Unidade II

Indexação e Hashing

- Índices Ordenados
- Índices em Árvore
- Hashing Estático
- Hashing Dinâmico

Indexação

- Indexação permite a rápida localização de dados
  - Índices são menores que o BD → busca mais rápida
  - Exemplos: índices de livros, listas telefônicas, acervo de biblioteca ordenado por título/autor/ISBN, etc.
  - Ordenação em *hashing* é uma alternativa aos índices
- Chave de Procura (*Search Key*): um ou mais atributos usados para procurar entradas no BD
- Índices possuem entradas com a forma geral:

Chave	Ponteiro
Chave	Ponteiro
...	...

Indexação

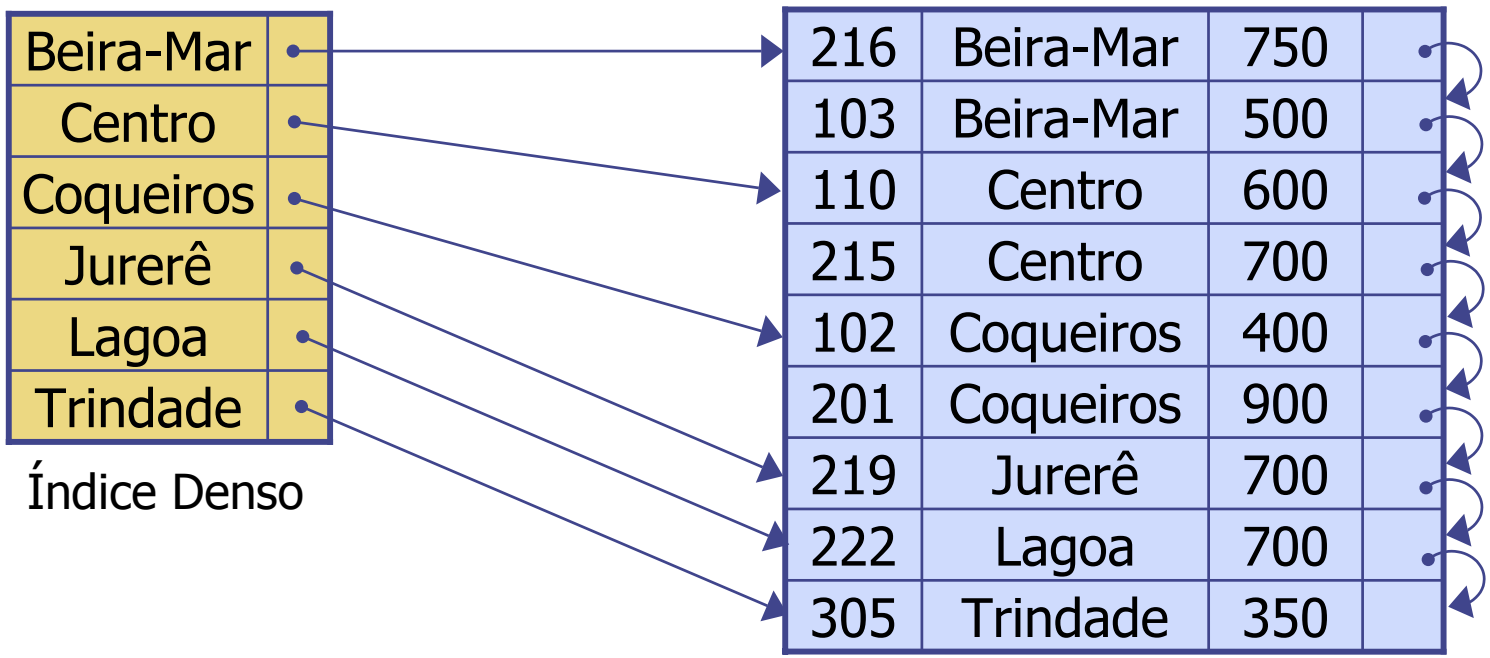
- Fatores que devem ser considerados ao construir índices:
  - Tempo de acesso aos dados
  - Tempo de inserção de remoção de entradas
  - Uso de espaço em disco
- Tipos principais de índice
  - Índices Ordenados: valores da chave de procura aparecem em ordem no índice
  - Índices em Árvore: valores da chave dispostos nas folhas de uma árvore, que apontam para os dados
  - Índices *Hash*: usam função de *hash* para localizar os valores da chave de procura

Índices Ordenados

- Nos índices ordenados, as entradas do índice são ordenadas pelo valor da chave de procura
  - Ordem numérica, alfabética ou temporal, de acordo com o tipo da chave de procura
  - Desempenho cai com o aumento do BD
  - Requer reorganização periódica do índice
- Tipos de índice ordenado
  - Denso: uma entrada no índice para cada valor da chave no banco de dados
  - Esparso: somente alguns valores de chave indexados
  - Multi-Nível: usa dois ou mais índices – um denso e os demais esparsos – para tornar procura mais eficiente

