, ordenada, hash)
- Índice = arquivo auxiliar busca de um registro mais eficiente
  - Arquivo de índice geralmente ocupa MENOS espaço que blocos no disco. Por quê?

*O tamanho da entrada no índice MUITO MENOR que tamanho do registro*
- Exemplo: um arquivo com entradas  
*<valor do campo, ponteiro ao registro>*: ordenado pelo valor
- Índice geralmente especificado em um campo do arquivo (embora possível especificar em vários campos)

## FATORES DE AVALIAÇÃO

- **Tipos de Acesso**
  - Localização de Registros
- **Tempo de Acesso**
  - Tempo gasto para localizar determinado item de dados ou um conjunto de itens.
- **Tempo de Inserção**
  - Tempo gasto para inserir um novo item de dado

## FATORES DE AVALIAÇÃO

### ➤ Tempo de Exclusão

- Além da exclusão em si, deve-se considerar o tempo de localização do registro a ser excluído assim como a atualização da estrutura de índice

### ➤ Espaço Adicional

- Espaço ocupado por uma estrutura de índices. O espaço adicional deve ser moderado para se conseguir desempenho.

## AVALIAÇÃO DAS TÉCNICAS

*Tipos de Acesso*

Acessos Eficientes

*Tempo de Acesso*

Depende da Técnica

*Tempo de Inserção*

Localizar e Atualizar

*Tempo de Exclusão*

Localizar e Atualizar

*Espaço Adicional*

Espaço x Desempenho

## ÍNDICES ORDENADOS

### ➤ Chave de Busca – O que é?

- Difere de chave primária, chave candidata,...
- Um índice ordenado armazena os valores das chaves de buscas em ordem classificada e associada aos registros que as utiliza.
- Atributo ou conjunto deles utilizados para pesquisar registros em um arquivo
- Acesso aleatório rápido aos registros
- Índice armazena a chave de busca
- Chave de busca é associada aos registros

## ÍNDICES ORDENADOS

- Um arquivo pode conter vários índices em diferentes chaves de busca

### ➤ Índice Primário

- Chave de busca especifica a ordem sequencial do arquivo
- Geralmente é a chave primária, mas nem sempre

### ➤ Índice de Agrupamento ou clustering

- Semelhante ao índice primário quando a chave de busca não for chave primária

### ➤ Índice Secundário

- A chave de busca especifica uma ordem diferente da sequencial

## ÍNDICES PRIMÁRIOS

- Este índice consiste em um arquivo ordenado cujos registros são de tamanho fixo, contendo dois campos:
  - 1º- mesmo tipo do campo chave no arquivo de dados
  - 2º- ponteiro para o bloco do disco, ou seja, a localização física do registro do dado

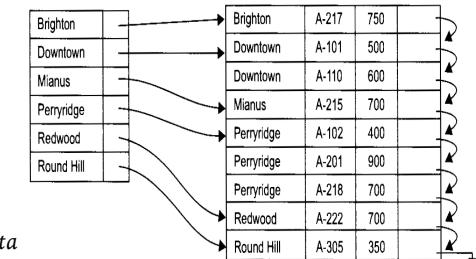
*Campo chave                      Ponteiro*

- Arquivos com índices primários são chamados
  - Arquivos indexados sequencialmente
- Como o índice primário mantém ordenado os registros no arquivo, os processos de inserção e remoção são um grande “problema” para este tipo de estrutura.

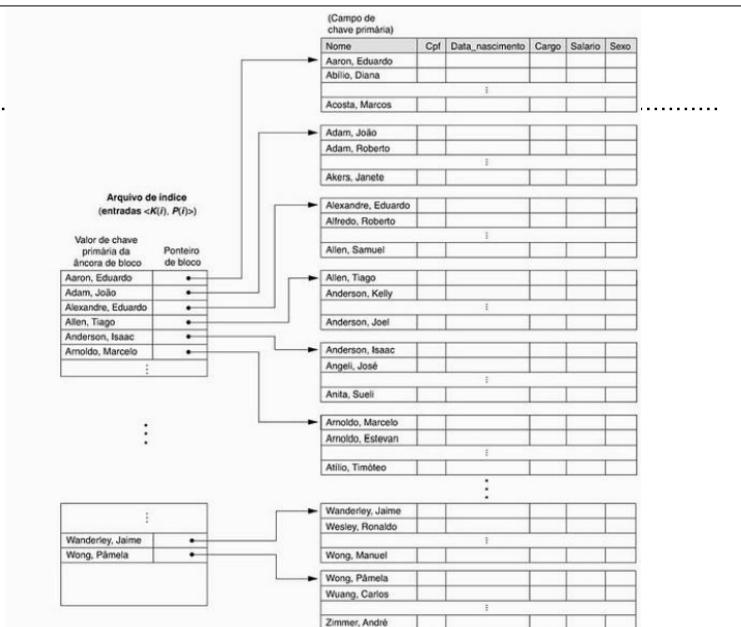
## ÍNDICES PRIMÁRIOS

Brighton	A-217	750	↑
Downtown	A-101	500	↑
Downtown	A-110	600	↑
Mianus	A-215	700	↑
Perryridge	A-102	400	↑
Perryridge	A-201	900	↑
Perryridge	A-218	700	↑
Redwood	A-222	700	↑
Round Hill	A-305	350	↑

*Arquivo sequencial para registros de conta*



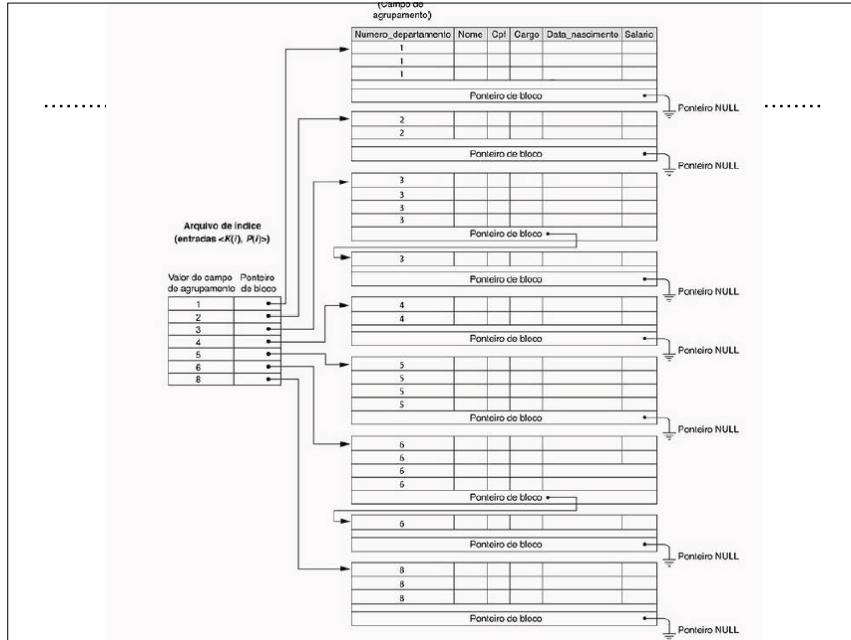
*Arquivo de índices para registros de conta*



## ÍNDICES DE AGRUPAMENTO

Brighton	↑	A-217	Brighton	750	↑
Downtown	↑	A-101	Downtown	500	↑
Mianus	↑	A-110	Downtown	600	↑
Perryridge	↑	A-215	Mianus	700	↑
Redwood	↑	A-102	Perryridge	400	↑
Round Hill	↑	A-201	Perryridge	900	↑
		A-218	Perryridge	700	↑
		A-222	Redwood	700	↑
		A-305	Round Hill	350	↑

