Nota: Este material representa uma cópia resumida do capítulo 8,10 e 11 do livro Sistemas de Gerenciamento Sistemas de Gerenciamento de Banco de Dados - Ramakrishnan Gehrke (3ª edição), versão disponível em pdf na internet.

Estruturas de Indexação em Memória Externa - B-tree e Hash

Armazenamento Externo de Dados

A **organização de arquivos** é crucial para eficiência no armazenamento em disco, embora possa ter custos variados. A **indexação** é uma técnica vital para acessar registros de diferentes maneiras de forma eficiente.

Os discos são essenciais como dispositivos de armazenamento externo, com custos fixos por página. Se lermos diversas páginas na ordem em que estão armazenadas fisicamente, o custo pode ser muito menor do que o da leitura das mesmas páginas em ordem aleatória. A leitura sequencial pode ser mais eficiente

do que a leitura aleatória. As fitas são dispositivos de acesso sequencial e são usadas para dados arquivados que não precisam de acesso regular.

Cada registro possui um identificador único, o **rid**, que aponta para o endereço de disco da página que o contém. O **gerenciador de buffer** é responsável por ler dados para processamento e gravá-los no disco. O **gerenciador de espaço em disco** cuida do espaço disponível, alocando e desalocando páginas conforme necessário.

Organização de Arquivos de Indexação

Organizações de Arquivos de Indexação são estruturas essenciais em sistemas de gerenciamento de banco de dados (SGBD). Um **arquivo de registros** é a base dessa estrutura, podendo ser manipulado para inserção, exclusão e varredura de registros. Os registros são armazenados em **páginas** de disco, e a estrutura de arquivo mais simples é um arquivo não ordenado, ou **arquivo heap**.

Os **índices** são estruturas adicionais que otimizam operações de recuperação, permitindo a busca eficiente de registros que atendam a certos critérios nos campos de **chave de pesquisa (chave de procura - não confundir com super chave, chave candidata, chave primária)** do índice. No contexto dos bancos de dados, o **índice** toma forma de uma estrutura auxiliar, frequentemente na forma de um arquivo, associando-se a uma tabela específica. Sua finalidade é otimizar o tempo de acesso às linhas da tabela, criando ponteiros direcionados diretamente aos dados em colunas específicas, com base nas chaves de acesso. O índice armazena os valores das chaves de procura de forma **ordenada** e cada chave de procura associa os registros que a contêm. Considere a analogia com um catálogo para um livro em uma biblioteca:

- procura por autor: catálogo de autores
- procura por título: catálogo de títulos
- catálogo físico segue uma determinada ordem (alfabética, por exemplo)

Por que são eficientes?

- O número de blocos de índices é pequeno em comparação com o número de blocos de dados;
- Tendo em vista que as chaves são ordenadas, a pesquisa é rápida, podendo se usar um algoritmo de pesquisa binária;
- O índice pode ser pequeno o bastante para ser mantido permanentemente em buffers da memória principal.
- Nesse caso, uma pesquisa para uma determinada chave envolve apenas acessos à memória principal, sem precisar de operação de entrada e saída.

Resumo

- A indexação é uma pequena tabela que consiste em duas colunas.
- Dois tipos principais de métodos de indexação são 1) Indexação Primária 2) Indexação Secundária.
- O índice primário é um arquivo ordenado com tamanho fixo e dois campos.
    - A indexação primária também é dividida em dois tipos: 1) Índice denso e 2) Índice esparso.
        - Em um índice denso, um registro é criado para cada chave de pesquisa avaliada no banco de dados.
        - Um método de indexação esparso ajuda a resolver os problemas de indexação densa.
- O índice secundário no SGBD é um método de indexação cuja chave de pesquisa especifica uma ordem diferente da ordem sequencial do arquivo.
- O Cluster índice de processamento é definido como um arquivo de dados do pedido.
- A indexação multinível é criada quando um índice primário não cabe na memória.

- O maior benefício da Indexação é que ela ajuda a reduzir o número total de I/O necessárias para recuperar esses dados.
- É importante observar que, embora índices acelerem consultas, eles consomem espaço em disco e podem impactar operações de inserção, atualização ou deleção, pois o índice precisa ser mantido atualizado. Assim, é essencial avaliar cuidadosamente quais colunas serão indexadas.

Em índices muito grandes utilizam-se outras organizações:
- **Hashing**, caso a velocidade de acesso seja a maior prioridade
  - Acesso direto apenas
- **Árvores-B**, caso se deseje combinar acesso por chaves e acesso sequencial eficientemente

Índice Hashing

https://www.ic.unicamp.br/~thelma/gradu/MC326/2010/Slides/Aula08a-hash.pdf

O índice hashing é uma técnica de armazenamento e recuperação de dados que busca proporcionar acesso direto aos registros em um banco de dados. Esta técnica emprega uma função **hash**, que converte uma chave de entrada em um endereço específico em uma estrutura denominada **tabela hash**. A intenção é que a **função distribua** as chaves de forma **uniforme** pela tabela, **minimizando**, assim, a superposição de chaves no mesmo endereço (**colisão**).

Por exemplo, se no arquivo de registros de funcionários aplicarmos hashing no campo nome, podemos recuperar todos os registros de Joe. Nesta abordagem, os registros de um arquivo são agrupados em **buckets**, onde um bucket consiste em uma **página primária** e, possivelmente, páginas adicionais conectadas em uma cadeia.

O bucket ao qual um registro pertence pode ser determinado pela aplicação de uma função especial, chamada função hash, (ou função hashing) na chave de pesquisa. Dado um número de bucket, uma estrutura de índice baseada em hash nos permite recuperar a página primária para o bucket em uma ou duas E/S em disco.

A indexação por hashing é ilustrada na Figura 8.2, onde os dados estão armazenados em um arquivo sobre o qual se aplica hashing em idade; as entradas de dados neste primeiro arquivo de índice são os registros de dados reais. Aplicar a função hash no campo da idade identifica a página à qual o registro pertence. A função hash h para este exemplo é bastante simples; ela converte o valor da chave de pesquisa para sua representação binária e usa os dois bits menos significativos como o identificador do bucket.