Índices Ordenados

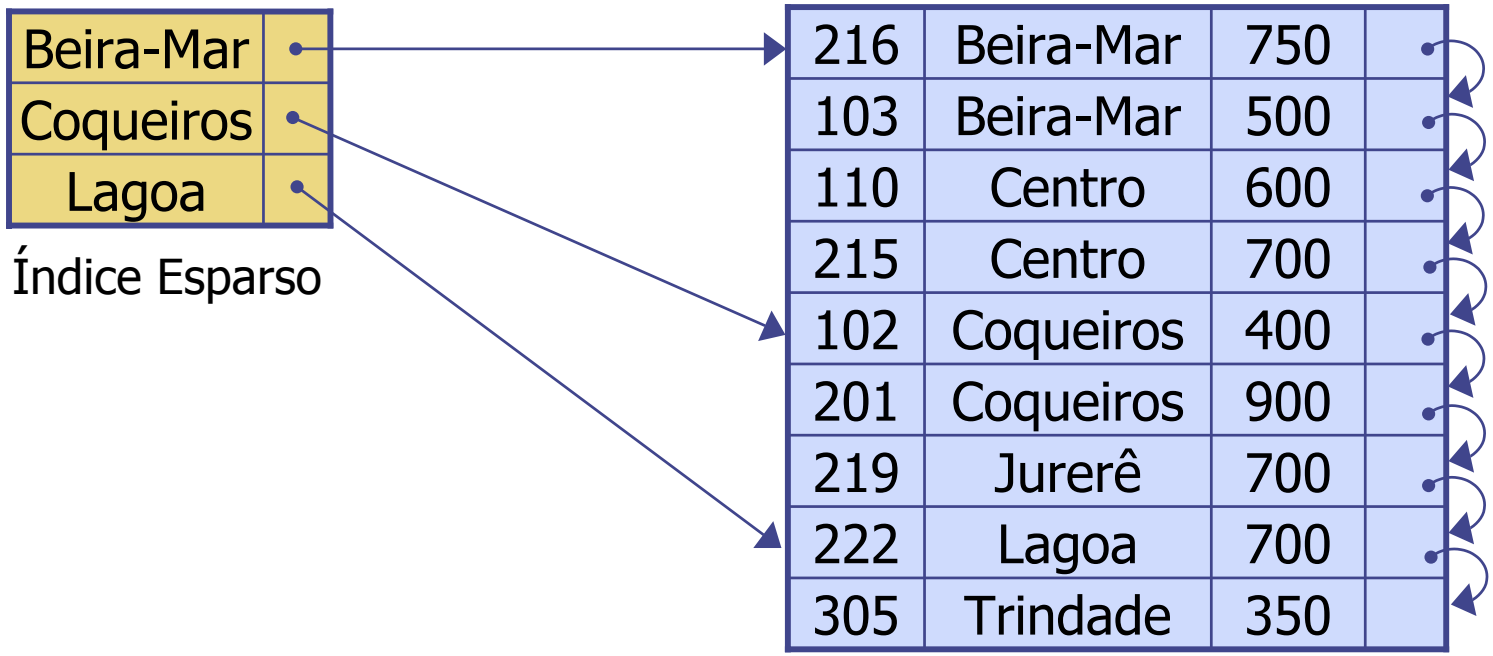
- Índice Denso
  - Uma entrada no índice para cada valor da chave
  - Índice pode se tornar muito grande e ineficiente



Adaptado de Silberschatz, Korth and Sudarshan

Índices Ordenados

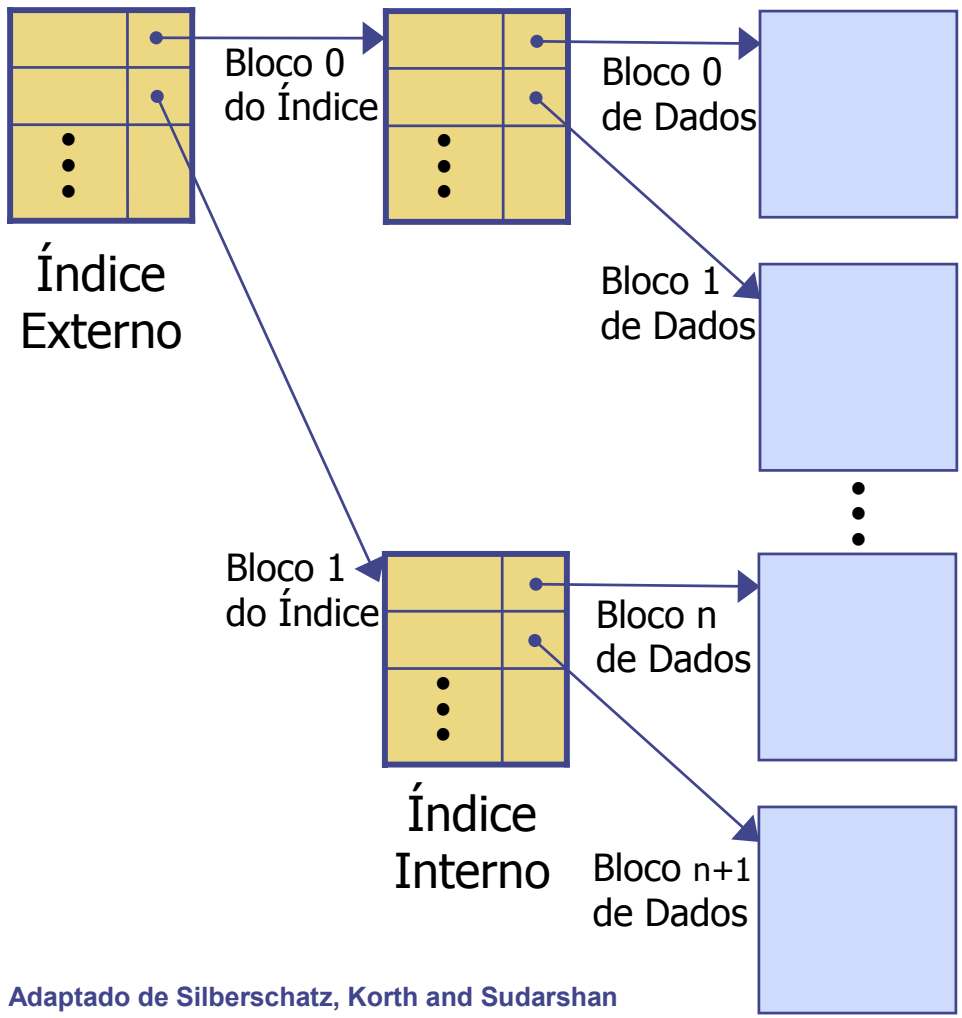
- Índice Esparso
  - Somente algumas entradas do BD são indexadas
  - Procura-se entrada anterior no índice, e depois no BD
  - Procura feita em duas etapas, mas o índice é menor



