

Indexação em Bancos de Dados

Estruturas de dados fundamentais para otimização de consultas e desempenho em sistemas de gerenciamento de bancos de dados relacionais

Eduardo Ogasawara

eduardo.ogasawara@cefet-rj.br
<https://eic.cefet-rj.br/~eogasawara>

Conceitos Básicos de Indexação

Os mecanismos de indexação são estruturas auxiliares projetadas para acelerar significativamente o acesso aos dados desejados em um banco de dados. Funcionam de maneira análoga a um catálogo de autores em uma biblioteca, onde você pode localizar rapidamente um livro sem precisar examinar todas as prateleiras.

Chave de busca: atributo ou conjunto de atributos utilizados para pesquisar e localizar registros específicos em um arquivo de dados

Estrutura do Índice

Um arquivo de índice é composto por registros chamados **entradas de índice**, cada uma contendo uma chave de busca e um ponteiro para o registro correspondente

Eficiência

Arquivos de índice são tipicamente muito menores que o arquivo de dados original, permitindo carregamento rápido em memória

Índices Ordenados

Chaves de busca armazenadas em ordem classificada, permitindo buscas eficientes por valores específicos ou intervalos

Índices de Hash

Chaves de busca distribuídas uniformemente entre "baldes" usando uma função matemática de dispersão

Arquivo Organizado vs Estrutura de Índice

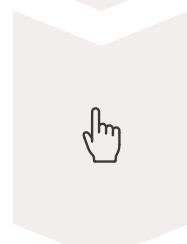
É fundamental distinguir entre a organização física dos dados e as estruturas de acesso auxiliares. Esta distinção permite flexibilidade arquitetural e otimização independente de cada camada.



Arquivo Organizado



Define como os registros são **armazenados fisicamente** no disco. Pode ser organizado sequencialmente (ordenado por chave) ou por hash (distribuído por função)



Estrutura de Índice



É uma **estrutura auxiliar** que mantém ponteiros para registros do arquivo principal. Existe independentemente da organização física escolhida

- ❑ **Observação importante:** Um arquivo organizado por hash não necessita de um índice primário adicional, pois a própria função hash já fornece acesso direto aos registros. No entanto, índices secundários podem ser úteis para buscas por atributos não-hash.

Métricas de Avaliação de Índices

A escolha da estrutura de indexação adequada depende de diversos fatores de desempenho que devem ser cuidadosamente analisados conforme o padrão de uso do sistema.

1

Tipos de Acesso Admitidos

- Busca por valor exato de atributo
- Busca por intervalo de valores
- Suporte a consultas complexas

2

Tempo de Acesso

Latência para localizar e recuperar registros específicos. Inclui tempo de busca no índice e acesso ao arquivo de dados

3

Tempo de Inserção

Overhead para adicionar novos registros, incluindo atualização das estruturas de índice e possível reorganização

4

Tempo de Exclusão

Custo para remover registros e manter a consistência das estruturas de indexação

5

Sobrecarga de Espaço

Espaço adicional em disco necessário para armazenar as estruturas de índice além dos dados principais

Análise de Complexidade Assintótica

As estruturas de indexação apresentam características distintas de desempenho que podem ser analisadas através de notação assintótica, auxiliando na escolha da estrutura mais adequada.

Índices Ordenados (Árvore B+)

Tempo de busca: Proporcional à [altura da árvore](#)

Crescimento **logarítmico** em relação ao tamanho do arquivo: $O(\log n)$

Operações de modificação: Inserções e exclusões têm custo logarítmico devido à necessidade de percorrer a árvore

Hashing

Tempo de busca: Constante esperado $O(1)$ no caso médio

Não garante tempo constante no pior caso (colisões podem degradar performance)

Operações de modificação: Custo constante esperado, mas pode exigir rehashing completo em estruturas dinâmicas

- A escolha entre B+ e hashing depende do perfil de consultas: B+ é superior para buscas por intervalo, enquanto hashing excelle em buscas por chave exata.

Índices Ordenados

Em índices ordenados, as entradas são mantidas classificadas pelo valor da chave de busca, similar a um catálogo de autores em biblioteca. Esta ordenação permite buscas eficientes e suporta operações por intervalo.