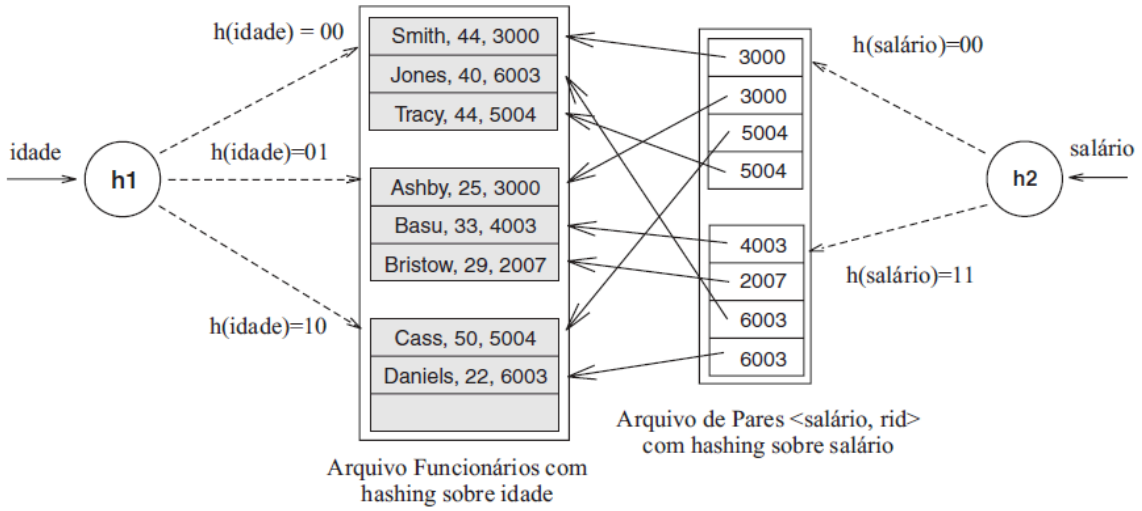


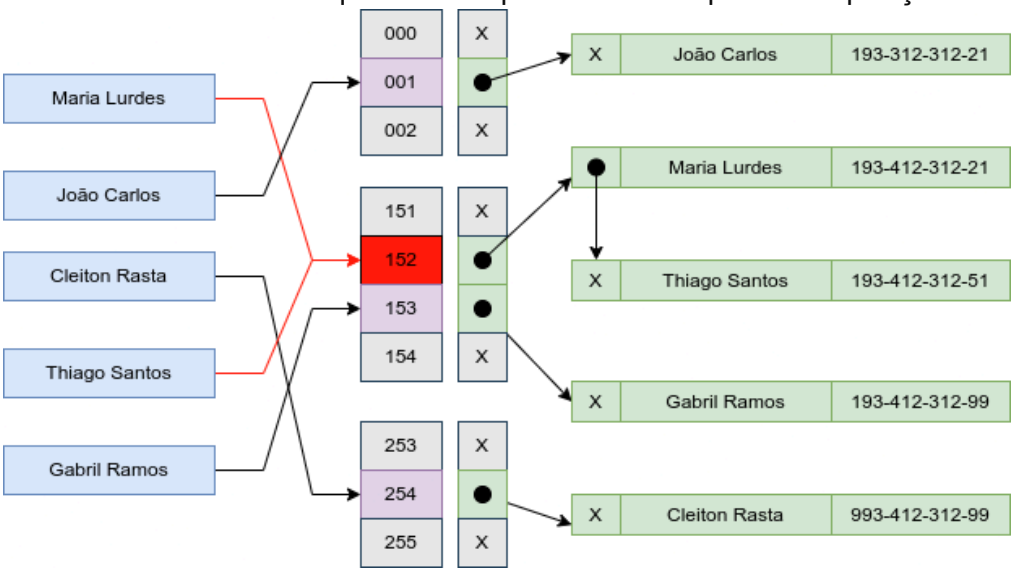
Figura 8.2 Arquivo organizado por índice com hashing sobre idade, com índice auxiliar sobre salário.

A Figura 8.2 também mostra um índice com chave de pesquisa salário que contém pares <salário, rid> como entradas de dados. O componente rid (abreviação de id do registro) da entrada de dados neste segundo índice é um ponteiro para um registro com valor de chave de pesquisa salário (e é mostrado na figura como uma seta apontando para o registro de dados). Ao arquivo de registros de funcionários é aplicada uma operação de hashing sobre idade. O segundo índice, sobre salário, também usa hashing para localizar entradas de dados, que agora são pares <salário, rid do registro do funcionário>.

Observe que a chave de pesquisa de um índice pode ser qualquer sequência de um ou mais campos e não precisa de registros identificados univocamente. Por exemplo, no índice do salário, duas entradas de dados possuem o mesmo valor 6003 para a chave de pesquisa. (Uma chave primária ou chave candidata, campos que identificam unicamente um registro, não está relacionada ao conceito de chave de pesquisa).

Quando ocorre uma colisão, ou seja, quando duas chaves distintas são mapeadas para o mesmo endereço na tabela hash, é necessário adotar estratégias para solucionar o conflito. Existem duas técnicas que você pode usar para evitar uma colisão de hash:

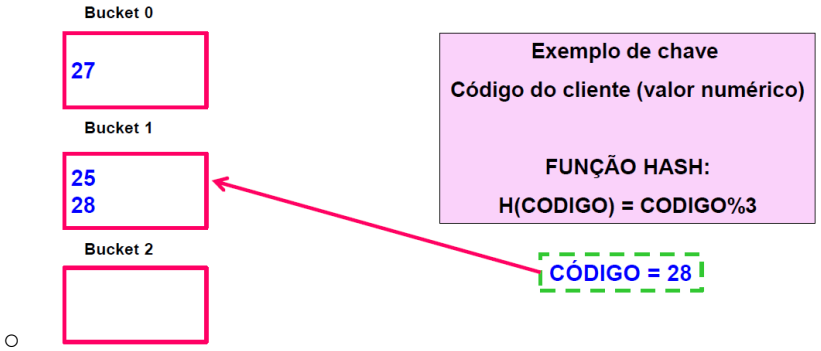
- **Refazendo:** Este método invoca uma função hash secundária, que é aplicada continuamente até que um slot vazio seja encontrado, onde um registro deve ser colocado.
- **Encadeamento:** o método de encadeamento cria uma lista vinculada de itens cuja chave tem hash para o mesmo valor. Este método requer um campo de link extra para cada posição da tabela.



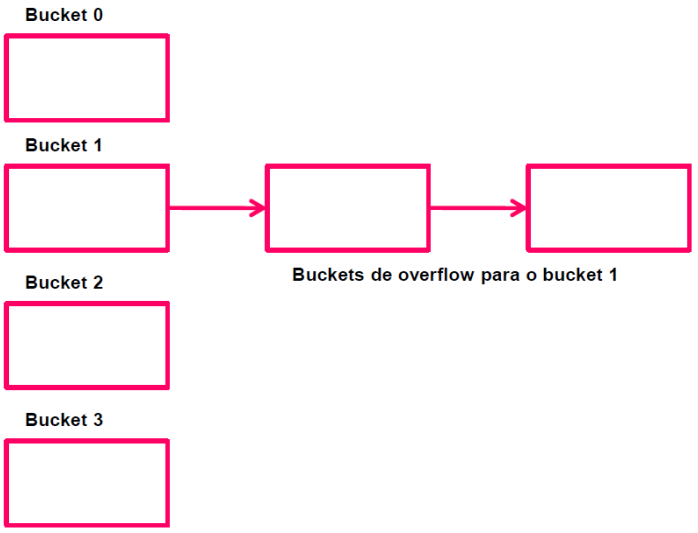
Definições

Organização de arquivos em Hashing **(hashing fechado)**

- obtém diretamente o endereço do bloco de disco que contém um registro desejado usando uma função sobre o valor da chave de procura;
- **bucket** -> tipicamente um bloco de disco, mas pode ser menor ou maior que o bloco.
    - Se bucket não tem espaço para armazenar registro -> **overflow** de bucket
    - Razões: buckets insuficientes ou desequilíbrio (skew)



    - Solução: bucket de overflow
        - se bucket b está cheio ao inserir um registro -> este é armazenado no bucket de overflow
        - se bucket de overflow cheio -> novo bucket de overflow
        - buckets de overflow de um bucket são encadeados em uma lista ligada
        - manipulação desse tipo de lista -> encadeamento de overflow



- K = conjunto de todos os valores de chaves de procura
- B = todos os endereços de bucket
- Função **hash h** = função de K para B
- **Inserção** de um registro com chave de procura Ki
    - calcula h(Ki): fornece endereço do bucket para aquele registro
    - registro é armazenado no bucket, se houver espaço
- **Procura** de um registro com chave de procura Ki
    - calcula h(Ki): fornece endereço do bucket para aquele registro

    - conferir o valor da chave de procura de todos os registros no bucket para verificar se o registro é um dos desejados *(porque duas chaves de procura podem ter o mesmo valor de hash).*
        - *se bucket b possuir buckets de overflow -> todos registros de todos os buckets de overflow de b devem ser examinados*
- **Remoção** de um registro com chave de procura Ki
    - *calcula h(Ki): fornece endereço do bucket para aquele registro*
    - *registro é removido do bucket*