Adaptado de Silberschatz, Korth and Sudarshan

Índices Ordenados

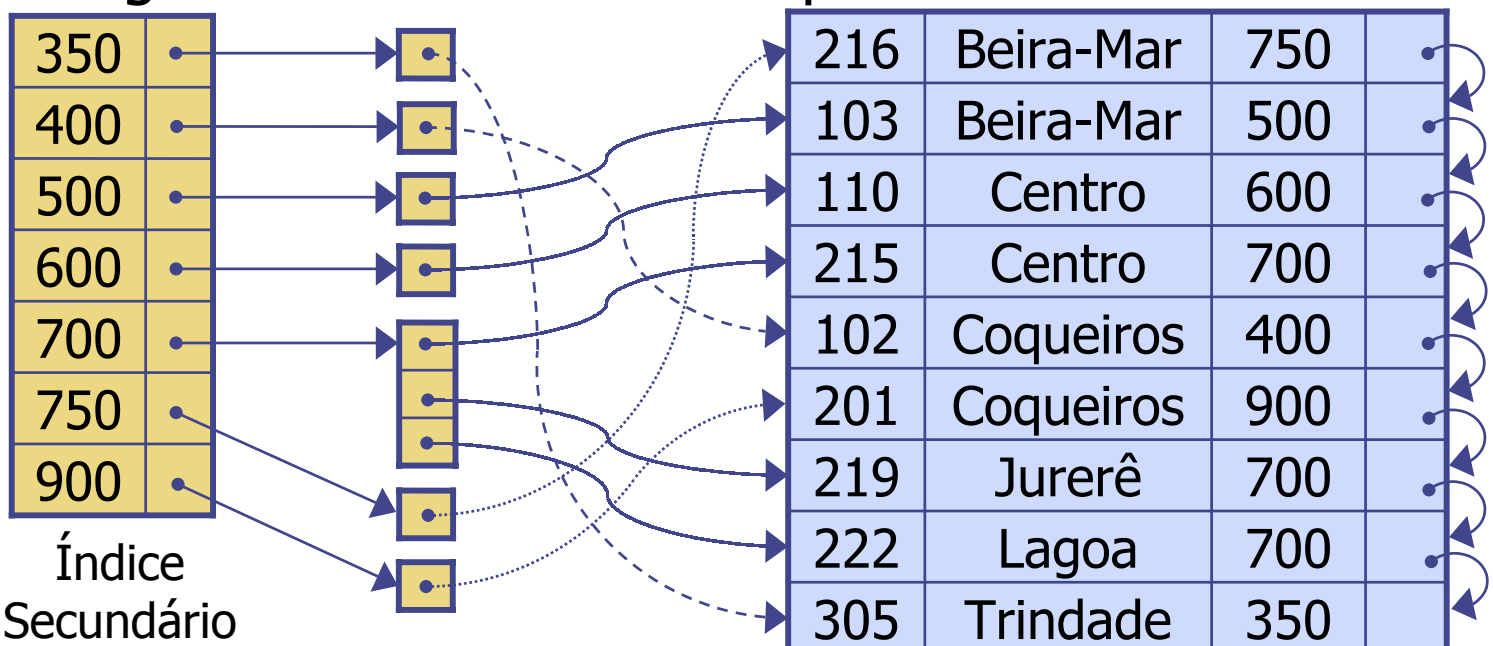
- Índice Multi-Nível
  - Usado se o índice ocupar vários blocos do disco
  - Índice passa a ser um índice interno
  - Índice externo indexa os blocos do índice original
  - Criar mais níveis se índice externo ainda for grande



Adaptado de Silberschatz, Korth and Sudarshan

Índices Ordenados

- Um índice ordenado pode classificado como:
  - Primário: quando as entradas no índice estão na mesma ordem os que dados no BD
  - Secundário: quando a chave de procura do índice segue uma ordem não-sequencial dos dados no BD



Adaptado de Silberschatz, Korth and Sudarshan

Índices Ordenados

- Necessário atualizar os índices quando dados forem inseridos ou removidos do BD
- Inserção de Dados
  - Em índices densos, inserir a entrada no índice
  - Em índices esparsos, inserir somente se o dado tiver um valor de chave que não consta do índice e que exige muita procura no BD
- Remoção de Dados
  - Em índices densos, remover a entrada no índice
  - Em índices esparsos, se o dado removido for o único com aquele valor de chave, substituir pelo valor da chave de procura no dado subsequente do BD

Índices em Árvore

- Nos índices em árvore, as entradas do índice são colocadas nas folhas de uma árvore
  - Cada folha possui um pequeno número de entradas
  - É preciso percorrer vários níveis – ramos da árvore, partindo da raiz – para procurar entradas no índice
  - O número de níveis deve ser logaritmicamente proporcional ao tamanho do BD, para limitar a profundidade de busca e otimizar o acesso
- Tipos de árvore
  - Árvore B: folhas dispostas ao longo da árvore
  - Árvore B+: folhas somente nas extremidades dos ramos da árvore