*Índice aparece para cada valor da chave de busca no arquivo. O restante dos registros do mesmo valor da chave de busca são armazenados sequencialmente após o primeiro registro*



## ÍNDICES ORDENADOS

### ► Denso

- Sequência de blocos contendo apenas as chaves dos registros e os ponteiros para os próprios registros

- Índice denso = (chave, ponteiro para registros)

### ► Esparsos

- Usa menos espaço de armazenamento que o índice denso ao custo de um tempo maior para localizar o registro

- Índice esparsos = (chave, ponteiro para blocos de dados)

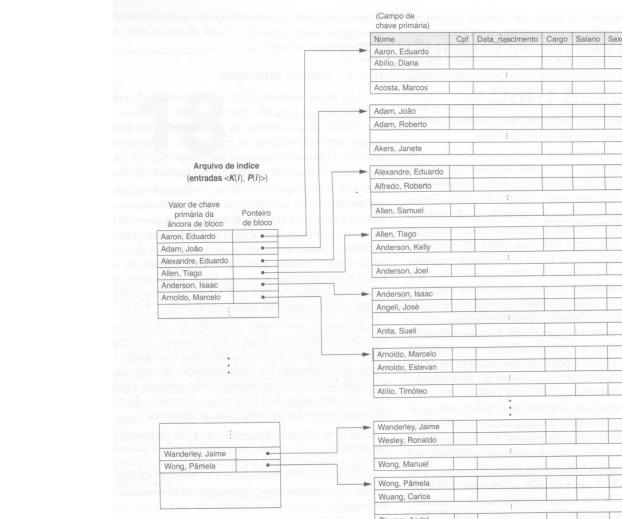
*Aponta para o primeiro registro de cada bloco*

## ÍNDICES DENSOS - CHAVE - SILBERSCHATZ

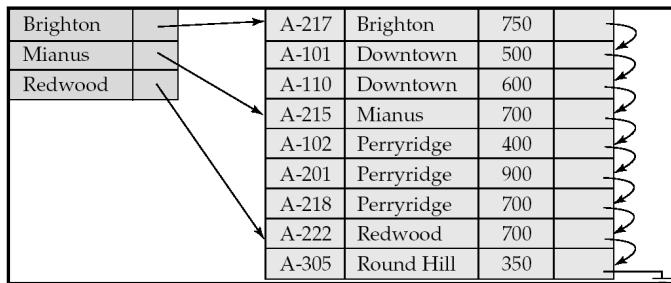
This diagram shows a dense index structure where each index entry points directly to its corresponding data record. The primary key values (10101, 12121, etc.) are circled in black. Arrows point from each circled value to its respective row in the data table below. The data table consists of columns: Numero\_departamento, Nome, Cpf, Cargo, Data\_nascimento, and Salario.

10101	10101	Srinivasan	Comp. Sci.	65000	
12121	12121	Wu	Finance	90000	
15151	15151	Mozart	Music	40000	
22222	22222	Einstein	Physics	95000	
32343	32343	El Said	History	60000	
33456	33456	Gold	Physics	87000	
45565	45565	Katz	Comp. Sci.	75000	
58583	58583	Califieri	History	62000	
76543	76543	Singh	Finance	80000	
76766	76766	Crick	Biology	72000	
83821	83821	Brandt	Comp. Sci.	92000	
98345	98345	Kim	Elec. Eng.	80000	

## ÍNDICES ESPARSOS - CHAVE-NAVATHE



## ÍNDICES ESPAROSOS



**Contém índice de registro somente para alguns valores da chave de busca e o restante da busca é feita sequencialmente.**

## ÍNDICES DE NÍVEL ÚNICO

- Índices esparsos muito grandes
- Ocupam muitos blocos
- Processamento ineficiente
- Armazenados sequencialmente
- Arquivo maior que a memória principal
  - Busca exige várias leituras de bloco de disco
- Solução
  - Índices de Níveis Múltiplos

## ÍNDICES DENSOS X ESPAROSOS

### ➤ Densos

- São mais rápidos para localizar um registro

### ➤ Esparsos

- Necessita menos espaço e menos sobrecarga de manutenção de ponteiros para operações de inclusão e exclusão

### ➤ Qual melhor?

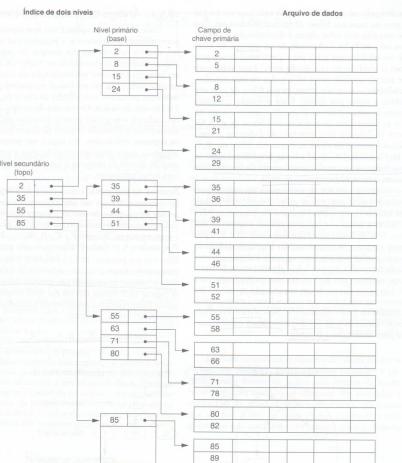
- Depende da aplicação
- Decisão tomada pelo DBA

## ÍNDICES DE NÍVEIS MÚLTIPLOS

- Índices com dois ou mais níveis
  - Índice esparsos sobre o índice primário
  - Exemplo fora de banco de dados
    - Dicionário

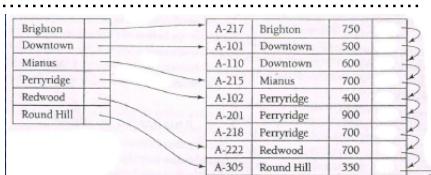
## ÍNDICE ESPARSO EM DOIS NÍVEIS

- Procedimento de busca
  - Busca binária sobre o índice externo
  - Busca o registro com maior valor de chave  $\leq$  valor desejado
- Varredura do bloco de índice interno
- Encontra o registro
  - Ponteiro aponta para o bloco do arquivo que contém o registro
- Exemplo
  - Encontrar o registro para a chave 36



## INSERÇÃO EM ÍNDICES DENSOS

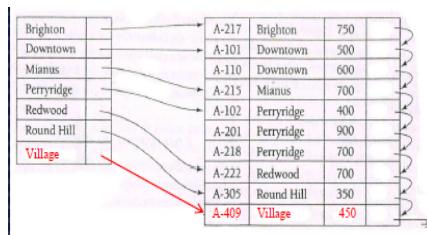
- Se o valor da chave de busca não aparece no índice, o sistema insere um registro de índice com o valor de chave de busca no índice, na posição apropriada.



Exemplo:

Inserir no índice a agência de nome Village.

Considere ainda haver espaço no bloco.



## ATUALIZAÇÃO DE ÍNDICES

- Cada índice precisa ser atualizado sempre que um registro é inserido ou excluído do arquivo

### Inserção

{índices densos}

{índices esparsos}

### Exclusão

{índices densos}

{índices esparsos}

## Caso contrário:

- o registro de índice armazena um ponteiro somente no primeiro registro com o valor da chave de busca. O sistema, então, coloca o registro sendo inserido após os outros registros com os mesmos valores de chave de busca.

## INSERÇÃO EM ÍNDICES ESPARSSOS

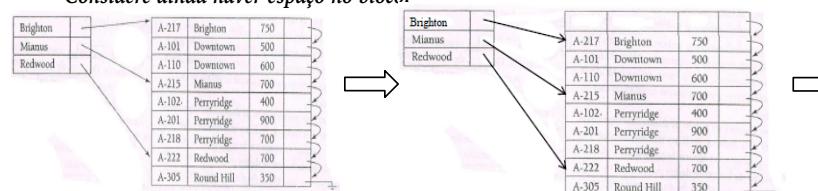
- Se o novo registro tiver o menor valor de chave de busca em seu bloco, o sistema atualiza a entrada de índice apontando para o bloco.
- Se o sistema cria um novo bloco, ele insere no índice o primeiro valor de chave de busca (na ordem de chave de busca) que aparece no novo bloco.