Organização de arquivos em Hashing **(hashing aberto)**

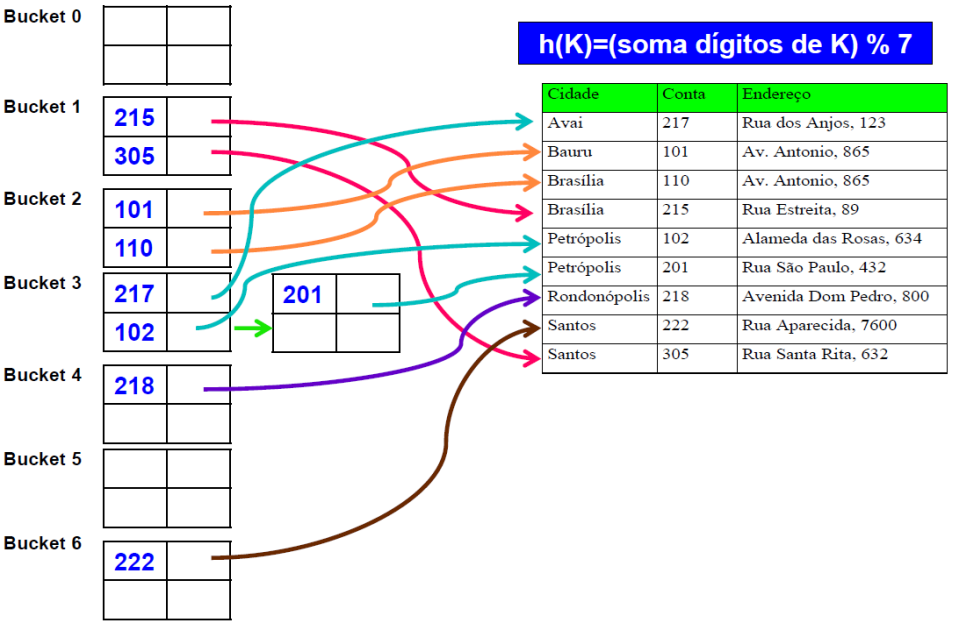
Conjunto fixo de buckets e não há cadeia de overflow

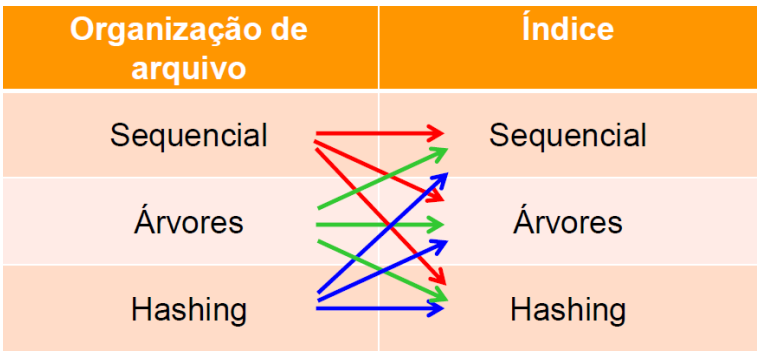
- são estabelecidas políticas para escolher novo bucket quando um bucket está cheio
- Exemplo de política -> registros inseridos em outro bucket que tem espaço (ordem cíclica) – **linear probing**
- outras políticas: calcular **funções hash adicionais**
- SGBD preferem hashing fechado. Por quê?
    - A remoção no hashing aberto é complicada.

Índices Hashing

- Termo índice hash -> denota estruturas de arquivo hash e também os índices hash secundários.
- Índice hash nunca é necessário como uma estrutura de índice primária -> se um arquivo é organizado usando hashing não é necessária estrutura de índice hash para ele.

Organiza as chaves de procura, com seus ponteiros associados, em uma estrutura de arquivo hash (exemplo considerando K=número da conta)





Hashing estático

O principal problema com o Hashing Estático é que o número de buckets é fixo. Se um arquivo diminuir muito, muito espaço é desperdiçado; mais importante, se um arquivo crescer muito, longas cadeias de overflow se desenvolvem, resultando em desempenho ruim. Uma alternativa simples ao Hashing Estático é executar um “hash” periodicamente sobre o arquivo para restaurar a situação ideal (sem cadeias de overflow, ocupação em torno de 80 por cento). Entretanto, executar o “rehashing” consome tempo e o índice não pode ser usado enquanto o “rehashing” estiver sendo executado. Outra alternativa é usar técnicas de **hashing dinâmico** como **Hashing Linear** e **Extensível**, que lidam com inserções e exclusões harmoniosamente. Analisamos estas técnicas no restante deste capítulo.

???Hashing dinâmico

Problema do Hashing estático -> definição do número de buckets
Opções:

- escolher função hash com base no tamanho atual do arquivo -> degradação de desempenho à medida que BD cresce
- escolher função hash com base no tamanho previsto do arquivo -> desperdício de espaço
- reorganizar periodicamente estrutura de hash, escolher função hash nova e gerar novas atribuições de bucket -> operação demorada (proibição de acesso durante a operação)

Técnicas de **hashing dinâmico** -> permitem modificar função hash dinamicamente para acomodar crescimento ou diminuição do BD.

- Uma das formas: hashing expansível (ou extensível)

Hashing Linear

Hash Linear

- Função hash varia
- Não há diretório de buckets
- Pode haver páginas de overflow à medida que os buckets se enchem
- Regularmente, a função hash se modifica e páginas de overflow são realocadas.

Neste método, quando ocorre uma colisão (ou seja, quando duas chaves são mapeadas para a mesma posição na tabela hash), a próxima posição livre na tabela é encontrada linearmente, uma por uma, até que uma posição vazia seja encontrada. A fórmula básica é:

$h(k,i)=(h'(k)+i) \bmod m$

Onde:

- h(k,i) é a função de hash linear que calcula a posição de índice para a chave k e a tentativa i.

- h′(k) é a função de hash original que mapeia a chave k para um índice inicial.
- m é o tamanho da tabela hash.
- i é a tentativa de resolução de colisão (começando com i=0).

A principal desvantagem do Hashing Linear é que, se a tabela hash estiver ficando cheia, pode haver um grande número de colisões, levando a um aumento na complexidade de pesquisa.

Hashing expansível (extensível)

https://www.ic.unicamp.br/~thelma/gradu/MC326/2010/Slides/Aula09a-hash-Extensivel.pdf

O Hashing Extensível é uma técnica mais sofisticada que visa minimizar o número de colisões e melhorar o desempenho em tabelas hash dinâmicas. Ele divide a tabela hash em várias páginas, onde cada página pode conter um número fixo de entradas.

Quando ocorre uma colisão, em vez de procurar linearmente por uma posição livre na tabela, o Hashing Extensível reorganiza a estrutura de dados, muitas vezes usando uma estrutura de árvore, para distribuir melhor as chaves e reduzir as colisões.

Uma das estruturas de dados comumente usadas no Hashing Extensível é a árvore B (ou uma variante dela). Nesse caso, quando uma colisão ocorre em uma página, a página é dividida em duas, criando uma nova página e redistribuindo as chaves entre elas.

Solução 1 : quando algum bucket ficar cheio
- Dobrar o número de buckets
- Distribuir as entradas nos novos buckets
- Defeito : o arquivo todo deve ser lido e reorganizado e o dobro de páginas devem ser escritas.

Solução 2 : utilizar um diretório de ponteiros para os buckets.
- Dobrar o número de entradas no diretório
- Separar somente os buckets que ficaram cheios.

Trata mudança no tamanho do BD por meio de **divisão** e **fusão** de **buckets**
- eficiência espacial mantida
- reorganização realizada apenas em um bucket por vez overhead de desempenho aceitável e baixo

Como funciona:
- escolhe função hash h com propriedades desejadas de uniformidade e aleatoriedade
- função gera valores dentro de grande faixa – inteiros binários de b bits (valor típico para b = 32)
    - 2^{32} -> mais de 4 bilhões de buckets – impossível!
    - o que se faz: criação de buckets por demanda.Não usa os b bits, mas i bits, sendo $0 \leq i \leq b$
- valor de i cresce ou diminui de acordo com o tamanho do BD

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

Valor de *hash* de 32 bits para a chave nome_cidade

Considerando que cabem 2 registros de dados em cada bucket

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

Estrutura inicial (vazia)

Hashing → Hashing dinâmico → Hashing expansível

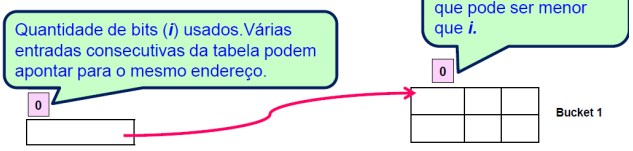
✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011					

Quantidade de bits (*i*) usados. Várias entradas consecutivas da tabela podem apontar para o mesmo endereço.