Índice Primário

Aplicado a arquivos ordenados sequencialmente. A chave de busca define a ordem sequencial do arquivo. Também chamado de **índice de agrupamento** (clustering index)



Arquivo Sequencial Indexado

Combinação de um arquivo ordenado sequencialmente com um índice primário, oferecendo acesso rápido e varredura eficiente



Índice Secundário

Índice cuja chave de busca especifica uma ordem diferente da ordem sequencial do arquivo. Também chamado **índice não de agrupamento**

A chave de busca de um índice primário é normalmente a chave primária da relação, mas não é um requisito obrigatório

Arquivos de Índice Denso

Um índice denso contém uma entrada de índice para **cada valor** de chave de busca presente no arquivo de dados. Esta abordagem oferece acesso direto extremamente rápido, mas requer mais espaço de armazenamento.

Características

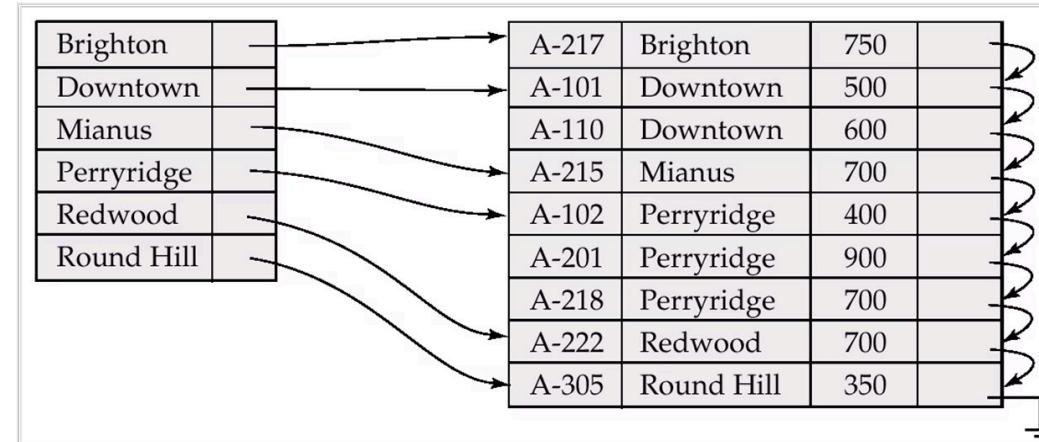
- Entrada de índice para cada registro do arquivo
- Acesso direto sem busca sequencial
- Maior consumo de espaço

Vantagens

- Velocidade máxima de busca
- Não depende da ordenação do arquivo
- Ideal para chaves secundárias

Desvantagens

- Alto custo de armazenamento
- Maior overhead em atualizações
- Pode não caber em memória



Arquivos de Índice Esparsos

Um índice esparsos contém entradas apenas para **alguns valores** de chave de busca, reduzindo significativamente o espaço necessário. Esta técnica só é aplicável quando os registros estão ordenados sequencialmente pela chave de busca.

Conceito

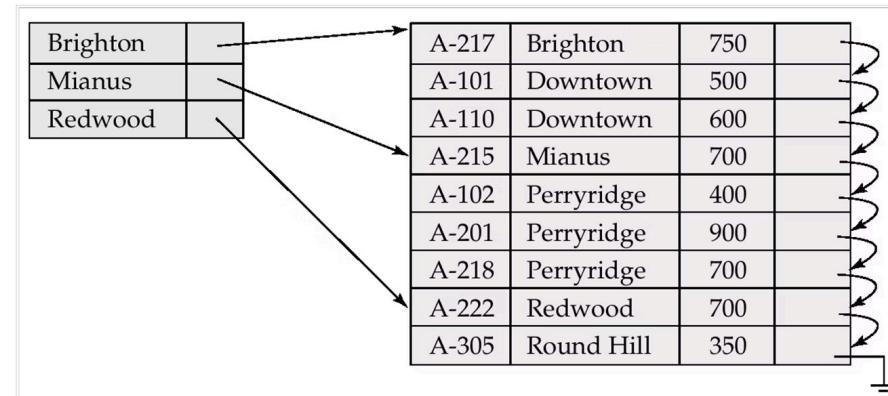
O índice aponta para o primeiro registro de cada bloco ou grupo de registros. Para localizar um registro específico, o sistema:

1. Encontra a entrada de índice apropriada
2. Segue o ponteiro até o bloco
3. Busca sequencialmente dentro do bloco

Requisitos

Aplicável **somente** quando:

- Os registros estão fisicamente ordenados por chave de busca
- A chave de busca é a mesma usada na organização do arquivo
- Não há duplicatas entre blocos diferentes



Características de Índices Esparsos

Eficiência de Espaço

Requer significativamente **menos espaço** de armazenamento comparado a índices densos, pois mantém apenas uma fração das entradas

Manutenção Simplificada

Menor sobrecarga de manutenção para operações de **inserção e exclusão**, já que menos entradas de índice precisam ser atualizadas

Desempenho de Busca

Geralmente **mais lento** que índices densos para localizar registros individuais, pois pode requerer busca sequencial dentro de blocos

Prática recomendada: Uma configuração eficiente é manter um índice esparso com uma entrada para cada bloco de disco, correspondendo ao menor valor de chave de busca presente naquele bloco.

Esta abordagem balanceia eficientemente o uso de espaço com a desempenho de acesso, minimizando o número de acessos ao disco enquanto mantém o índice compacto.

Índices Multiníveis

Quando um índice se torna muito grande para caber completamente na memória principal, o acesso torna-se custoso devido a múltiplas operações de E/S. A solução é criar uma hierarquia de índices.

Problema	Solução	Hierarquia	Expansão
Índice primário em disco não cabe em memória, gerando muitos acessos ao disco	Criar índice esparsão sobre o índice primário, tratando-o como arquivo sequencial	Índice externo aponta para índice interno, que aponta para arquivo de dados	Se necessário, criar níveis adicionais até que o nível superior caiba em memória

- Manutenção crítica:** Todos os níveis de índice devem ser atualizados consistentemente durante operações de inserção ou exclusão no arquivo, o que aumenta o custo de modificações.

Exemplo Visual de Índice Multinível

A estrutura hierárquica permite reduzir drasticamente o número de acessos ao disco necessários para localizar um registro, navegando progressivamente dos níveis superiores (mais esparsos) até os níveis inferiores (mais densos).

Nível Superior

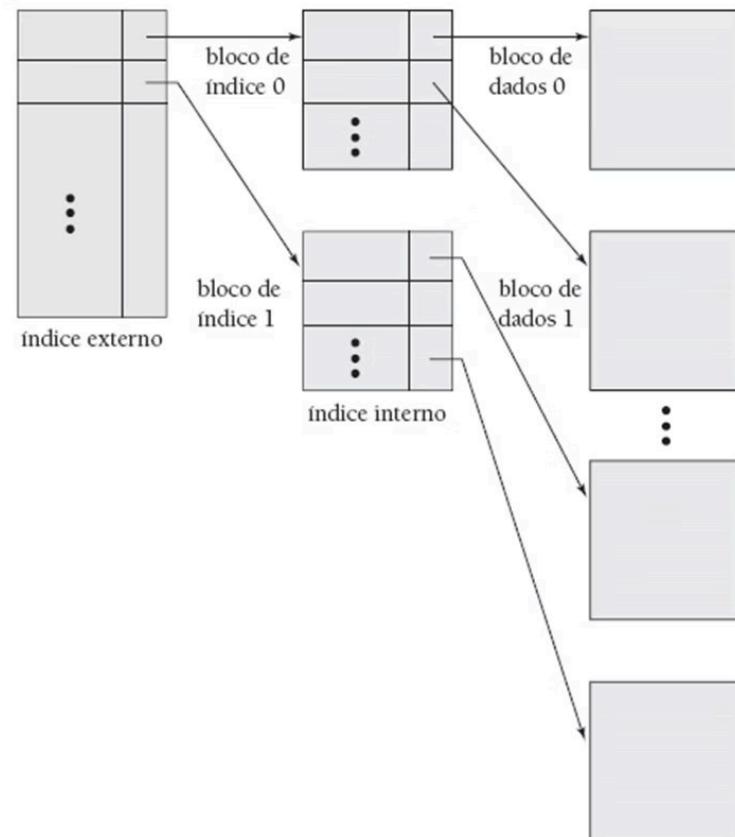
Índice mais esparsos, permanece em memória. Aponta para blocos do nível intermediário

Nível Intermediário

Índice de granularidade média. Pode haver múltiplos níveis intermediários se necessário

Nível Inferior

Índice primário denso ou esparsos que aponta diretamente para os registros de dados



Índices Secundários

Frequentemente precisamos localizar registros baseados em atributos que não são a chave de busca do índice primário. Índices secundários fornecem caminhos alternativos de acesso aos dados.