- Caso contrário, o sistema não faz qualquer mudança no índice.

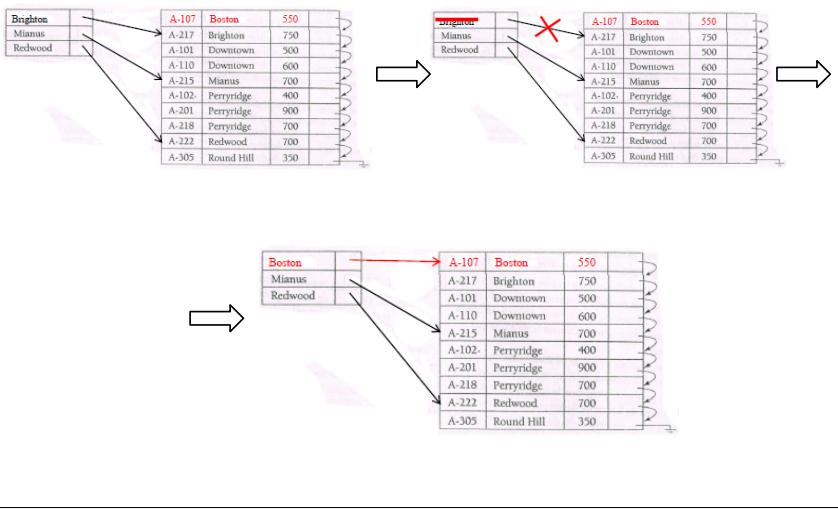
Exemplo:

Inserir no índice a agência de nome Boston.

Considere ainda haver espaço no bloco.



## INSERÇÃO EM ÍNDICES ESPAROSOS

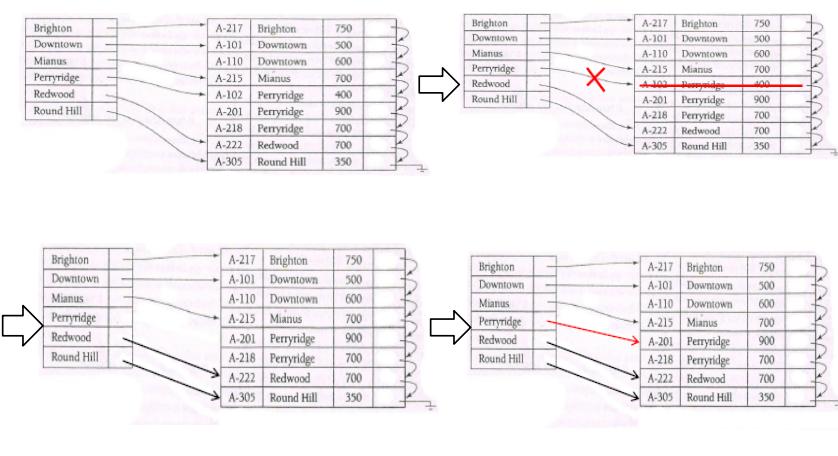


## EXCLUSÃO EM ÍNDICES DENSOS

- Se o registro excluído foi o único registro com seu valor específico de chave de busca, então o sistema retira do índice o registro de índice correspondente.
- Senão:
  - O sistema atualiza o registro de índice para que aponte para o próximo registro.

## EXCLUSÃO EM ÍNDICES DENSOS

Exemplo: Registro referente a conta A-102 foi excluído

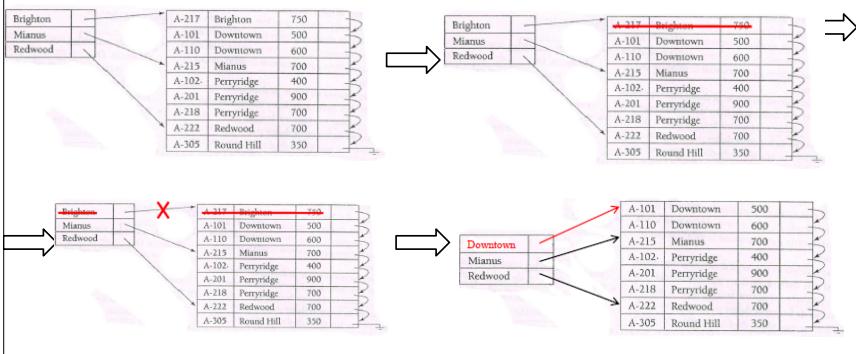


## EXCLUSÃO EM ÍNDICES ESPAROSOS

- Se o índice não tiver um registro de índice com o valor de chave de busca do registro excluído, nada precisa ser feito com o índice.
- Senão:
  - Se o registro excluído foi o único com sua chave de busca, o sistema substitui o registro de índice correspondente por um registro de índice para o próximo valor da chave de busca. Se o próximo valor de chave de busca já tiver uma entrada de índice, a entrada é excluída, em vez de ser substituída.
  - Caso contrário, se o registro de índice para o valor da chave de busca apontar para o registro sendo excluído, o sistema atualiza o registro de índice para que aponte para o próximo registro com o mesmo valor da chave de busca.

## EXCLUSÃO EM ÍNDICES ESPARSO

Exemplo: Registro referente a conta A-217 foi excluído



## COMPARANDO

- Denso: Pode informar se o registro existe sem precisar acessar o arquivo
- Esparsos: Espaço de índice menor por registro, pode manter porção maior do índice na memória

## EXERCÍCIO

- Suponha que uma chave inteira utilize 2 bytes, um ponteiro utilize 4 bytes e um registro de uma certa tabela utilize 20 bytes.
- Considere blocos de 60 bytes. Considere índices contendo pares chave-ponteiro.
- Quantos blocos serão necessários para armazenar dados e índice se a tabela possuir 90 registros, utilizando-se:
  - um índice denso;
  - um índice esparsos.

## ÍNDICES SECUNDÁRIOS

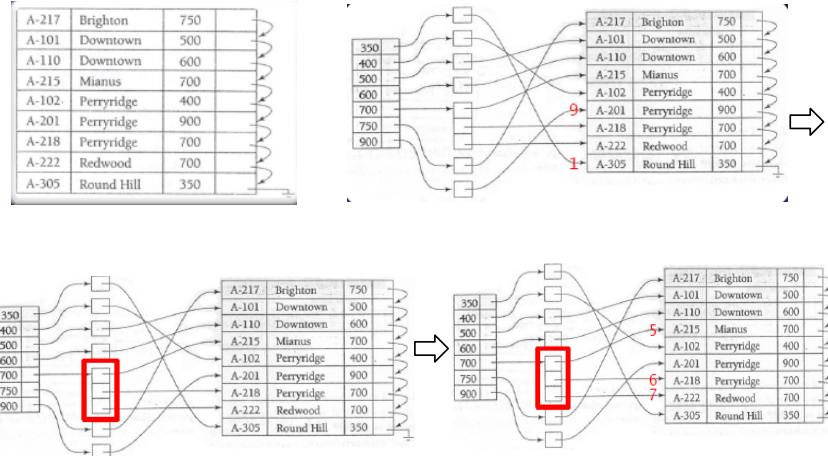
- Chaves de procura não é aquela da ordem sequencial do arquivo
  - Podem não ser chaves candidatas
- Melhoraram o desempenho de consultas que usam chaves diferentes das chaves primárias
- Organização
  - há uma entrada no índice para cada valor de chave que ocorre em um registro de dados
  - a entrada aponta para **todos os registros que contém aquele valor de chave x**
- Os índices secundários precisam ser
  - densos, como uma entrada de índice para cada valor de chave de busca,
  - e um ponteiro para cada registro no arquivo