A cada **bucket** é associado um prefixo, que pode ser menor que *i*.



Estrutura inicial (vazia)

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

Estrutura após inserir 2 registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

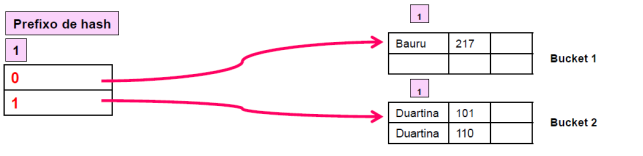
Terceiro registro não cabe no *bucket*. Tem que aumentar número de bits, dividir o *bucket*. *Dobra quantidade de entradas na tabela de endereços*. Recalcular *hash* dos registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



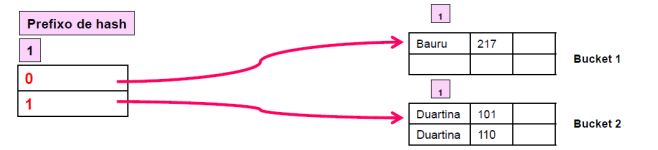
Estrutura após inserir 3 registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



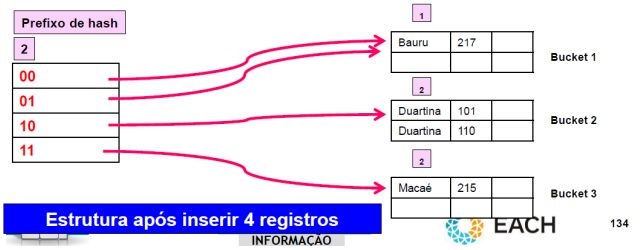
Não consigo inserir Macaé = Buffer cheio (1º bit = 1). Tem que aumentar número de bits e repetir o processo (dividir o último *bucket*).

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



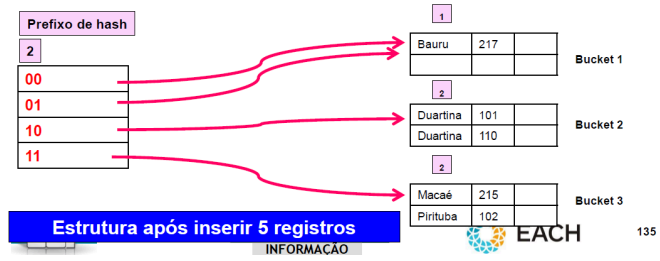
Estrutura após inserir 4 registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



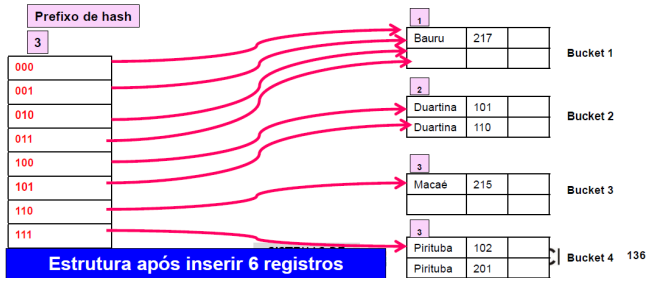
Estrutura após inserir 5 registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



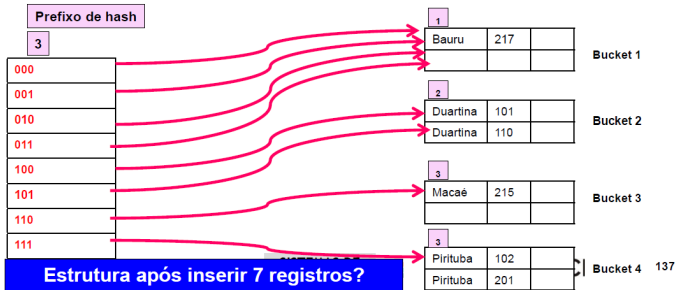
Estrutura após inserir 6 registros

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



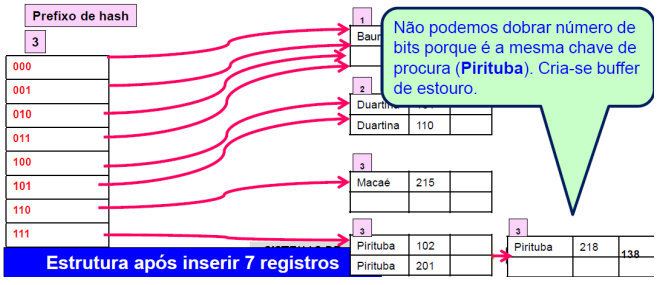
Estrutura após inserir 7 registros?

Hashing → Hashing dinâmico → Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



Estrutura após inserir 7 registros

Procura de um registro com chave de procura K_1

- pegar primeiros i bits de maior ordem de $h(K_1)$
- verifica-se entrada da tabela correspondente para a sequência de bits
- segue ponteiro de bucket na entrada da tabela

Inserção de um registro com chave de procura K_1

- mesmo procedimento da procura, parando no bucket j
  - se há espaço em j, insere registro no bucket

    - se j está cheio -> dividir j e redistribuir registros atuais, mais novo registro
        - Para dividir bucket -> determinar se precisa aumentar o número de bits utilizado
            - Incrementa o valor de i em 1 -> dobra tamanho da tabela de endereços de bucket
            - Cada entrada é substituída por duas, ambas com mesmo ponteiro da entrada original (bucket j)
            - Aloca novo bucket (bucket z)
            - Configura segunda entrada para apontar para novo bucket
            - Atribui i a i_j e i_z
            - Recalcula valor hash de cada registro no bucket j a fim de verificar se fica no bucket j ou z

Remoção de um registro com chave de procura K_1

- mesmo procedimento da procura, parando no bucket j
- remove chave de procura de j e o registro do arquivo
- remover j se ficar vazio
- se buckets forem fundidos -> tamanho da tabela de endereço de bucket pode ser cortado pela metade

Características e Vantagens

- **Acesso quase imediato:** O índice de hashing é especialmente eficiente para recuperações pontuais onde apenas um registro é buscado. Quando a função de hash é bem projetada e a tabela de hash não está sobrecarregada, a localização de um registro pode ser quase imediata *(Custo O(1) para leitura e escrita)*.
- **Hashing Dinâmico:** O hashing extensível (expansível) e o hashing linear são métodos para tratar colisões (quando dois registros têm o mesmo valor de hash) e para adaptar-se ao crescimento do banco de dados.
    - Vantagens:
        - desempenho não se degrada com crescimento do arquivo
        - overhead mínimo
        - tabela de endereços contém só um endereço para cada valor hash
        - nenhum bucket reservado para crescimento futuro: economia de espaço
    - Desvantagens:
        - procura envolvê nível de acesso indireto adicional
        - maior complexidade na implementação
- **Limitação para consultas de intervalo:** A principal desvantagem é que os índices de hash não são adequados para consultas de intervalo. Por exemplo, encontrar todos os registros entre dois valores seria ineficiente, pois não há ordem lógica no hashing como na B-Tree.
- **Limitação de memória:** Todas as chaves devem estar em memória principal.

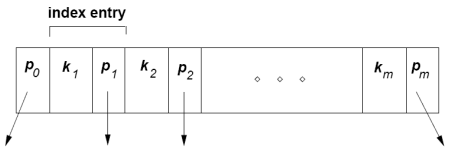
Índice em Árvore B

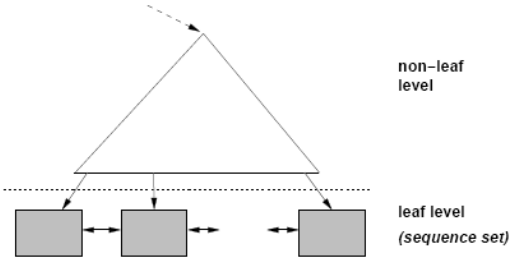
Árvores B+

Uma alternativa à indexação baseada em hash é organizar os registros usando uma estrutura de dados em árvore. As entradas de dados são organizadas de maneira ordenada pelo valor da chave de pesquisa e uma estrutura de dados de pesquisa hierárquica é mantida direcionando as pesquisas às páginas corretas das entradas de dados.