Exemplo prático: Se contas bancárias são armazenadas sequencialmente por agência e número de conta, mas queremos encontrar todas as contas com saldo superior a R\$ 10.000, precisamos de um índice secundário no atributo saldo.

1 Estrutura com Baldes

Cada entrada do índice secundário aponta para um **balde** (bucket) que contém ponteiros para todos os registros com aquele valor específico de chave de busca

2 Necessidade de Densidade

Diferentemente de índices primários, índices secundários **devem ser densos**, pois os registros não estão fisicamente ordenados por este atributo

Exemplo: Índice Secundário em Saldo

Visualização de como um índice secundário organiza o acesso a registros através de um atributo não-ordenante, usando baldes para agrupar ponteiros de registros com o mesmo valor de chave.

No diagrama acima, observe que múltiplas contas podem compartilhar o mesmo saldo, exigindo que a entrada de índice aponte para um conjunto de ponteiros ao invés de um único registro.

Chave de Índice

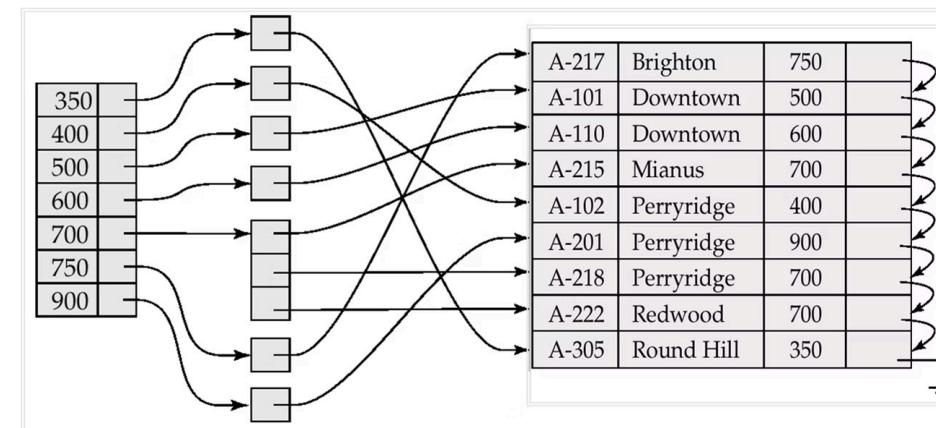
Valor do atributo saldo (ex: 500, 750, 1000)

Balde

Lista de ponteiros para todos os registros com aquele saldo

Registros

Dados completos das contas no arquivo principal



Trade-offs de Índices Primários e Secundários

A decisão de criar e manter índices envolve analisar cuidadosamente o equilíbrio entre benefícios de consulta e custos de manutenção.

Benefícios

Velocidade de Busca

Índices oferecem **benefícios substanciais** ao buscar registros específicos ou intervalos

Varredura Eficiente

Varredura sequencial usando índice primário é altamente eficiente devido à localidade física