Índices em Árvore

- Árvore B
  - Nó raiz e ramos: nível inicial e níveis intermediários

P_1	B_1	K_1	P_2	B_2	K_2	...	P_{M-1}	B_{M-1}	K_{M-1}	P_M

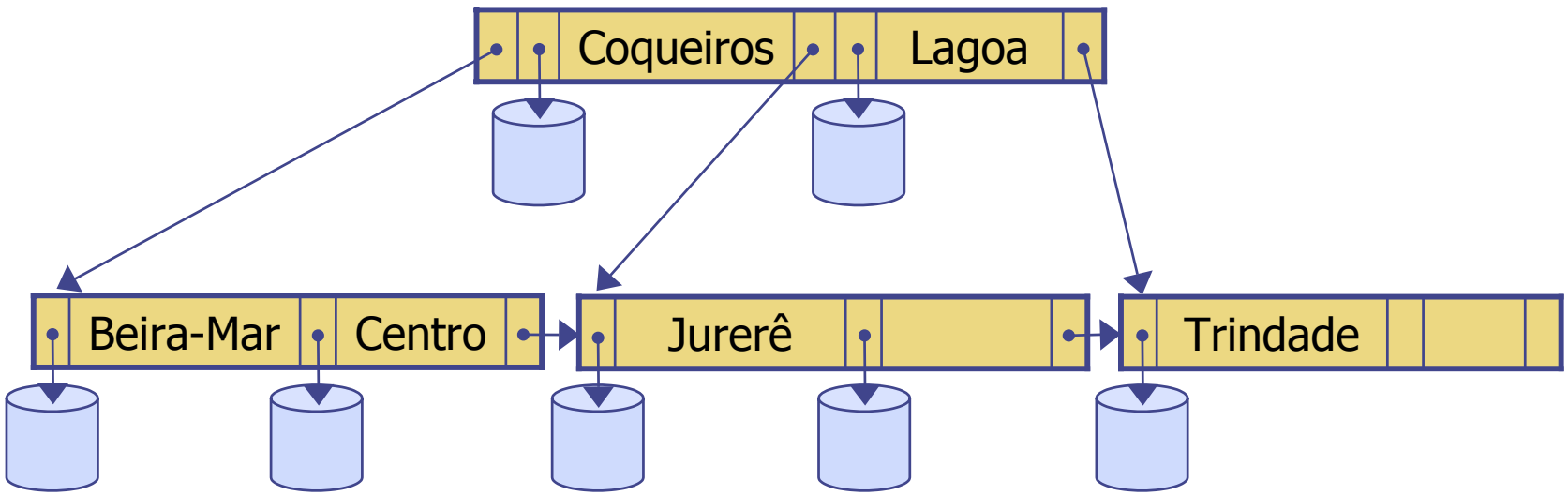
  - Nós folha: último nível

B_1	K_1	B_2	K_2	B_3	K_3	...	B_{N-1}	K_{N-1}	P_N

K_X	valor da chave
B_X	bloco de disco correspondente
P_X	entrada anterior no nível inferior
P_{X+1}	entrada posterior no nível inferior
P_N	a folha seguinte

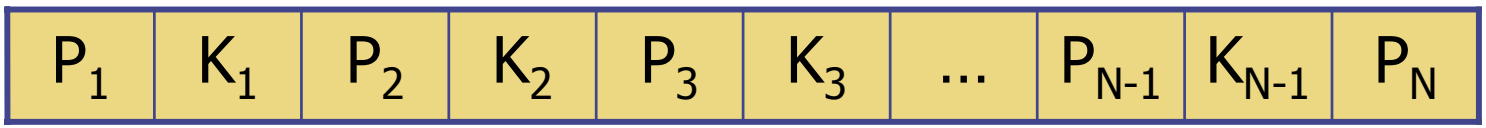
Índices em Árvore

- Profundidade de busca em Árvores B é variável
  - Os blocos de disco podem ser encontrados a partir dos níveis intermediários
- Exemplo de Árvore B com N=M=3



Índices em Árvore

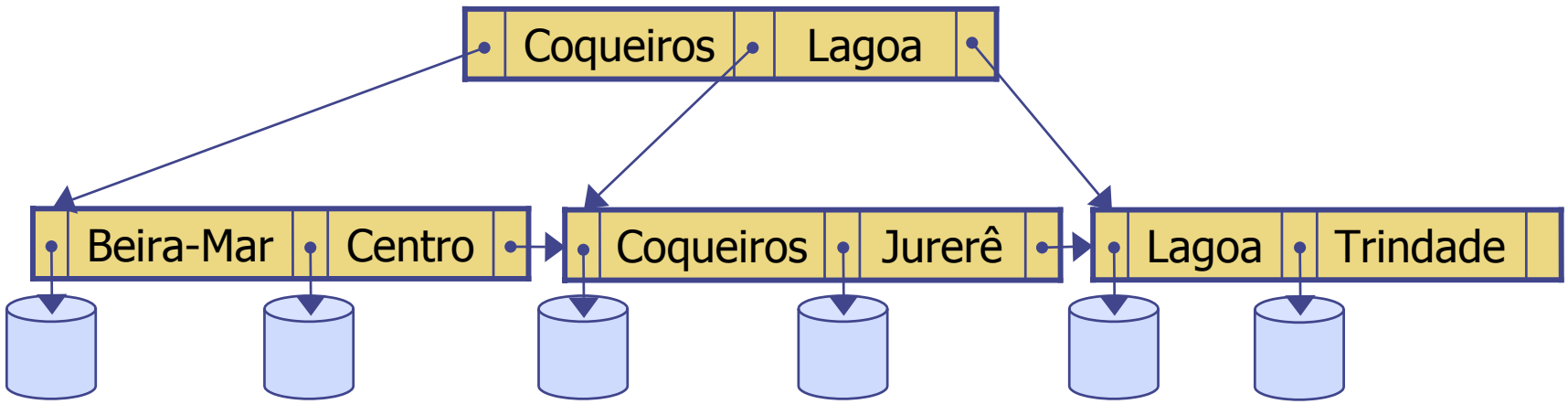
- Árvore B+
  - Cada entrada possui o valor da chave e um ponteiro
  - Ponteiros apontam para as entradas anterior e posterior no próximo nível, ou para o bloco de disco nas folhas da árvore
  - Entradas podem se repetir em diferentes níveis



- Árvores B+ são usadas com freqüência em BDs

Índices em Árvore

- A profundidade de busca é constante em Árvores B+
  - Os blocos de disco só podem ser encontrados no último nível da árvore (nas folhas)
- Exemplo de Árvore B+ com N=3