## ÍNDICES SECUNDÁRIOS

- Os índices secundários em chaves não candidatas não são suficientes porque
  - Ponteiros apontam apenas para o primeiro registro de cada valor de chave de busca
  - Registros de índice secundário não são sequenciais, assim não tem como dar continuidade à localização do registro
  - Necessário pesquisar o arquivo inteiro
  - Solução
    - Nível extra de indireção
    - Ponteiros não apontam diretamente para o arquivo
    - Ponteiros apontam para *buckets(baldes)* que contém ponteiros para o arquivo

## ÍNDICES SECUNDÁRIOS

Índice secundário com chave não-candidata saldo

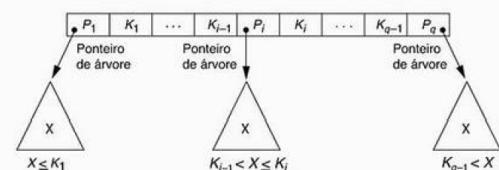


## ARQUIVOS DE ÍNDICES DE ÁRVORE B<sup>+</sup>

- Organização de arquivo sequencial indexado:
  - Arquivo cresce e desempenho diminui
  - Remediado pela reorganização do arquivo
- **Índice de árvore B<sup>+</sup>**
  - Mantém eficiência apesar da inserção e exclusão
  - Forma de árvore balanceada
  - Caminho da raiz até uma folha qualquer é do mesmo tamanho
- **Implicações:**
  - Sobrecarga de desempenho na inserção e exclusão
  - Aumenta o espaço adicional para o uso de ponteiros

## ÁRVORES B<sup>+</sup>

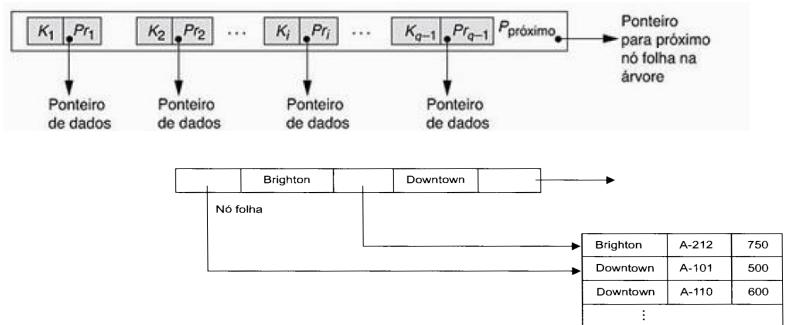
1. Ponteiro de cada chave
  - 1.1. Esquerda: sub-árvore de valores menores ou iguais a chave
  - 1.2. Direita: sub-árvore de valores maiores a chave
2. **Três tipos de nós para uma árvore com p ponteiros**
  - 2.1. **Raiz** - tem, pelo menos, dois ponteiros
  - 2.2. **Internos**
    - 2.2.1. pelo menos,  $\lceil (p/2) \rceil$  ponteiros
    - 2.2.2.  $K_1 < K_2 < \dots < K_{p-1}$



## ÁRVORE B+

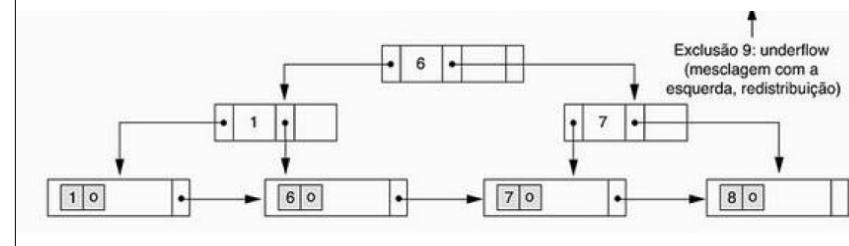
3. **Folha** - apontam para arquivos de dados. Tem pelo menos  $\lceil p_{folha}/2 \rceil$  valores

1. último ponteiro é para o próximo bloco ( $p_{folha}=p-1$ )
2. os demais é para os registros de dados



## CARACTERÍSTICAS DE ÁRVORE B+

- Inserção num nó não cheio: muito eficiente
- Inserção num nó cheio: divisão em dois nós
  - Essa divisão pode ser propagada para outros níveis
- Remoção eficiente se não deixa nó menor da metade cheio
  - Se for o caso, faz-se a fusão com nós vizinhos



## CONSULTAS EM ÁRVORES B+

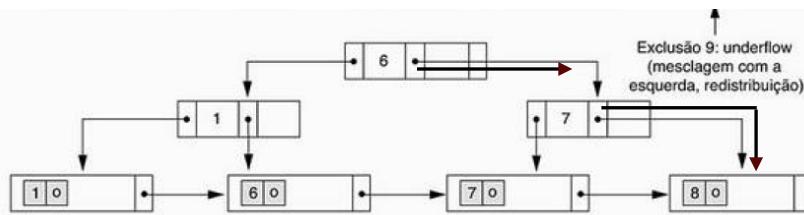
- Árvores caracterizadas por serem baixas e largas
- Atravessa um caminho na árvore desde a raiz até algum nó de folha
- Se houver k chaves de busca o caminho  $\leq \lceil \log_{n/2}(k) \rceil$
- Exemplo:
  - n=100
  - Arquivo com 1.000.000 valores de chave de busca (K)
  - Pesquisa acessa apenas 4 nós em árvore B+

## CONSULTAS EM ÁRVORES B+

- Pesquisar chave de busca V
  - Examina-se o nó raiz procurando o menor valor da chave de busca que seja maior do que V
  - Assuma que esse valor seja  $K_i$ .
  - Segue-se o ponteiro  $P_i$  até outro nó
  - Pode-se alcançar um nó-folha para o qual o ponteiro leva ao registro ou ao bucket desejado

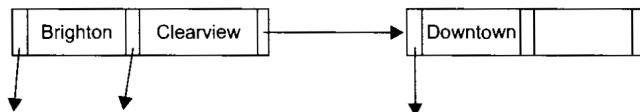
## CONSULTAS EM ÁRVORES B+

- V= 8



## INSERÇÃO COM DIVISÃO

- Coloca-se  $[n/2]$  no nó existente e o restante das chaves no novo nó, onde n é os n-1 valores existentes + o valor a ser inserido
- Regra Geral: determinar o nó folha a ser inserido o novo valor.
  - Se isso resultar numa divisão, então insere-se o primeiro valor do novo nó no pai do nó que foi dividido.
- Caso necessário, procede-se recursivamente até a raiz.



Divisão de um nó folha na inserção "Clearview"

## ATUALIZAÇÕES EM ÁRVORES B<sup>+</sup>

- Inserção ou exclusão:
  - Semelhante a técnica da pesquisa
  - Pode necessitar divisão ou união de nós para garantir balanceamento
  - Na inserção com divisão:
    - Coloca-se  $[n/2]$  no nó existente e o restante das chaves no novo nó, onde n é os n-1 valores existentes + o valor a ser inserido
    - Regra Geral: determinar o nó folha a ser inserido o novo valor.
      - Se isso resultar numa divisão, então insere-se o primeiro valor do novo nó no pai do nó que foi dividido.
    - Caso necessário, procede-se recursivamente até a raiz.