Noções básicas

- Inspiração em arquivo ordenado:
    - Motivação 1: reduzir tamanho da busca binária

    - Motivação 2: facilitar inserções e remoções
- Eficiência em busca de intervalo, varredura ordenada, inserção e remoção
- Eficiência em busca com igualdade, embora inferior a Hash
- Estrutura dinâmica derivada do ISAM com objetivo de:
    - Manter desempenho dependente somente da altura da árvore
    - Eliminar cadeias de overflow
    - Manter árvore balanceada sempre
    - Manter eficiência em insert/delete
    - Exceto o root, manter ocupação mínima do nó em 50%
- Formato dos **nós não-folha**
    - Nós não folha com Entradas de Índice, “IE-index entry”, do tipo : <Ki, Pi>
    - 

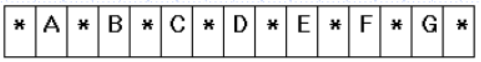
- Formato das **folhas**
    - Folhas com Entrada de Dados, “DE-data entries”, com três alternativas:
        - (1) Registro: **<..., k, ...>**
        - (2) Chave + identificador do registro: **<k, rid>**, onde o **rid=<#page_id, #slot_id>** identifica a página e o slot onde está localizado o registro no arquivo de dados
        - (3) Chave + lista de identificadores de registros: **<k, lista_rids>**
- Nós
    - Seja **d** a ordem da Árvore B+, então todo nó, exceto o root, terá **m chaves**, onde **d ≤ m ≤ 2d**
        - Já o root terá **1 ≤ m ≤ 2d**
        - O número de **ponteiros** no nó será **m+1**
    - Index Entry (ie) no nodo interno
        - **P0** aponta para subárvore com chaves **K < K1**
        - **Pm** aponta para subárvore com chaves **K ≥ Km**
        - **Pi|0<i<m** aponta para subárvores com chaves **Ki ≤ K< Ki+1**
    - Folhas
        - Entradas nas folhas seguem alternativas 1, 2, ou 3
        - Folhas com ponteiros para folhas adjacentes(next, previous), pois a alocação é dinâmica
        - 

Utilização de árvores B

- árvore de busca **balanceada**, onde ao se inserir uma chave ela é colocada sempre numa **folha**
- por meio de **sub-divisão (split)** e **promoção (promote)**, a árvore fica sempre **balanceada**

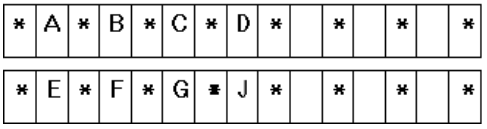
- Cada página (ou nó) é formada por um **sequência ordenada de chaves** e um **conjunto de ponteiros**.
- O **número de ponteiros** em um nó = **número de chaves + 1**.
- O **número máximo de ponteiros** que podem ser armazenados em um nó **é a ordem da árvore**.
    - EX: uma árvore-B de ordem 8 possui nós com, no máximo, 7 chaves e 8 filhos
- O **número máximo de ponteiros** é igual ao **número máximo de descendentes** de um nó.
- Os nós folha não possuem filhos, e seus ponteiros são **nulos**.

Seja a seguinte página inicial de uma árvore-B de **ordem 8**, que armazena **7 chaves**. Observe que, em uma situação real, além das chaves e ponteiros armazena-se outras informações associadas às chaves, como uma referência para a posição do registro associado à chave em um arquivo de dados. Esta **folha** que, coincidentemente, é também a **raiz da árvore**, está cheia. Como inserir uma nova chave, digamos J?



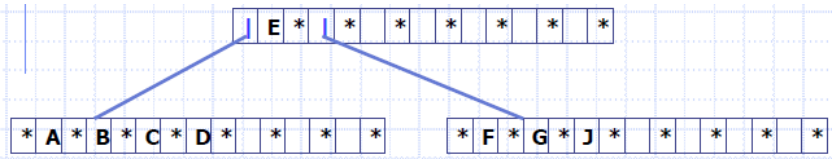
Subdivisão

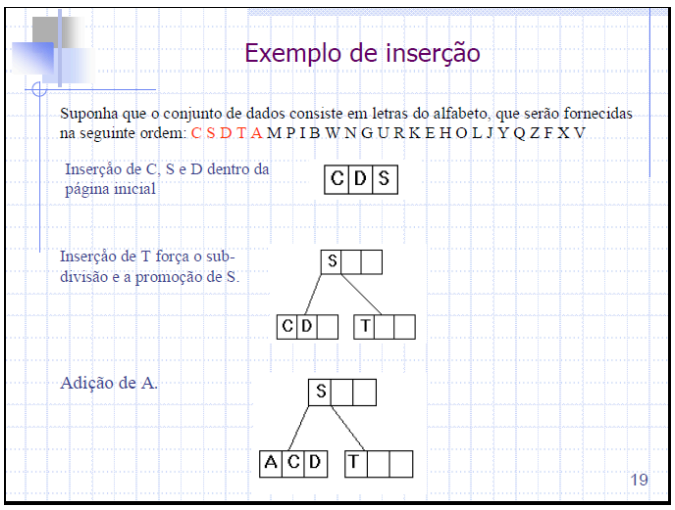
- Subdividimos (split) o nó folha em dois nós folhas, distribuindo as chaves igualmente entre os nós.
    - Temos agora duas folhas: precisamos criar uma nova raiz.
    - Fazemos isso "promovendo", ou "subindo", uma das chaves que estão nos limites de separação das folhas.

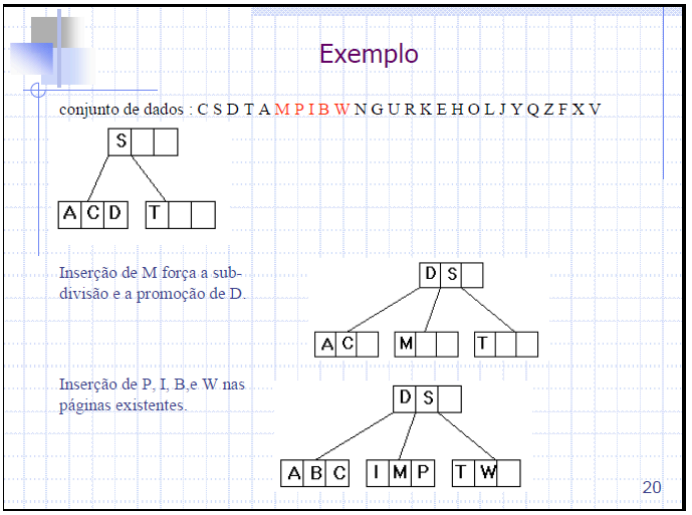


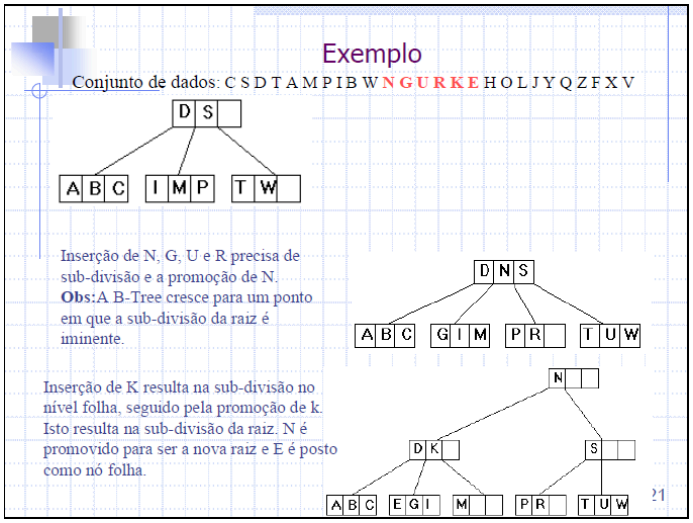
Promoção

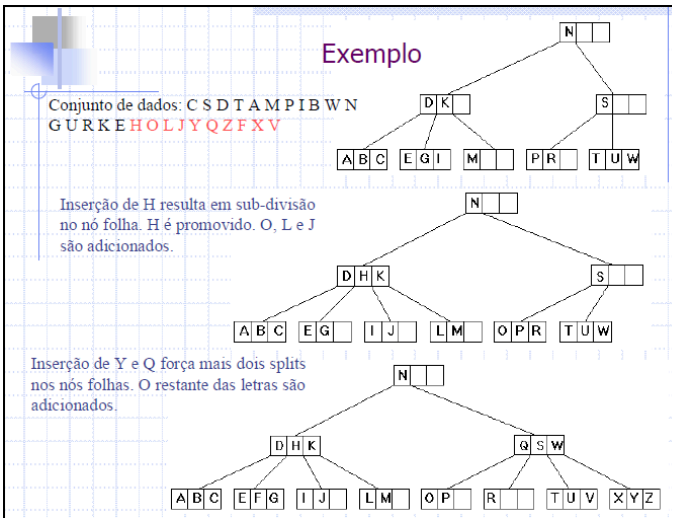
- Nesse caso, "promovemos" a chave E para a raiz:

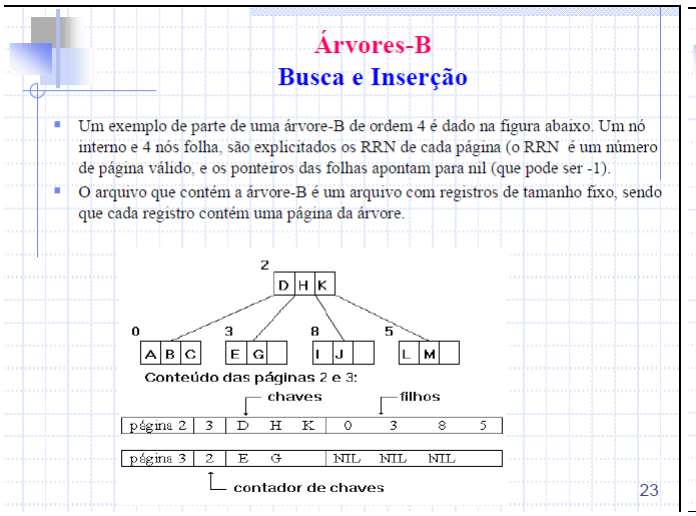


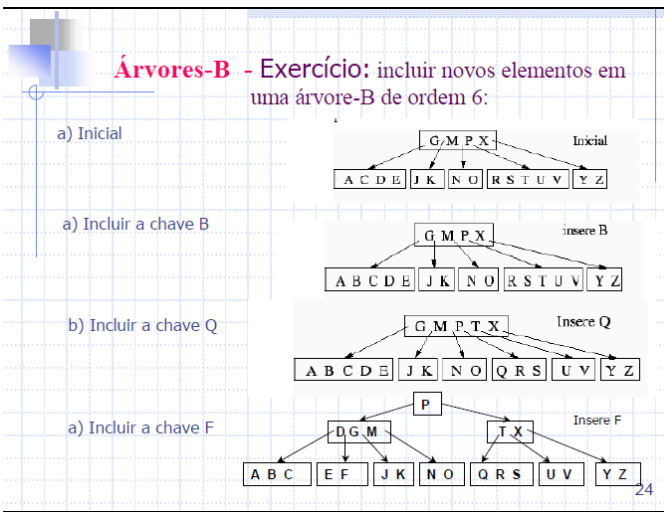












A Figura 8.3 mostra os registros de funcionários da Figura 8.2, desta vez organizados em um índice estruturado como árvore com chave de pesquisa idade. Cada nó nesta figura (por exemplo, nós rotulados como A, B, L1, L2) é uma página física, e a recuperação de um nó envolve uma E/S em disco.

O menor nível da árvore, chamado de **nível folha**, contém as entradas de dados; no nosso exemplo, são os registros dos funcionários. Para melhor ilustrarmos, desenhamos a Figura 8.3 como se houvesse registros adicionais de funcionários, alguns com idade menor que 22 e alguns com idade maior que 50 (o maior e o menor valor de idades que aparecem na Figura 8.2). Registros adicionais com idade menor que 22 apareceriam nas páginas folhas à esquerda da página L1 e registros com idade superior a 50 apareceriam em páginas folhas à direita da página L3

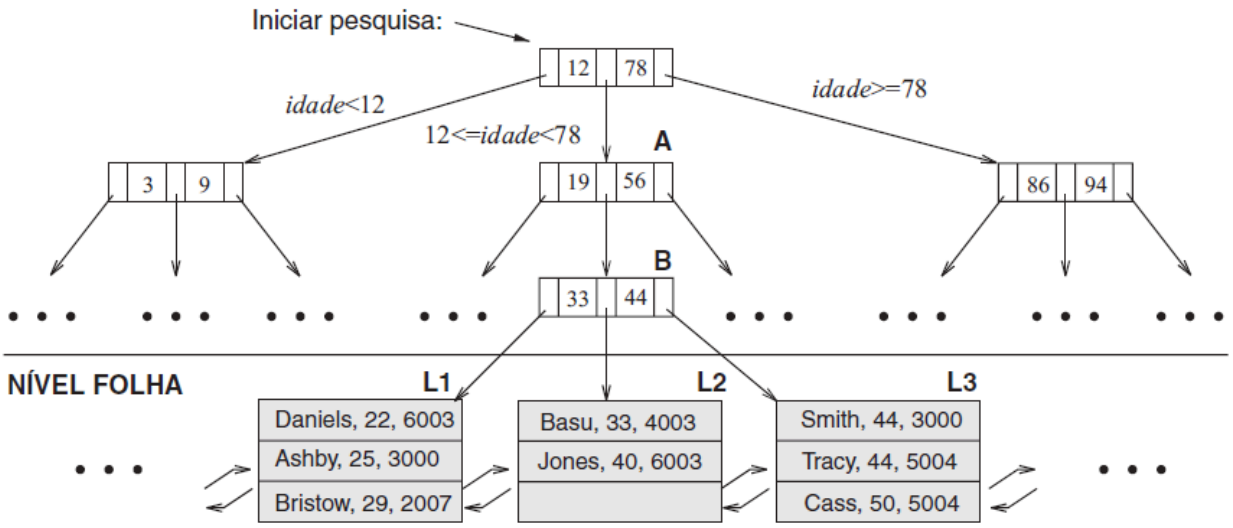


Figura 8.3 Índice estruturado como árvore.

Todas as pesquisas começam no nó mais acima, chamado **raiz**, e os conteúdos das páginas em níveis que não são folhas direcionam as pesquisas para a página folha correta. Páginas que não são folhas contêm ponteiros de nós separados por valores de chave de pesquisa. O ponteiro de nó à esquerda de um valor de chave k aponta para uma subárvore que contém apenas entradas de dados menores que k. O ponteiro de um nó à direita de um valor chave k aponta para uma subárvore que contenha apenas entradas de dados maiores ou iguais a k.

No nosso exemplo, suponha que queiramos encontrar todas as entradas de dados com 24 < idade < 50. Cada aresta do nó raiz para um nó filho na Figura 8.2 possui um rótulo que explica o que as subárvores correspondentes contêm. Na nossa pesquisa de exemplo, procuramos entradas de dados com valor de chave de pesquisa > 24 e somos direcionados para o filho do meio, o nó A. Novamente, examinando os conteúdos deste nó, somos direcionados para o nó B. Examinando os conteúdos do nó B, somos direcionados para o nó folha L1, o qual contém entradas de dados que estamos procurando.

Observe que os nós folhas L2 e L3 também contêm entradas de dados que satisfazem ao nosso critério de pesquisa. Para facilitar a recuperação de tais entradas qualificadas, todas as páginas folhas são mantidas em uma lista duplamente encadeada. Assim, podemos buscar a página L2 usando o ponteiro “próximo” na página L1 e então trazer a página L3 usando o ponteiro “próximo” em L2.