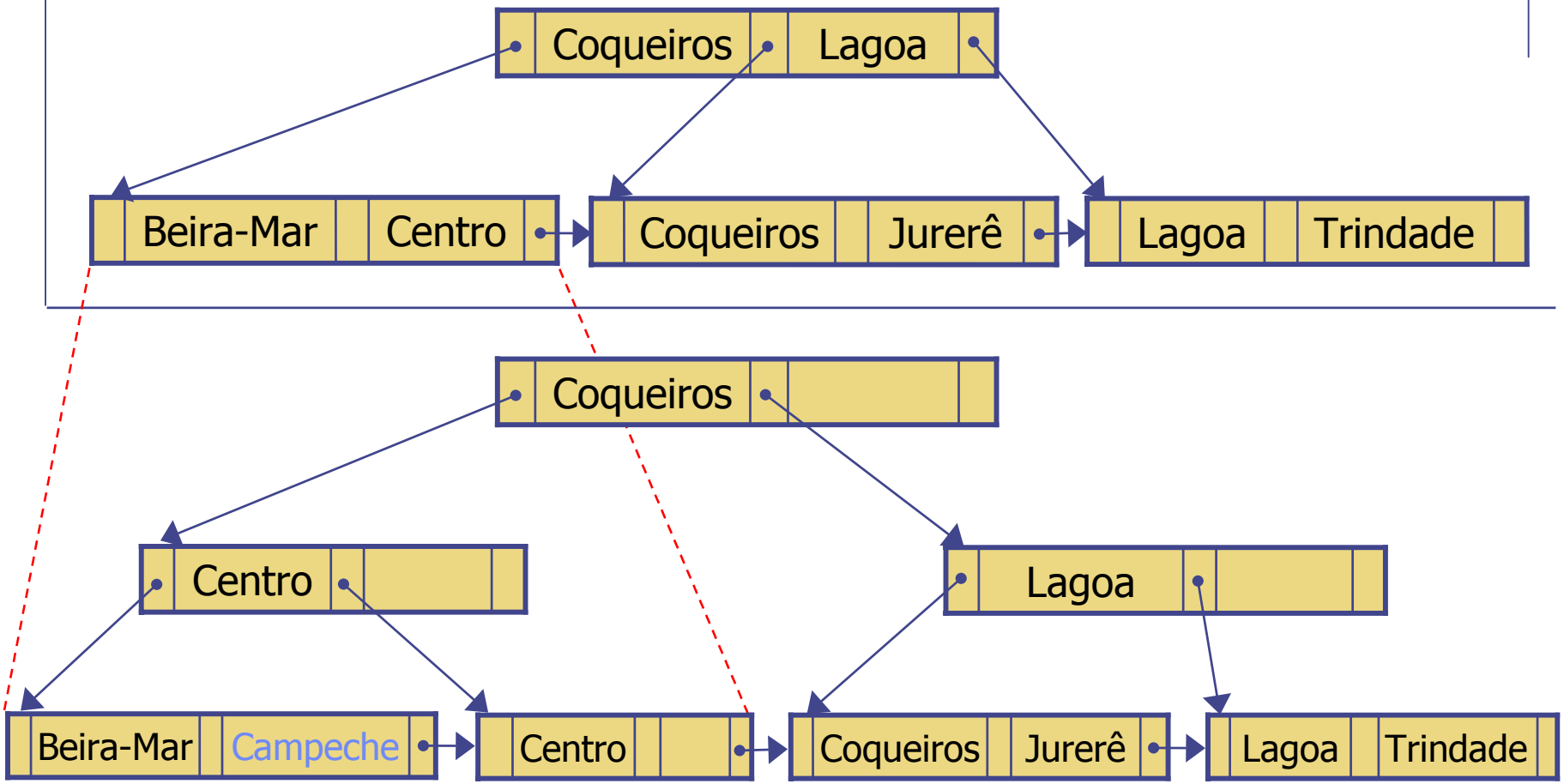


Índices em Árvore

- Inserção e remoção de dados
  - Além de inserir/remover entradas no índice, é preciso organizar a árvore (modificar ponteiros)
  - Carga de trabalho para inserção e remoção de dados é maior se comparada aos índices ordenados
- Regras para reorganizar a árvore
  - Unir folhas com menos que (N-1)/2 entradas
  - Dividir folhas com mais que N-1 entradas
  - Obs.: N é definido ao criar o índice