## INSERÇÃO ÁRVORE B<sup>+</sup>

Inserir 8, 5, 1, 7, 3, 12, 9, 6

$p=3$



$p_{folha} = 2$

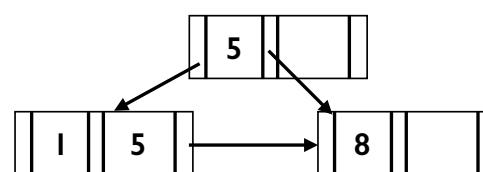
$nós\ internos = \lceil p/2 \rceil$  ponteiros

$nós\ folhas = \lceil p_{folha}/2 \rceil$  valores

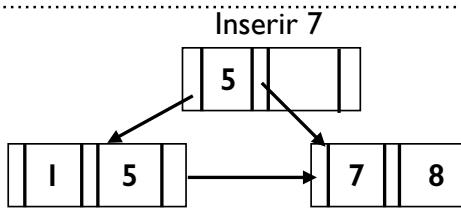
Inserir 5



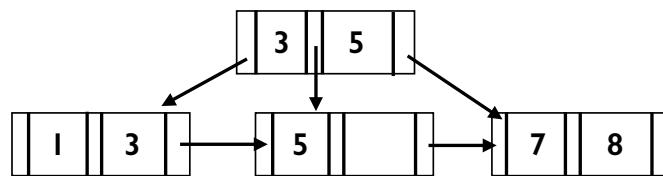
Inserir 1 - não cabe no nó. Acontece a divisão



## INSERÇÃO EM ÁRVORE B+

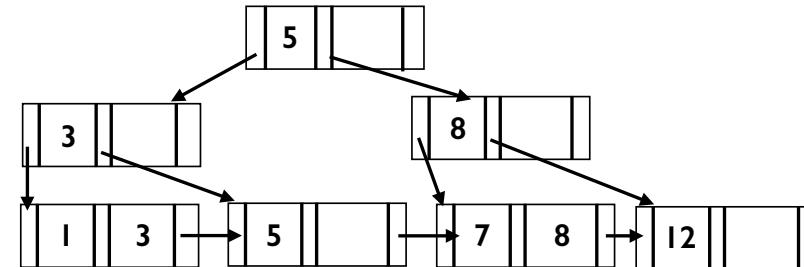


Inserir 3 - não cabe no nó adequado. Nova divisão

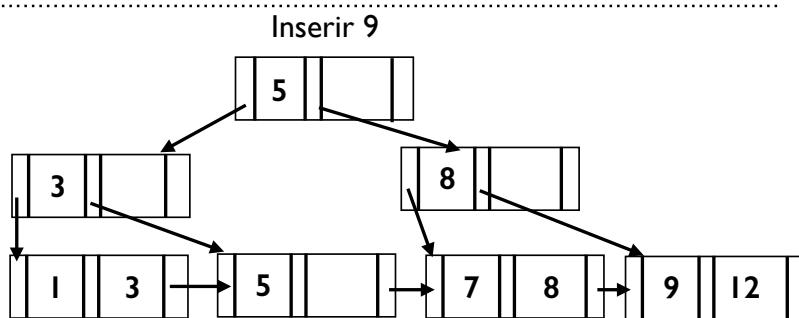


## INSERÇÃO EM ÁRVORE B+

Inserir 12 - não cabe no nó adequado. Nova divisão

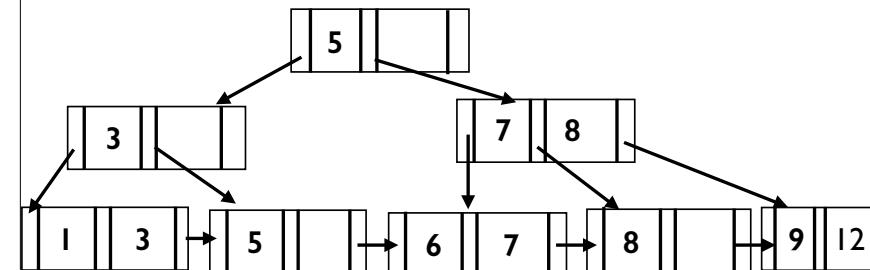


## INSERÇÃO ÁRVORE B+



## INSERÇÃO ÁRVORE B+

Inserir 6 - não cabe no nó adequado. Nova divisão



## EXERCÍCIO

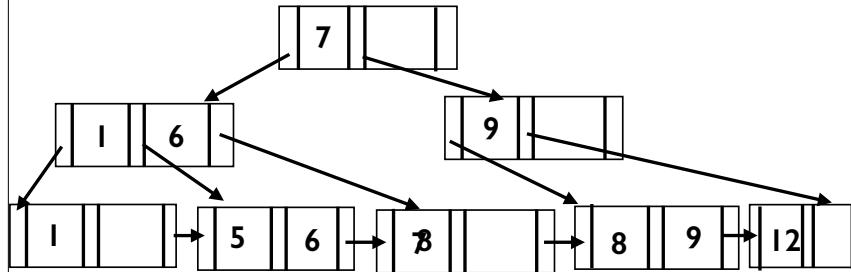
Construa uma árvore B+ para o seguinte conjunto de valores chave:  
(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Suponha que a árvore esteja inicialmente vazia e que os valores são adicionados na ordem descrita acima. Construa as árvores B+ para os casos em que o número de ponteiros que cabem em um nó são:

- a. Quatro
- b. Seis
- c. Oito

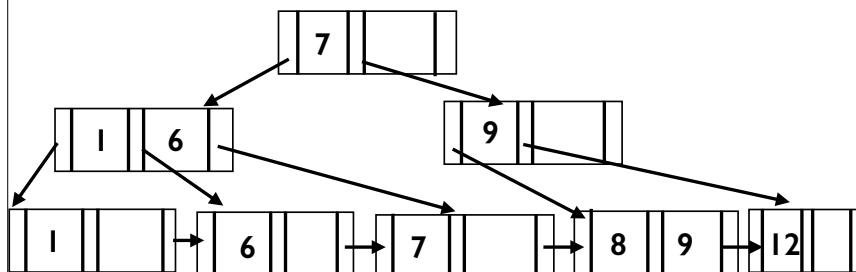
## EXCLUSÃO EM B+

Excluir 5 - Retira-se o 5 da folha



## EXCLUSÃO EM B+

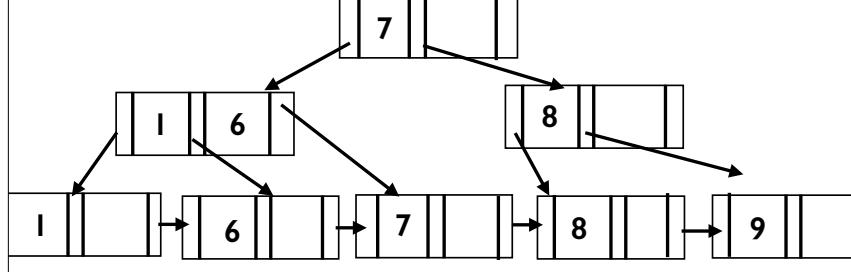
Árvore resultante da exclusão do 5



Agora, excluir 12 - underflow (redistribuição)

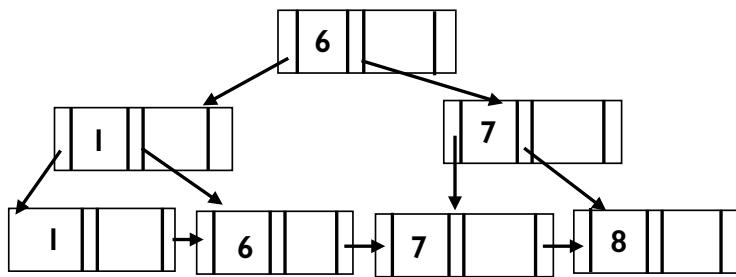
## EXCLUSÃO B+

Árvore resultante da exclusão do 12



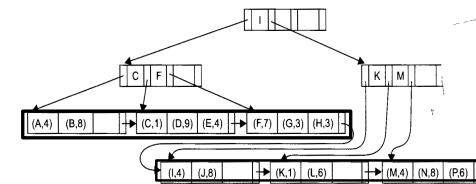
Excluir 9 - underflow (redistribuição)

## EXCLUSÃO B+



## ORGANIZAÇÃO DE ARQUIVO

- Estrutura de árvore B+ usada como organizador para registros em um arquivo
- Nós-folha sempre armazenam registros ao invés de ponteiros para registros

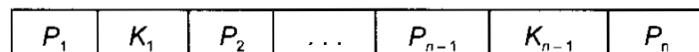


- Inserção e exclusão de registros e entradas de índice são tratadas da mesma maneira;
- Nós-folha adjacentes entre si na árvore podem estar localizados em diferentes locais do disco.