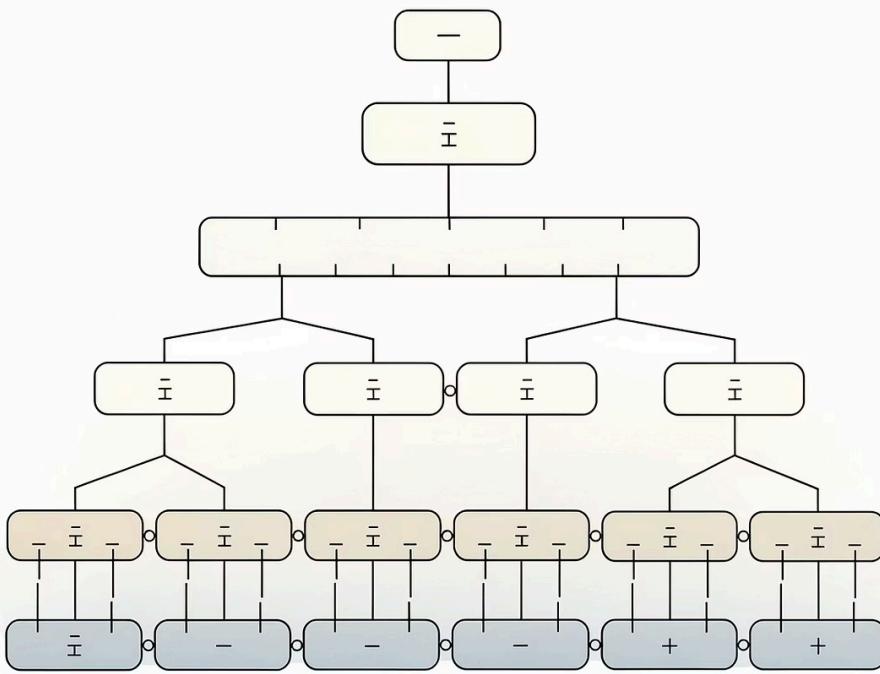
Custos

Overhead de Atualização

Cada índice deve ser **atualizado** em modificações, impondo sobrecarga significativa

Varredura Secundária

Varredura sequencial via índice secundário é **dispendiosa** - cada acesso pode requerer novo bloco de disco



Árvore B+

Estrutura de indexação mais popular em sistemas de bancos de dados modernos, combinando eficiência de busca com manutenção dinâmica equilibrada.

Arquivos de Índice de Árvore B+

Árvores B+ são estruturas de dados sofisticadas especialmente projetadas para sistemas de armazenamento que realizam leitura/escrita em blocos, como discos magnéticos e SSDs.



Auto-reorganização

Reorganizam-se automaticamente com **pequenas mudanças locais** em face a inserções e exclusões, mantendo o balanceamento



Performance Consistente

Não requerem reorganização do arquivo inteiro para manter desempenho ótimo, evitando operações custosas



Altura Balanceada

Mantêm todas as folhas no mesmo nível, garantindo tempo de busca **$O(\log n)$** previsível e consistente

- ❑ A árvore B+ é uma evolução da árvore B, otimizada para sistemas de banco de dados ao concentrar todos os dados nas folhas e manter os nós internos apenas como índice de navegação.

Propriedades Fundamentais da Árvore B+

A estrutura da árvore B+ é governada por propriedades rigorosas que garantem balanceamento e eficiência. Estas propriedades são mantidas através de algoritmos sofisticados de inserção e exclusão.

01

Balanceamento

Todos os caminhos da raiz até as folhas têm o mesmo comprimento, garantindo busca uniforme

02

Ocupação Mínima

Cada nó (exceto raiz) está preenchido entre 50% e 100% da capacidade, otimizando espaço

03

Ordem da Árvore

Parâmetro n define capacidade máxima: nós internos têm $n-1$ chaves e n ponteiros

04

Dados nas Folhas

Todos os valores de chave e ponteiros para registros residem apenas nos nós folha

05

Lista Encadeada

Nós folha são ligados sequencialmente, permitindo varredura eficiente por intervalo

Estrutura de Nós da Árvore B+

Os nós da árvore B+ possuem uma estrutura interna cuidadosamente projetada para otimizar operações de busca e permitir splits/merges eficientes durante atualizações.

Nós Internos

- Contêm **Ki** valores de chave usados para navegação
- Contêm **Pi** ponteiros para nós filhos
- Número de ponteiros = número de chaves + 1
- Guiam a busca mas não armazenam dados

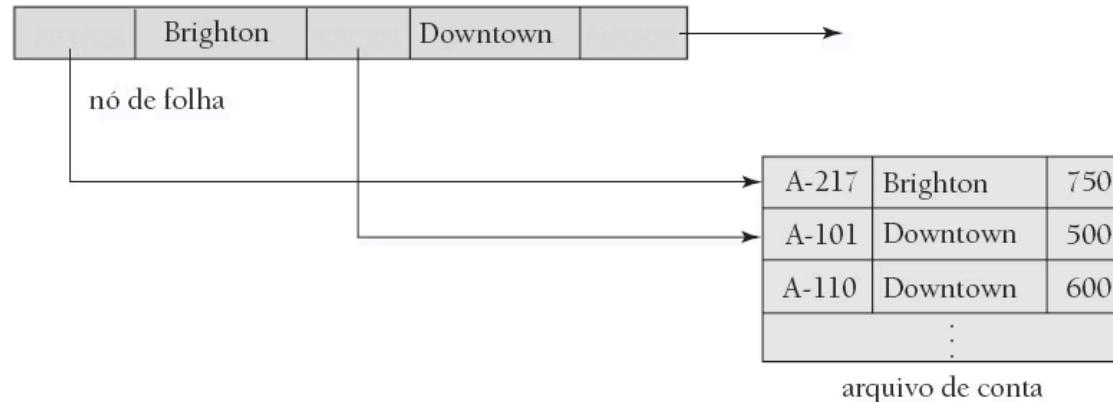
Nós Folha

- Armazenam todos os valores de **chave de busca**
- Mantêm **ponteiros para registros** de dados
- Ligados sequencialmente (lista encadeada)
- Última posição: ponteiro para próxima folha