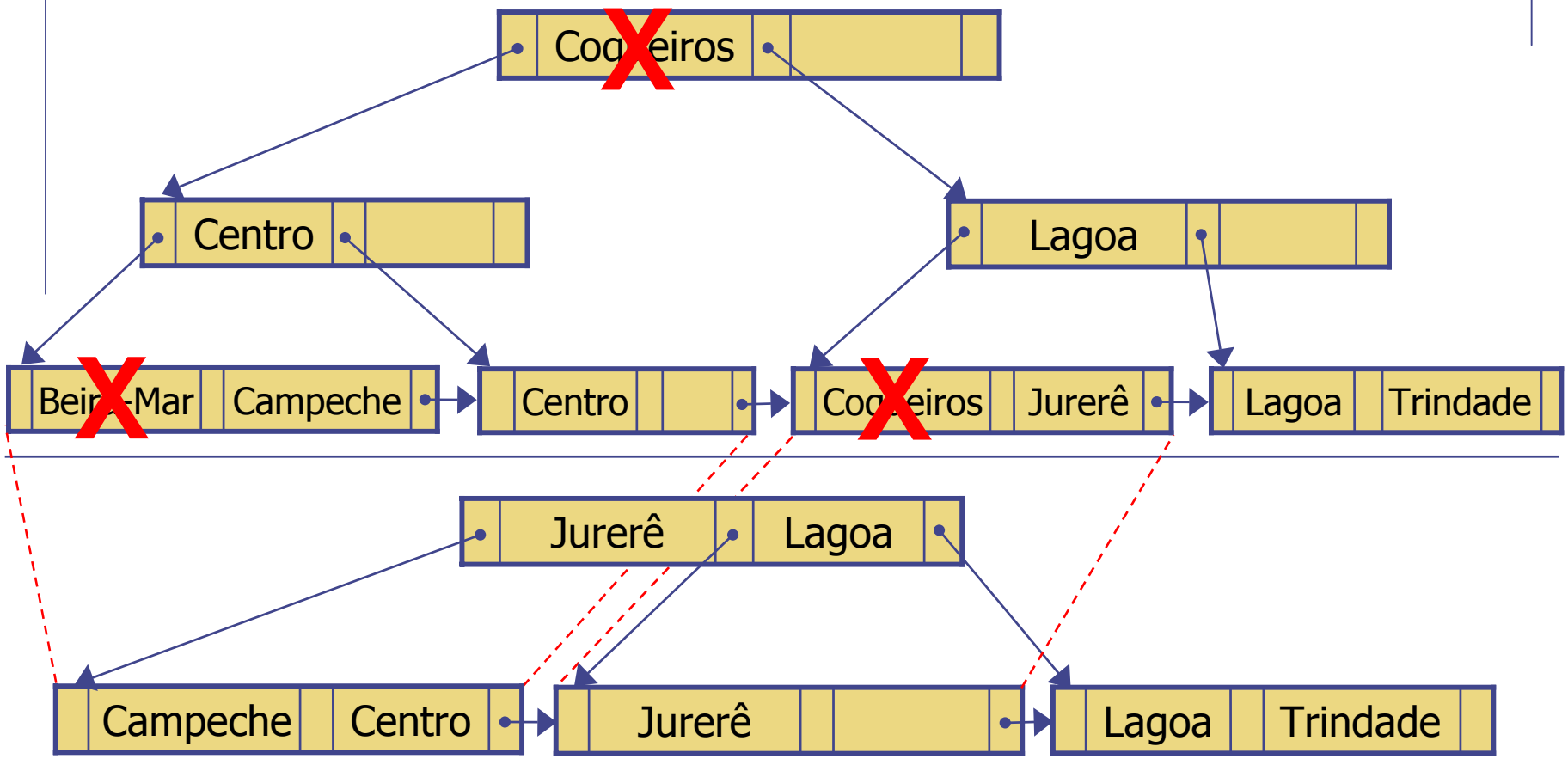
Índices em Árvore

- Exemplo de inserção de dados em árvores B+



Índices em Árvore

- Exemplo de remoção de dados em árvores B+



Índices em Árvore

- Vantagens e desvantagens de índices em árvore
  - Espaço de disco ocupado pelos índices em árvore B é maior que dos índices ordenados; árvores B+ são ainda maiores devido à repetição de entradas
  - Carga de trabalho para inserção e remoção de dados nos índices em árvore B ou B+ é maior, pois os índices precisam ser reorganizados
  - A reorganização automática dos índices em árvore B ou B+ elimina a necessidade de reorganização total que existe nos índices ordenados
  - Procura é geralmente mais rápida em árvores B+ que em árvores B e em índices ordenados
  - Procura de faixas de valores em árvores B é difícil

Hashing

- Organização sequencial dos dados implica em um excesso de operações de I/O para:
  - Acessar o índice
  - Buscar sequencialmente os dados
  - Inserir e remover dados na ordem sequencial
- Organização em *hashing* dá maior flexibilidade
  - Não exige ordenação
  - Dispensa o uso de índices
  - Localização de dados é extremamente rápida
  - Retorna diretamente a localização do bloco de disco com o dado com base no valor da chave de procura

Hashing

- Organização de BDs em *hashing*
  - Uma função de *hash* é aplicada às chaves de procura
  - Geralmente manipulam as chaves na forma binária
  - Resultado da função de *hash* pode ser o mesmo para diferentes valores de chave
  - Registros com mesmo resultado são colocados no mesmo bloco de disco
- A distribuição da função de *hash* deve ser:
  - Uniforme: a cada bloco de disco deve ser atribuído o mesmo número de valores de chave de procura
  - Aleatória: o resultado não deve seguir uma ordem facilmente previsível (numérica, alfabética, etc.)

Hashing Estático

- Princípio de funcionamento
  - O número de blocos de disco – a resposta da função de hash – é constante
  - A função de hash é usada para todo tipo de acesso ao BD – procura, inserção e remoção de dados
  - Os dados são organizados sequencialmente dentro dos bloco de disco
- Limitações
  - Pode ocasionar overflow dos bloco de disco
  - Ineficiente se o tamanho do BD variar muito (sobra espaço com BD vazio e falta quando estiver cheio)

Hashing Estático

- Exemplo:

  Função de *hash* retorna o módulo 4 do número da conta

Bloco 0

216	Beira-Mar	750

Bloco 1

201	Coqueiros	900
305	Trindade	350

Bloco 2

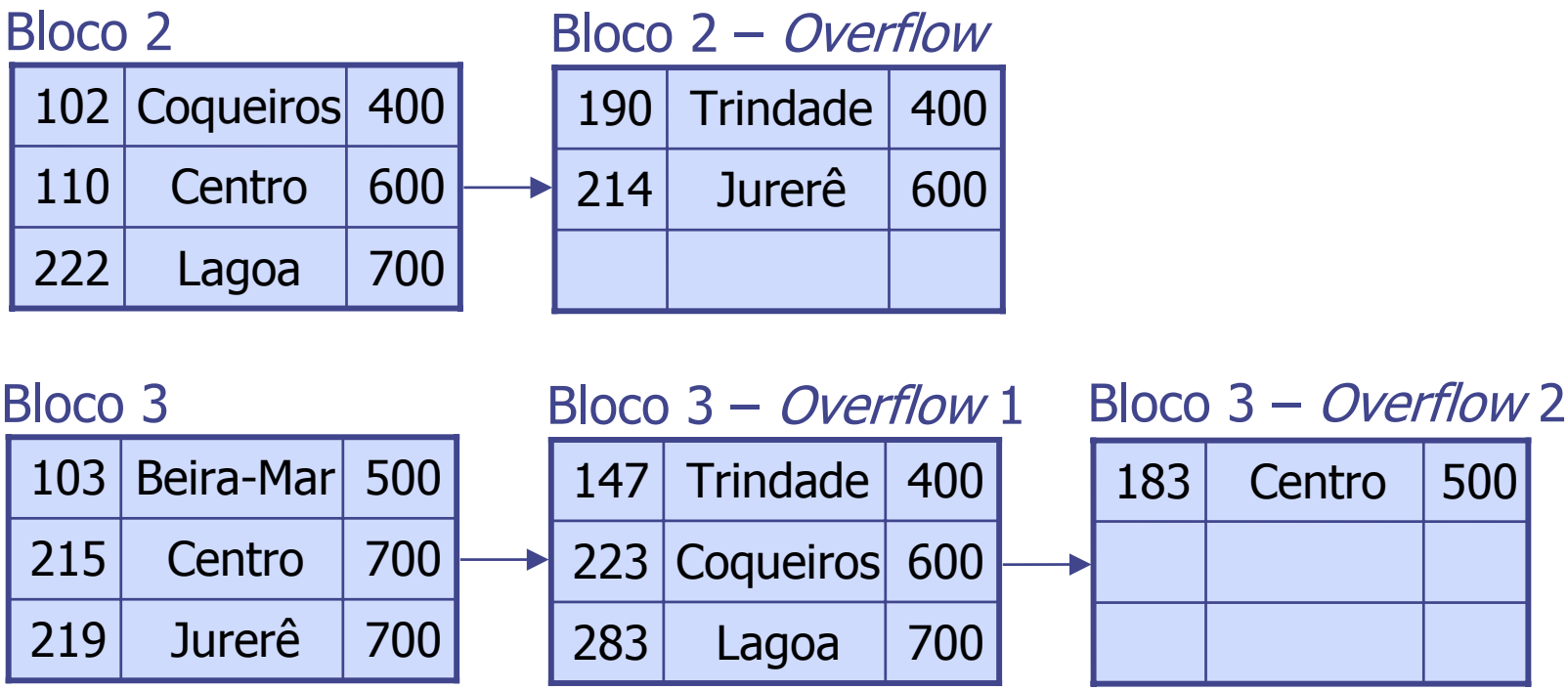
102	Coqueiros	400
110	Centro	600
222	Lagoa	700

Bloco 3

103	Beira-Mar	500
215	Centro	700
219	Jurerê	700

Hashing Estático

- Blocos de *overflow*
  - Criados se o tamanho do bloco for excedido
  - Bloco cheio aponta para o bloco de *overflow*



Hashing Dinâmico

- Princípio de funcionamento
  - O número de bloco de disco – a resposta da função de *hash* –varia em função da ocupação dos blocos
  - Se o tamanho do bloco for esgotado, a função de *hash* deve ser modificada e os blocos divididos
  - Blocos podem ser unidos se estiverem quase vazios
- Modificando a função de *hash* dinamicamente
  - Resposta da função de *hash* deve ter N bits
  - No início, considerar somente os M primeiros bits (M<N), e criar 2^M blocos no disco com tamanho X
  - Aumentar o número de bits à medida que o BD cresce, criando novos blocos e redistribuindo dados

Hashing Dinâmico

- Exemplo
  - Função de *hash* faz operação binária com agência
    h(Centro) = 00101101 h(Jurerê) = 10111101
    h(Coqueiros) = 11001101 h(Beira-Mar) = 11101011
  - Primeiros *bits* usados para identificar blocos de disco