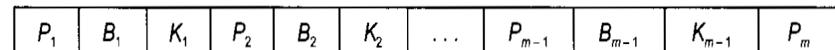
## ARQUIVOS DE ÍNDICE DE ÁRVORE B

- Elimina armazenamento redundante de valores de chave de busca
- Nós não folha tem ponteiro adicional que apontam para os registros de arquivo ou *buckets*
- Valores podem ser encontrados antes de chegar ao nó-folha
- Custo:
  - Maior altura
- Inserção e Exclusão mais complicada
- Vantagens não superam as desvantagens

## ARQUIVOS DE ÍNDICE DE ÁRVORE B

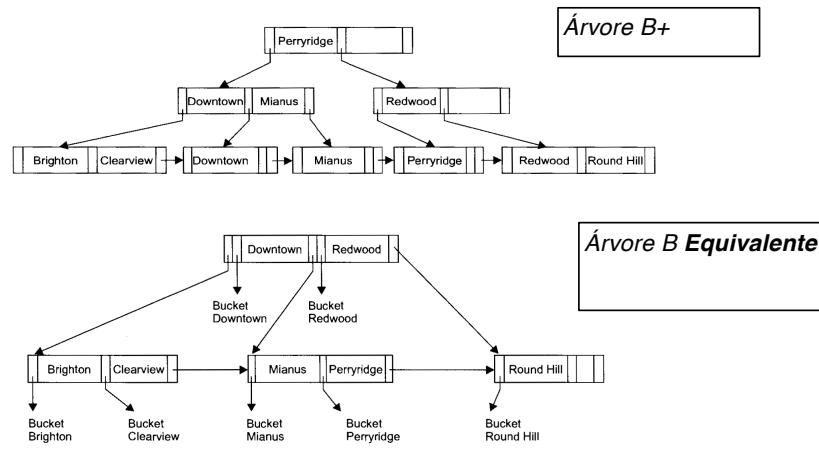


*Nó folha*



*Nó não-folha*

## ARQUIVOS DE ÍNDICE DE ÁRVORE B



## CLASSIFICAÇÃO

- Hash Estático: conjunto de buckets fixo
  - problemas com cadeias de overflow
- Hash Dinâmico: cria buckets de forma dinâmica para evitar ou reduzir cadeias de overflow
- Hash Extensível: usa mais um nível de indireção
  - diretório indexado por uma sequência de bits

**BUCKETS:** Unidade de armazenamento para um ou mais registros. Ex: bloco de disco

# HASHING

Baseia-se na distribuição uniforme dos valores determinados por uma função (função de hash)

## HASHING ESTÁTICO

- Evita o acesso a uma estrutura de índice
- Função de *hash*
  - K: conjunto de todos os valores de chave de busca
  - B: conjunto de todos os endereços de bucket
  - h: Função de *hash* (função de K para B)
- Pesquisa, inserção e exclusão:
  - Necessitam de calcular  $h(K_i)$
- Finalidades:
  - Organização de arquivo de *hash*
  - Organização de índice de *hash*

## ORGANIZAÇÃO DE ARQUIVOS

- Obtém-se diretamente o endereço do bloco do disco(bucket) onde o registro de dados se encontra
- Através da aplicação da função *hash* sobre o valor da chave de busca
- A função deve ser fácil de se computar
- Uma escolha comum: **K mod B**, onde B é o número de buckets disponíveis
- Para chaves de strings, cada caractere é tratado como um inteiro; soma-se os inteiros; divide somatório por B e pega o resto

## FUNÇÕES HASH

- Pior função
  - Mapeia todos os valores de chave de busca para o mesmo bucket
- Função ideal
  - Distribui as chaves armazenadas uniformemente por todos os buckets
  - Distribuição aleatória
    - o valor de *hash* não estará relacionado a qualquer ordenação externamente visível sobre os valores da chave de busca.
- Função típica
  - Cálculos sobre a representação binária interna à máquina de caracteres da chave de busca

## EXEMPLO

$f(\text{nome_agencia}) = \text{soma das representações binárias dos caracteres de uma chave e então retorna o módulo (MOD) da soma pelo número de buckets}$

Bucket 0			
Bucket 1			
Bucket 2			
Bucket 3	Brighton	A-217	750
	Round Hill	A-305	350
Bucket 4			
Bucket 5	Perryridge	A-102	400
	Perryridge	A-201	900
	Perryridge	A-218	700
Bucket 6			
Bucket 7	Mianus	A-215	700
Bucket 8	Downtown	A-101	500
	Downtown	A-110	600
Bucket 9			

Organização de *hash* do arquivo conta

## OVERFLOW(ESTOURO) DE BUCKET

- Motivos
  - Buckets insuficientes
    - $n_B > n_r / f_r$ ,
    - $n_B$  – número de buckets
    - $n_r$  - número total de registros armazenados
    - $f_r$  - número de registros que cabem num bucket
  - Pressupõe conhecimento do total de registros
- Desequilíbrio (*skew*)
  - Buckets recebem mais registros que outros
    - Registros com a mesma chave de busca
    - Função de *hash* escolhida não distribui uniformemente

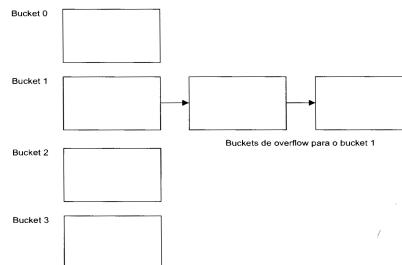
## TRATAMENTO DE OVERFLOW DE BUCKET

- Número de buckets:  $(n_r/f_r) * (1+d)$ 
  - d: fator de *fudge* ( $\sim 0,2$ )
- Cerca de 20% do espaço nos buckets estará vazio
- Ainda assim pode ocorrer *overflows*

### Bucket de overflow

- Encadeamento de estouro

- Lista ligada de buckets de overflow



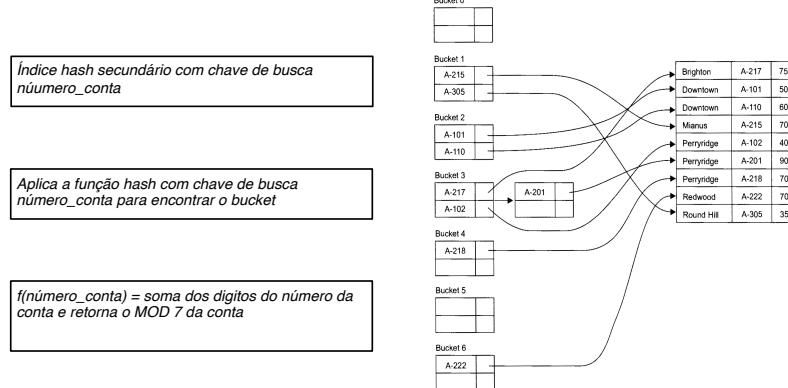
## EXERCÍCIO

- Considere um índice hash com 10 buckets. A seguinte função hash para números inteiros:

$f(i) = i^2 \text{ mod } 10$  é considerada boa? Por quê?