A separação entre nós de índice e nós de dados permite que a árvore mantenha um fanout alto (muitos filhos por nó), reduzindo a altura da árvore e consequentemente o número de acessos ao disco.

Anatomia dos Nós Folha

Os nós folha da árvore B+ têm estrutura especializada que suporta tanto acesso direto por chave quanto varredura sequencial eficiente, sendo fundamentais para a versatilidade da estrutura.



Pares Chave-Ponteiro

Cada entrada contém um valor de chave de busca **K_i** emparelhado com um ponteiro **P_i** que referencia o registro de dados correspondente no arquivo principal

Ordenação

As chaves dentro de cada nó folha são mantidas em [ordem crescente](#), facilitando buscas binárias e operações de intervalo

Encadeamento

O último ponteiro **P_n** não aponta para dados, mas para o próximo nó folha na sequência, permitindo varredura sequencial sem retornar à raiz

Esta organização permite que consultas por intervalo (range queries) sejam executadas eficientemente: após localizar a primeira chave do intervalo, basta seguir os ponteiros de encadeamento através das folhas adjacentes.

Nós Não-Folha nas Árvores B+

As árvores B+ são estruturas de dados fundamentais em sistemas de gerenciamento de banco de dados. Os nós não-folha desempenham um papel crucial como camadas intermediárias de indexação, guiando as operações de busca até os dados reais armazenados nas folhas.

Em um nó não-folha, a organização dos ponteiros é essencial para direcionar as buscas. Para um nó com n ponteiros, as chaves de busca K nos sub-árvores referenciadas por cada ponteiro P_i seguem as seguintes regras:

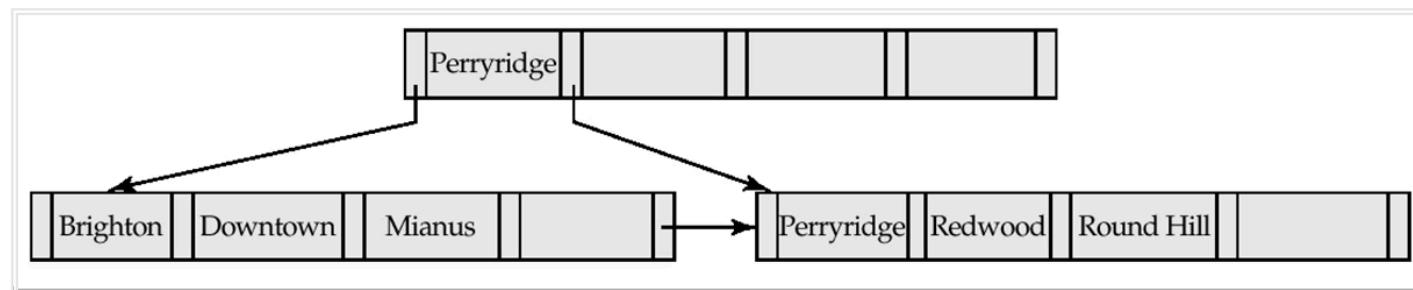
- Para o ponteiro P_i (onde $i < n - 1$), todas as chaves de busca K no sub-árvore correspondente têm valores maiores ou iguais a K_{i-1} e menores que K_i .
- Para o último ponteiro P_n , todas as chaves de busca K no sub-árvore correspondente têm valores maiores ou iguais a K_{n-1} .

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

Exemplo de Árvore B+

Visualizar a estrutura de uma árvore B+ ajuda a compreender como os dados são organizados hierarquicamente. Esta representação mostra a distribuição equilibrada de chaves entre os nós, garantindo eficiência nas operações de busca, inserção e exclusão.