0xxxxxxx		
110	Centro	600
215	Centro	700

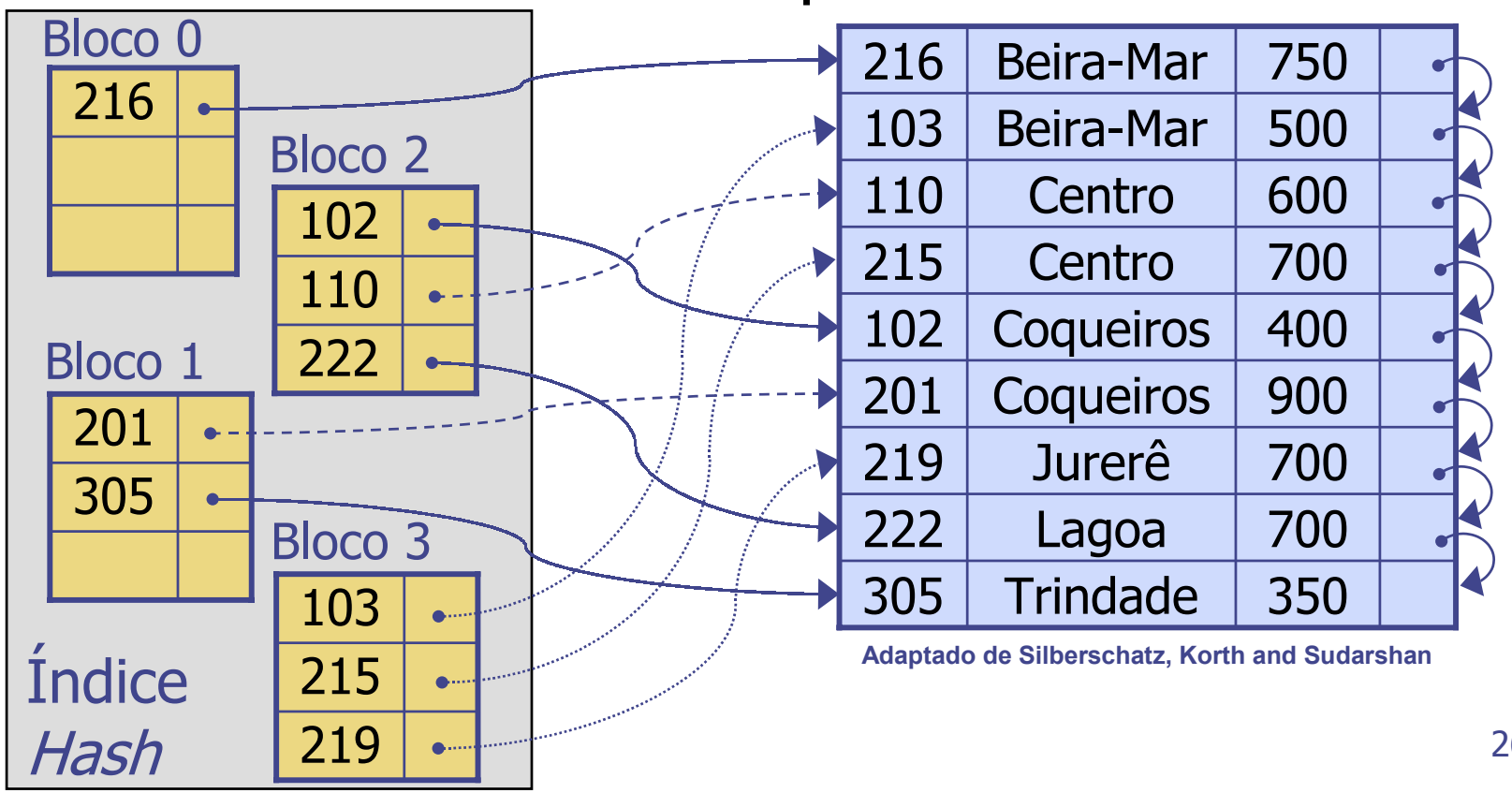
10xxxxxx		
219	Jurerê	700

110xxxxx		
102	Coqueiros	400
201	Coqueiros	900

111xxxxx		
216	Beira-Mar	750
103	Beira-Mar	500

Índices *Hash*

- São índices secundários organizados em *hash*
- Se o BD já é organizado em *hash*, não é necessário ter um índice primário



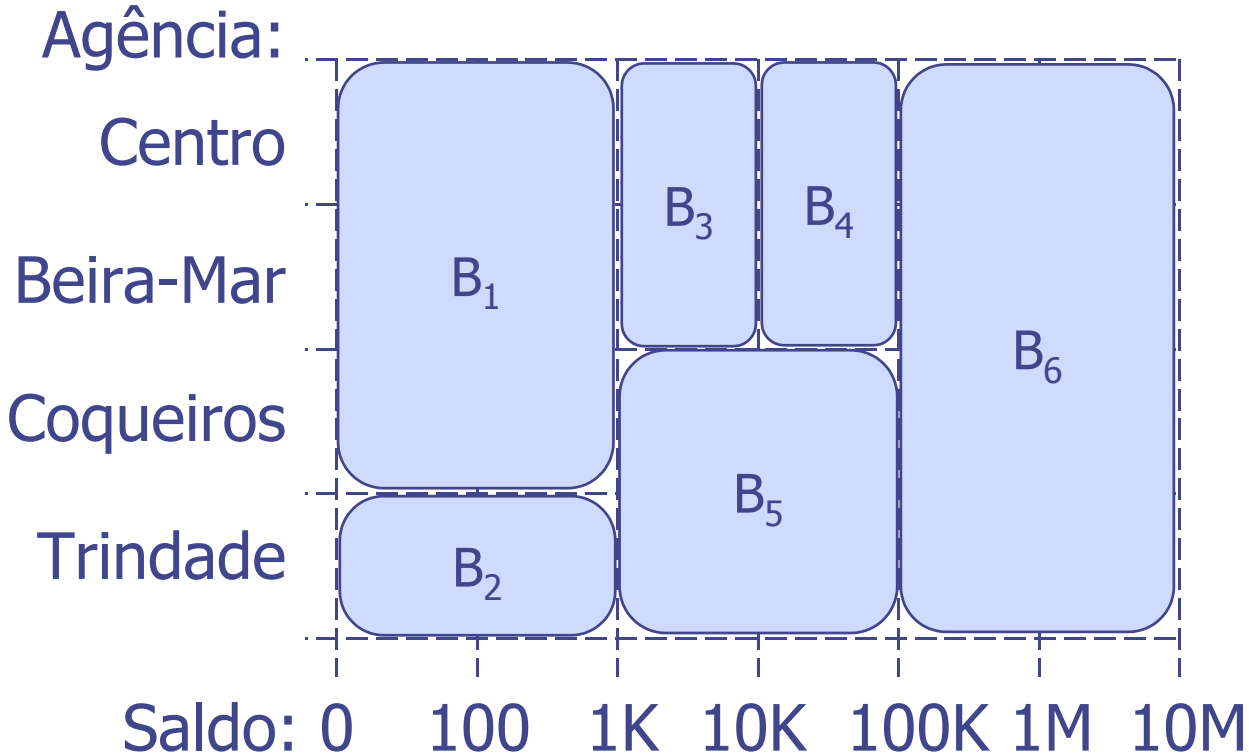
Adaptado de Silberschatz, Korth and Sudarshan

Hashing

- Vantagens e desvantagens
  - *Hashing* é eficiente para localizar dados com base em um valor de chave específico
    Ex: **SELECT** * **FROM** Banco
    WHERE Agencia = 'Downtown'
  - Faixas de valores são encontradas mais facilmente usando índices ordenados ou árvores B+, pois os dados podem estar em diferentes blocos de disco
    Ex: **SELECT** * **FROM** Banco
    WHERE Saldo **BETWEEN** 500 **AND** 1000

Grids

- Usados para agilizar procura com várias chaves



Adaptado de Silberschatz, Korth and Sudarshan

Prática: Índices em SQL

- Geralmente o SBD decide que índices criar
  - Usa chaves declaras como chaves de procura
  - Leva em conta as consultas mais frequentes
- O usuário pode criar índices explicitamente
  - SQL padrão não possui comandos para criação de índices, mas eles existem na maioria dos SBD
- Criação de Índices no SQL Server
  CREATE [**UNIQUE**] [**CLUSTERED**] **INDEX** *Índice* **ON** *Tabela* **(***Coluna* [**ASC**|**DESC**][,*Coluna* [**ASC**|**DESC**], ...]**)**
- Remoção de Índices no SQL Server
  DROP INDEX *Tabela.Índice*