## ÍNDICES HASH

- Organiza as chaves de busca, com seus ponteiros associados em uma estrutura de arquivo de *hash*



## HASHING DINÂMICO

### Hashing Estático

Escolher função com base no tamanho do arquivo atual

Escolher função com base no tamanho antecipado do arquivo em um ponto no futuro

Reorganizar periodicamente a estrutura de hash em resposta ao crescimento do arquivo

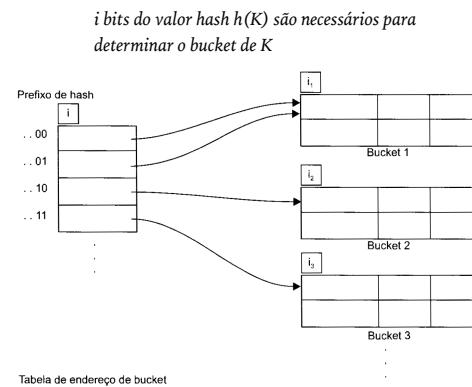
### Hashing Dinâmico

Função modificada dinamicamente para acomodar o crescimento ou encolhimento do banco de dados

Hashing extensível

## HASHING EXTENSÍVEL

- Divide e une os *buckets* enquanto o banco de dados cresce e curta
- Eficiência do espaço é mantida
- Reorganização acontece em apenas um *bucket* por vez
- Função gera valores por intervalos relativamente grandes
- Buckets* criados por demanda



## HASHING EXTENSÍVEL

- Aplica-se a função *hash* nas chaves obtendo-se um número binário
- Desse número devemos escolher uma quantidade de bits que fará a diferenciação entre os índices a serem armazenados.

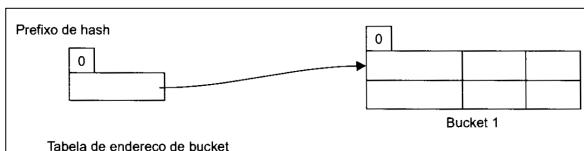
Brighton	A-217	750
Downtown	A-101	500
Downtown	A-110	600
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900
Perryridge	A-218	700
Redwood	A-222	700
Round Hill	A-305	350

nome_agência	$h(nome\_agência)$							
Brighton	0010	1101	1111	1011	0010	1100	0011	0000
Downtown	1010	0011	1010	0000	1100	0110	1001	1111
Mianus	1100	0111	1110	1101	1011	1111	0011	1010
Perryridge	1111	0001	0010	0100	1001	0011	0110	1101
Redwood	0011	0101	1010	0110	1100	1001	1110	1011
Round Hill	1101	1000	0011	1111	1001	1100	0000	0001

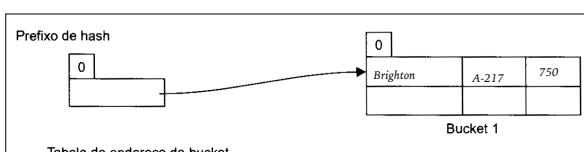
Valores hash de 32 bits para nome\_agencia

## CONSULTAS E ATUALIZAÇÕES

### Hashing extensível inicial

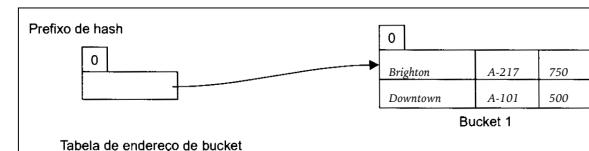


### Após uma inserção

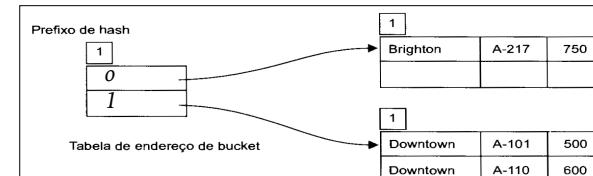


## CONSULTAS E ATUALIZAÇÕES

### Após duas inserções

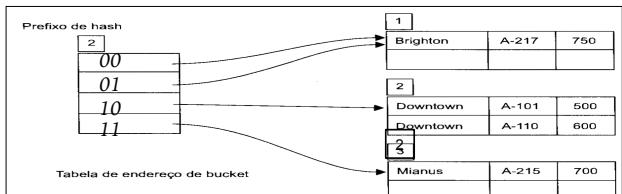


### Após três inserções

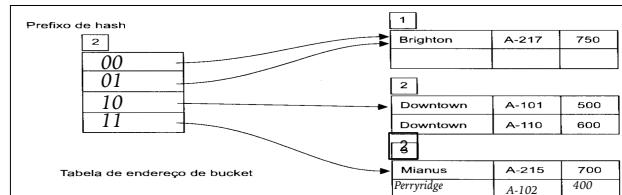


## CONSULTAS E ATUALIZAÇÕES

*Após quatro inserções*

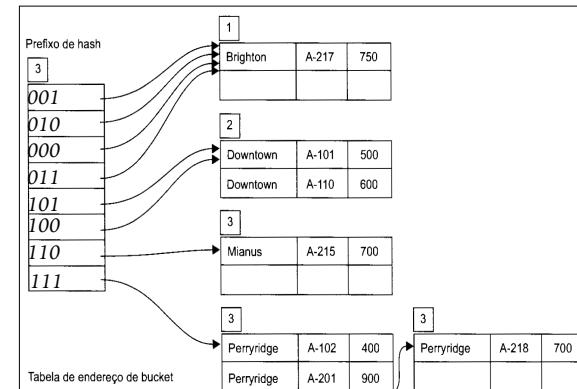


*Após cinco inserções*



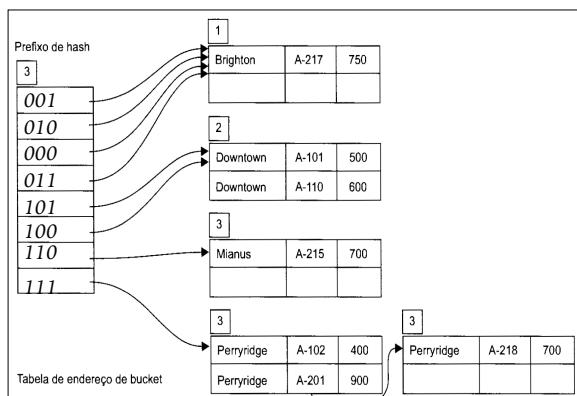
## CONSULTAS E ATUALIZAÇÕES

*Após seis inserções*



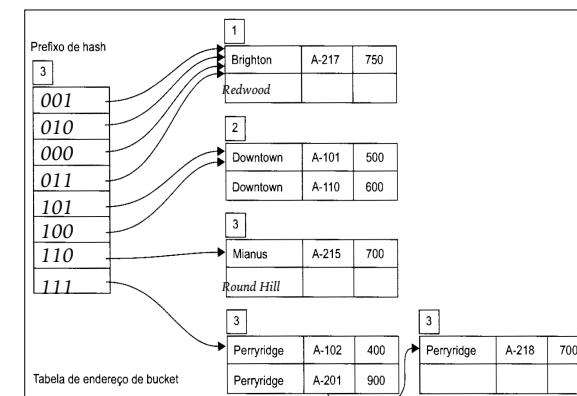
## CONSULTAS E ATUALIZAÇÕES

*Após sete inserções*



## CONSULTAS E ATUALIZAÇÕES

*Ao final das inserções*



## HASHING ESTÁTICO X DINÂMICO

- Vantagens do hashing dinâmico
  - Desempenho mantido quando o arquivo aumenta
  - Sobrecarga de espaço mínima
  - Buckets não precisam ser reservados
- Desvantagens do hashing dinâmico
  - Pesquisa com um nível de indireção adicional

## COMPARAÇÃO ENTRE INDEXAÇÃO ORDENADA E HASHING

- O custo da reorganização periódica do índice ou da organização *hash* é aceitável?
- Qual a frequência relativa de inserções e remoções?
- É desejável otimizar o tempo médio de acesso às custas do aumento no tempo de acesso no pior caso?
- Quais tipos de consultas têm maior probabilidade de acontecer?