Observe como os valores são distribuídos: os nós internos contêm chaves que servem como guias de navegação, enquanto os nós-folha armazenam os dados efetivos ou ponteiros para os registros.



Observações Sobre Árvores B+

ESTRUTURAS DE DADOS

Organização Física vs. Lógica

As conexões entre nós são implementadas através de ponteiros, permitindo que blocos logicamente adjacentes não precisem estar fisicamente próximos no disco. Isso oferece grande flexibilidade na alocação de espaço.

Hierarquia de Índices

Os níveis não-folha formam uma hierarquia de índices esparsos, criando uma estrutura em camadas que acelera significativamente as operações de busca comparado a uma varredura sequencial.

Eficiência Logarítmica

A árvore B+ mantém uma quantidade relativamente pequena de níveis, proporcional ao logaritmo do tamanho do arquivo. Isso garante que buscas sejam executadas rapidamente, mesmo em grandes volumes de dados.

Operações Dinâmicas

Inserções e exclusões são tratadas eficientemente através da reestruturação da árvore em tempo logarítmico, mantendo o balanceamento e preservando as propriedades que garantem o desempenho.

Consultas em Árvores B+

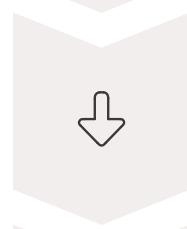
Processamento de Buscas



Início na Raiz



A consulta começa no nó raiz, comparando a chave de busca com os valores armazenados para determinar qual sub-árvore explorar.



Navegação Descendente



O processo desce pela árvore, seguindo ponteiros baseados em comparações de chaves, até alcançar o nível das folhas.



Localização dos Dados

Ao atingir uma folha, os dados desejados são encontrados ou conclui-se que não existem na estrutura.

O processo de consulta em árvores B+ é extremamente eficiente devido à altura logarítmica da árvore. Para buscas por intervalo, as folhas encadeadas permitem varredura sequencial após localizar o ponto inicial.

Tipos de Consultas em Árvores B+

Busca por Chave Única

Consultas que procuram um valor específico navegam pela árvore comparando a chave de busca com os valores nos nós internos. A complexidade é $O(\log n)$, onde n é o número total de registros.

Este tipo de busca é otimizado pela estrutura balanceada da árvore, garantindo que todas as folhas estejam aproximadamente à mesma distância da raiz.

Busca por Intervalo

Para consultas que buscam valores em um intervalo, a árvore B+ oferece vantagens adicionais. Após localizar o primeiro valor do intervalo, as folhas encadeadas permitem acesso sequencial eficiente.

Esta propriedade torna as árvores B+ especialmente adequadas para consultas SQL com cláusulas BETWEEN ou operadores de comparação como $>$ e $<$.

- ❑ **Vantagem Chave:** A combinação de busca logarítmica com acesso sequencial nas folhas oferece desempenho superior para diversos tipos de consultas.

Atualizações em Árvores B+: Inserção

O processo de inserção em árvores B+ segue um algoritmo estruturado que garante o balanceamento da árvore e a manutenção de suas propriedades fundamentais.



Localização

Encontre o nó folha em que o valor da chave de busca apareceria, navegando pela árvore a partir da raiz



Verificação de Existência

Se o valor da chave de busca já estiver presente no nó folha, o registro é acrescentado ao arquivo e, se necessário, um ponteiro é inserido no balde correspondente



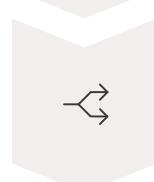
Nova Entrada

Se o valor da chave de busca não estiver presente, acrescente o registro ao arquivo principal e crie um balde, se necessário



Inserção no Nó

Se houver espaço disponível no nó folha, insira o par (valor de chave, ponteiro) diretamente no nó folha



Divisão de Nó

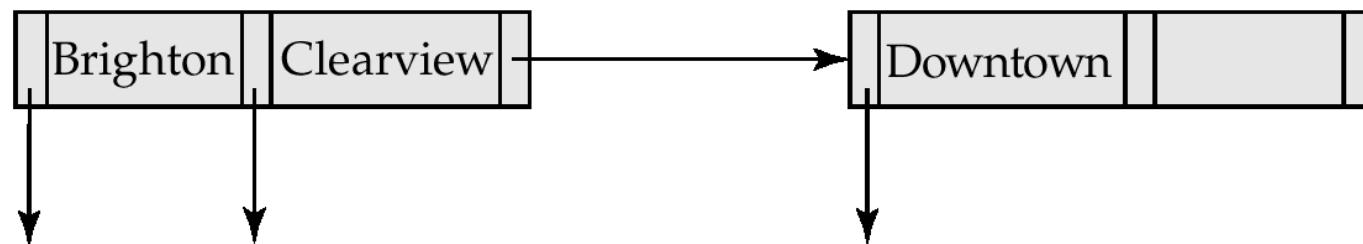
Caso contrário, divida o nó (junto com a nova entrada), redistribuindo as chaves e propagando mudanças para cima na árvore