A árvore B+ é uma estrutura de índice que garante caminhos de raiz à folha de igual comprimento, proporcionando uma busca eficiente. Ela mantém a altura da árvore baixa, geralmente três ou quatro, resultando em rápida localização de páginas folha. Cada nó não folha pode conter muitos ponteiros, minimizando o número de operações de entrada/saída (E/S). O **fan-out** da árvore, ou seja, o número médio de filhos de um nó não folha, determina o número total de páginas folha. Com um fan-out de pelo menos 100, uma árvore de altura quatro pode conter até 100 milhões de páginas folha que podem ser encontradas com quatro E/S.

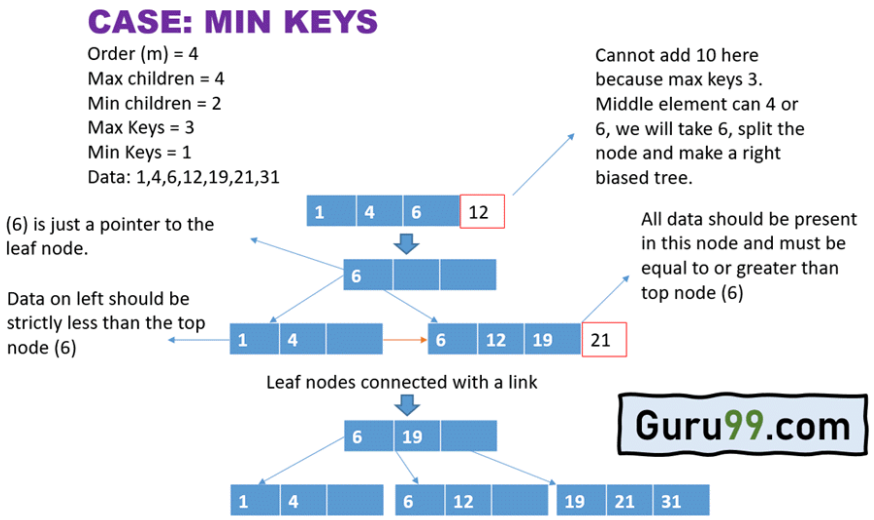
A inserção, por sua vez, pode requerer a divisão do nó caso ele esteja cheio. Se o nó-alvo da inserção tem espaço, a chave é diretamente inserida. Caso contrário, o nó é dividido para acomodar a nova chave e manter o balanceamento da árvore. Já a remoção de uma chave é mais direta se ela estiver em um nó folha. Entretanto, quando a chave a ser removida está em um nó interno, diversos cenários precisam ser considerados para assegurar que a árvore permaneça balanceada após a operação.

Operações em Árvore B+

Inserção

O seguinte algoritmo é aplicável para a inserção operação:

- 50% dos elementos nos nós são movidos para uma nova folha para armazenamento.
- O pai da nova Folha está vinculado precisamente ao valor mínimo da chave e a um novo local na árvore.
- Divida o nó pai em mais locais, caso ele seja totalmente utilizado.
- Agora, para melhores resultados, a chave central está associada ao nó de nível superior dessa Folha.
- Até que o nó de nível superior não seja encontrado, continue iterando o processo explicado nas etapas acima.



O exemplo de amostra da árvore B+ acima é explicado nas etapas abaixo:

- Primeiramente, temos 3 nós, e os primeiros 3 elementos, que são 1, 4 e 6, são adicionados em locais apropriados nos nós.
- O próximo valor na série de dados é 12, que precisa fazer parte da árvore.
- Para conseguir isso, divida o nó e adicione 6 como elemento ponteiro.
- Agora, uma hierarquia à direita de uma árvore é criada e os valores de dados restantes são ajustados de acordo, tendo em mente as regras aplicáveis de valores iguais ou maiores em relação aos nós de valor-chave à direita.

Pesquisar

Na Árvore B+, a pesquisa é um dos procedimentos mais fáceis de executar e obter resultados rápidos e precisos.
O seguinte algoritmo de pesquisa é aplicável:

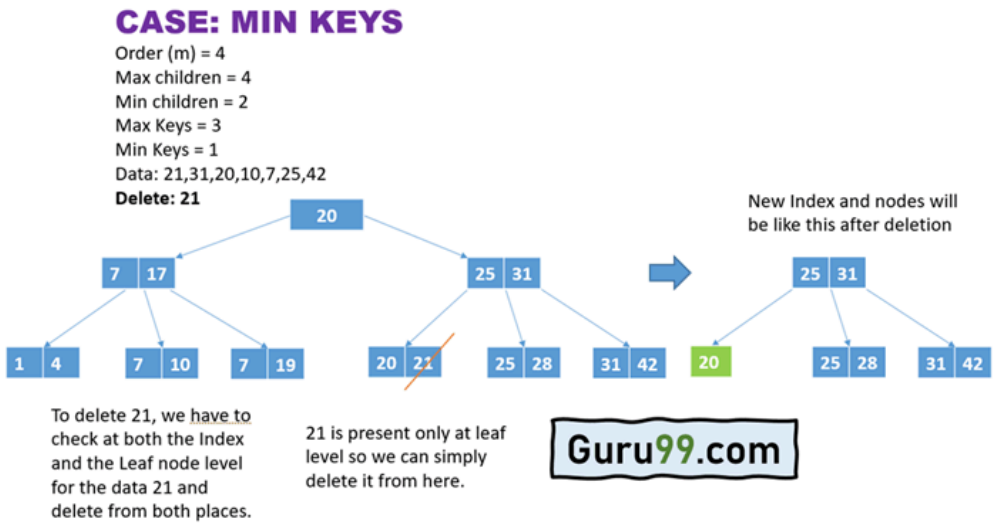
- Para encontrar o registro necessário, você precisa executar o busca binária nos registros disponíveis na árvore.
- No caso de correspondência exata com a chave de pesquisa, o registro correspondente é retornado ao usuário.

- Caso a chave exata não seja localizada pela pesquisa no nó pai, atual ou folha, uma “mensagem não encontrada” será exibida para o usuário.

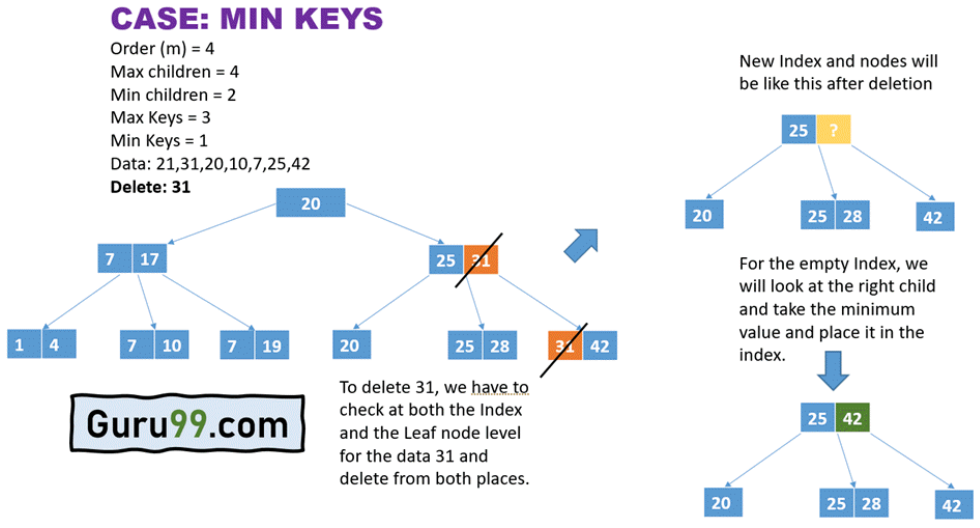
Apagar

O seguinte algoritmo é aplicável ao excluir um elemento da árvore B+:

- Primeiramente, precisamos localizar uma entrada folha na árvore que contém a chave e o ponteiro.
    - Exclua a entrada da folha da árvore se a Folha atender às condições exatas de exclusão do registro.
- Caso o nó folha atenda apenas ao fator satisfatório de estar meio cheio, então a operação está concluída; Caso contrário, o nó Folha possui entradas mínimas e não pode ser excluído.
- Os outros nós vinculados à direita e à esquerda podem desocupar quaisquer entradas e movê-las para a Folha. Se esses critérios não forem atendidos, eles deverão combinar o nó folha e seu nó vinculado à hierarquia da árvore.
- Após a fusão do nó folha com seus vizinhos à direita ou à esquerda, as entradas de valores do nó folha ou vizinho vinculado apontando para o nó de nível superior são excluídas.