## COMPARAÇÃO ENTRE INDEXAÇÃO ORDENADA E HASHING

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>➤ Indexação é bom para buscas por intervalo</li><li>➤ Facilidade de encontrar o próximo registro devido à ordenação</li><li>➤ Exemplo<ul style="list-style-type: none"><li>➤ SELECT saldo</li><li>➤ FROM conta</li><li>➤ WHERE saldo &gt;= 100 AND saldo &lt;=500</li></ul></li></ul> | <ul style="list-style-type: none"><li>➤ Hashing é bom para consultas exatas</li><li>➤ Tempo de busca constante</li><li>➤ Exemplo<ul style="list-style-type: none"><li>➤ SELECT saldo</li><li>➤ FROM conta</li><li>➤ WHERE saldo = 100</li></ul></li></ul> |
|---|---|

## DEFINIÇÃO DE ÍNDICE EM SQL

- Relembrando a sintaxe:  
`CREATE [UNIQUE] INDEX <nome Índice> ON <nome_relação> (<lista atributos>)`
- Exemplo:
  - `create index meu_idx on agencia(nome_agencia)`
  - `create unique index meu_idx on agencia(nome_agencia)`
- O atributo **unique** exige que a chave de procura seja uma chave candidata (não duplicada).
- Para remover um índice faz-se:
  - `DROP INDEX <nome_idx>`

## BENEFÍCIO DOS ÍNDICES

- Quando uma tabela não tem índices, os seus registros são desordenados e uma consulta terá que percorrer todos os registros
- O uso de índices pode ainda ser mais valioso em consultas envolvendo **joins** ou múltiplas tabelas:
  - Consultas em uma tabela → o número de valores que precisam ser examinados por coluna é o número de registros da tabela.
  - Em consultas em múltiplas tabelas, o número de possíveis combinações aumenta.

## BENEFÍCIO DOS ÍNDICES

- O resultado desta consulta deve ser uma tabela de 1.000 registros, cada um contendo três valores iguais.
- Se a consulta for executada sem o uso de índices, todas tuplas têm que ser percorridas.
- Consequentemente, o banco tenta todas as combinações possíveis para encontrar os registros que combinam com a condição da cláusula WHERE.
- O número de possíveis combinações é  $1.000 \times 1.000 \times 1.000 = 1.000.000.000$ .

## BENEFÍCIO DOS ÍNDICES

- Exemplo: Suponha que existam três tabelas sem índices, t1, t2 e t3, cada uma contendo as colunas c1, c2 e c3, respectivamente, e cada coluna contenha 1.000 registros com dados de 1 a 1.000. Suponha ainda a consulta abaixo:

```
SELECT t1.c1, t2.c2, t3.c3  
FROM t1, t2, t3  
WHERE t1.c1 = t2.c2 AND t2.c2 = t3.c3;
```

## BENEFÍCIO DOS ÍNDICES

- O uso de índices nas tabelas melhora o tempo de resposta da consulta, pois permitem que consultas sejam assim processadas:
  - Selecione a primeira tupla da tabela t1 e veja o seu valor;
  - Usando o índice da tabela t2, vá diretamente à tupla que combine com o valor de t1. Ainda, use o índice da tabela t3 para ir diretamente à tupla que combine com o valor de t2;
  - Vá para o próximo registro da tabela t1 e repita o procedimento anterior. Faça isto até que todas as linhas em t1 tenham sido examinadas.
- Ainda há uma varredura completa da tabela t1, mas índices são usados nas tabelas t2 e t3 para encontrar as tuplas diretamente.

## CUSTO DOS ÍNDICES

- Há custos de espaço e tempo. Na prática, as desvantagens tendem a serem minimizadas pelo valor das vantagens.
- Índices aumentam a velocidade de consultas, mas tornam mais lentas as operações de escrita (inserções, atualizações e remoções), pois gravar uma tupla requer não apenas escrevê-la, mas também mudar a ordenação dos índices.
- Quanto mais índices houver em uma tabela, mais mudanças de ordenação necessitam ser feitas, o que pode comprometer o desempenho.

## A ESCOLHA DOS ÍNDICES

- Índices devem ser criados em colunas usadas para pesquisa, ordenação ou agrupamento, ou seja, nas que aparecem na cláusula WHERE, em *joins*, ORDER BY ou GROUP BY.
- Colunas que aparecem em cláusulas JOIN ou em expressões como `col1 = col2` na cláusula WHERE são fortes candidatas à criação de um índice.
- É importante utilizar índices em chaves estrangeiras, já que estas são muito utilizados em *joins*

## A ESCOLHA DOS ÍNDICES

- Em determinados casos, em que haja várias consultas que utilizem os comandos ORDER BY, GROUP BY e DISTINCT é aconselhável criar um índice para a coluna que está sendo utilizada nestas consultas.
- Isso se deve ao fato de que cada vez que ocorre isto, o SGBD dispara um SORT para ordenação dos dados, o que pode comprometer o desempenho.
- Tendo índices para este caso, os dados já poderão estar ordenados, implicando em economias no processamento.

## ÍNDICES – DICAS PRÁTICAS

- Índices funcionam melhor em colunas que contenham um alto número de valores distintos.
- Mantenha, sempre que possível, chaves primárias pequenas (os SGBDs criam um índice para cada chave primária).
- Não crie índices em excesso. Índices devem ser atualizados e reorganizados quando o conteúdo de uma tabela é modificado. Logo, quanto mais índices, mais demorada será a atualização e a ordenação.

## ÍNDICES – DICAS PRÁTICAS

- Crie índices com valores pequenos. Use tipos de dados o menor possível. Por exemplo, não use uma coluna BIGINT se MEDIUMINT suporta os dados que serão armazenados.
  - Não use CHAR(100) se nenhum dos valores armazenados ultrapassa 25 caracteres.
- Valores pequenos melhoram o processamento de índices de muitas maneiras:
  - Podem ser comparados mais rapidamente
  - Ocupam menos espaço de disco nos arquivos de índices
  - É possível a permanência de mais registros em *cache*, fazendo com que o servidor tenha menos acessos a disco.

## ÍNDICES – DICAS PRÁTICAS

- Se um índice é raramente ou nunca utilizado, este diminui, desnecessariamente, o desempenho da tabela.
- Além disso, índices desnecessários podem fazer com que o otimizador de consultas não escolha o melhor índice a ser usado.
- As expressões de índice devem ser utilizadas somente quando as consultas que usam o índice forem muito frequentes.

## SQL SERVER

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX  
index_name  
ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )  
[ WITH <backward_compatible_index_option> [ ,...n ] ]  
[ ON { filegroup_name | "default" } ]
```

## ORACLE

```
CREATE [ONLINE|OFFLINE] [UNIQUE|FULLTEXT|SPATIAL]  
INDEX index_name  
[index_type]  
ON tbl_name (index_col_name,...)  
[index_option] ...  
  
index_col_name:  
col_name [(length)] [ASC | DESC]  
  
index_type:  
USING {BTREE | HASH}  
  
index_option:  
KEY_BLOCK_SIZE [=] value  
| index_type  
| WITH PARSER parser_name
```

## MYSQL

---

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
[index_type]
ON tbl_name (index_col_name,...)
[index_type]

index_col_name:
col_name [(length)] [ASC | DESC]

index_type:
USING {BTREE | HASH}
```