O exemplo acima ilustra o procedimento para remover um elemento da Árvore B+ de uma ordem específica.



- Primeiramente, as localizações exatas do elemento a ser excluído são identificadas na árvore.
- Aqui, o elemento a ser excluído só pode ser identificado com precisão no nível folha e não na colocação do índice. Consequentemente, o elemento pode ser excluído sem afetar as regras de exclusão, que é o valor da chave mínima.
- No exemplo acima, temos que deletar 31 da árvore.
- Precisamos localizar as instâncias de 31 em Index e Leaf.
- Podemos ver que 31 está disponível nos níveis de nó Índice e Folha. Portanto, nós o excluímos de ambas as instâncias.
- Mas temos que preencher o índice apontando para 42. Vamos agora olhar para o filho certo com menos de 25 anos e pegar o valor mínimo e colocá-lo como índice. Então, sendo 42 o único valor presente, ele se tornará o índice.

Desempenho

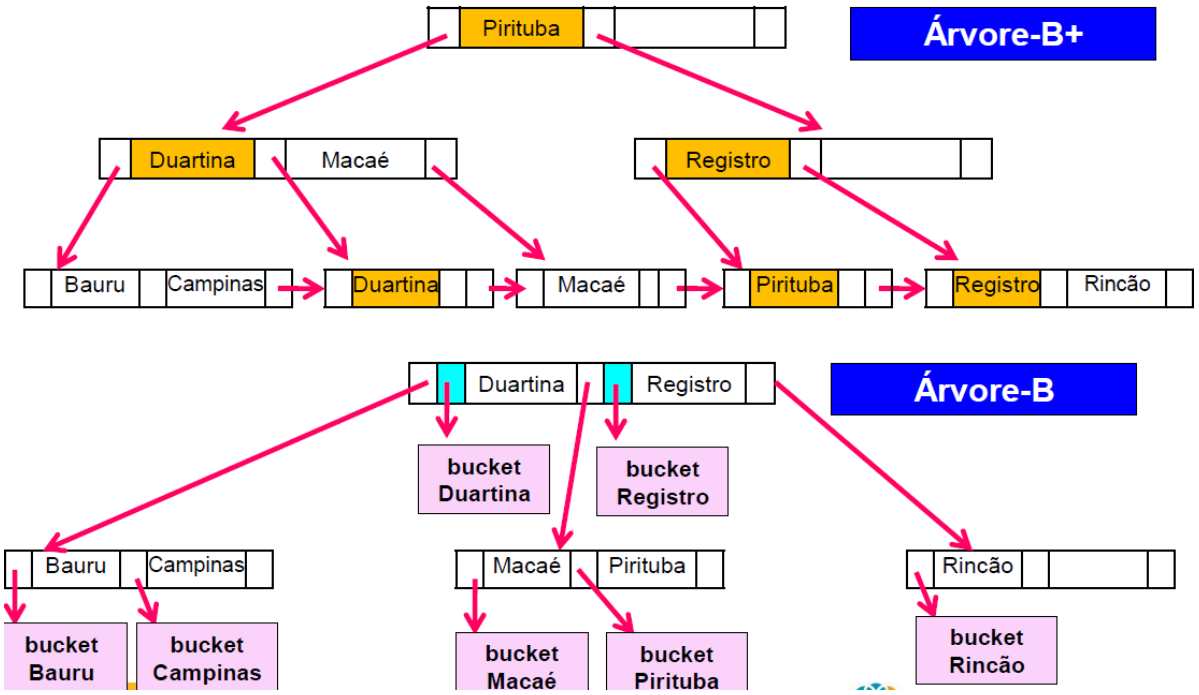
- Faz sentido ignorar o custo das operações na memória rápida e contar apenas o número de sondagens.
- Uma **sondagem** é o primeiro acesso a uma página durante uma busca ou inserção.
- *Proposição*.
    - Uma busca ou inserção em uma árvore B de ordem M com N chaves envolve entre $\log_M N$ e $\log_{M/2} N$ sondagens.
- *Prova*:
    - O número de sondagens é igual à altura da árvore. No melhor caso, todas as páginas internas têm M-1 filhos. No pior caso, todas as páginas internas (exceto a raiz) têm M/2 filhos.
    - Se M é da ordem de 1000, por exemplo, a altura da árvore não passa de 4 se N for menor que 62 bilhões.

Comparação entre Árvore B e B+

[B-tree vs B+ tree in Database Systems]

As árvores B e B+ são estruturas de dados comuns usadas em bancos de dados e sistemas de armazenamento para indexação eficiente. Embora ambas sejam semelhantes em muitos aspectos, existem algumas diferenças significativas entre elas:

Árvore B+	Árvore B
As chaves de pesquisa podem ser repetidas.	As chaves de pesquisa não podem ser redundantes.
Os dados são salvos apenas nos nós folha.	Tanto os nós folha quanto os nós internos podem armazenar dados
Os dados armazenados no nó folha tornam a pesquisa mais precisa e rápida.	Searching é lento devido aos dados armazenados na Folha e nos nós internos.
A exclusão não é difícil porque um elemento só é removido de um nó folha.	A exclusão de elementos é um processo complicado e demorado.
Os nós folha vinculados tornam a pesquisa eficiente e rápida.	Você não pode vincular nós folha.



Características e Vantagens

- **Balanceamento e Estrutura Flexível:** A Árvore B é intrinsecamente balanceada, garantindo que todos os caminhos da raiz até qualquer folha tenham o mesmo comprimento. Além disso, ao contrário das árvores binárias, os nós podem ter um número variável de filhos, proporcionando uma estrutura adaptável.
- **Eficiência em Operações:** A combinação do balanceamento com o alto fator de ramificação resulta em uma altura de árvore geralmente reduzida, mesmo com muitos elementos. Isso permite buscas (log n), inserções e remoções eficientes.
- **Busca por intervalos:** A Árvore B permite a busca em intervalos com eficiência.
- **Índices mantidos em disco:** permite indexação de grandes bases de dados.

O índice de árvore B é a estrutura de dados amplamente utilizada para indexação baseada em árvores em SGBD. É um formato multinível de indexação baseada em árvore que possui balanceamento. Todos os nós folha da árvore B significam ponteiros de dados reais. Além disso, todos os nós folha são interligados por uma lista de links, o que permite que uma árvore B suporte acesso aleatório e sequencial.

B-Tree Visualization
B+ Tree Visualization

SQL e Índices

O SQL padrão não provê que usuário ou DBA controle formas de construir índices. SGBD pode decidir automaticamente quais índices criar. Como isso não é fácil, é permitido que usuário crie e remova índices:

Criar um índice:

- create [unique] index <nome_indice> on <nome_relação> (<lista de atributos>)

Exemplo:

- create index iconta on conta (num_conta)

Remover um índice:

- drop index <nome_indice>

Exemplo:

- drop index iconta

Referências:

- Fundamentos de Sistemas de Gerenciamento de Banco de Dados - Elmasri & Navathe (6ª edição)
- Sistemas de Gerenciamento Sistemas de Gerenciamento de Banco de Dados - Ramakrishnan Gehrke (3ª edição)
- https://edisciplinas.usp.br/pluginfile.php/7535382/mod_resource/content/3/ACH2025-Aula08-IndexacaoEHashingParte1.pdf
- https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/B-trees.html
- https://www.facom.ufu.br/~ilmerio/gbd2/gbd2_s3_arvores.pdf
- https://www.guru99.com/pt/introduction-b-plus-tree.html
- https://edisciplinas.usp.br/pluginfile.php/7704750/mod_resource/content/1/ACH2025-Aula09-IndexacaoEHashing-Parte2.pdf