Divisão de um Nó

Quando um nó folha atinge sua capacidade máxima durante uma inserção, é necessário dividi-lo em dois nós. Este processo mantém o balanceamento da árvore e pode propagar mudanças para os nós ancestrais.

Resultado da Divisão

A imagem acima ilustra o resultado da divisão do nó contendo **Brighton** e **Downtown** na inserção de **Clearview**. Note como o novo valor é distribuído entre os nós resultantes, mantendo a ordem crescente das chaves.



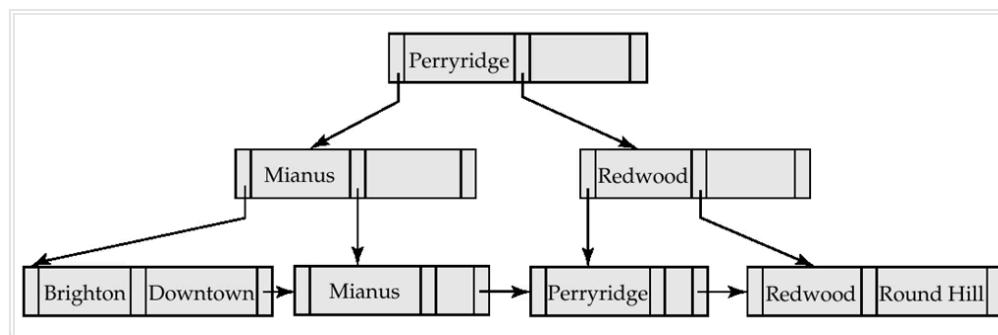
O processo de divisão envolve a criação de um novo nó, redistribuição das chaves existentes mais a nova chave, e atualização do nó pai para incluir um ponteiro para o novo nó criado.

Atualizações em Árvores B+: Inserção

Exemplo Completo de Inserção

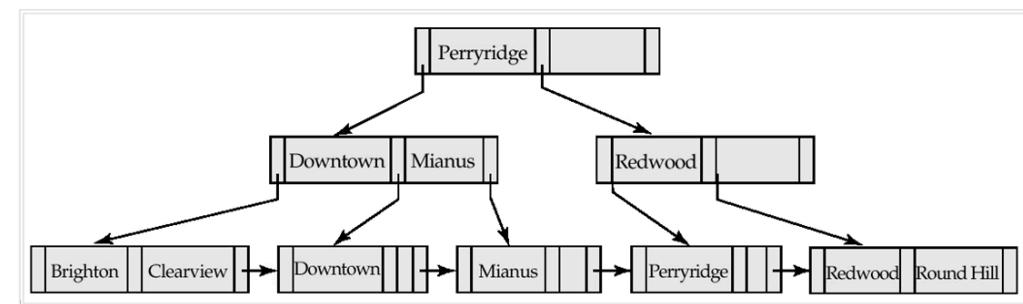
Veja como a estrutura da árvore B+ se modifica antes e depois da inserção do valor "**Clearview**". A transformação demonstra como a árvore se adapta dinamicamente para acomodar novos dados mantendo suas propriedades de平衡amento.

Estado Antes da Inserção



A árvore original possui nós balanceados com espaço para expansão

Estado Após a Inserção



Após inserir "Clearview", observe a divisão de nó e ajustes na estrutura

Este exemplo ilustra perfeitamente como as árvores B+ mantêm seu balanceamento por meio de divisões de nós quando necessário, garantindo que a altura da árvore cresça de forma controlada e uniforme.

Exclusão em Árvores B+

A exclusão em árvores B+ é uma operação complexa que pode envolver mesclagem de nós, redistribuição de chaves e ajustes na estrutura da árvore. O objetivo é manter as propriedades de balanceamento da árvore após a remoção de elementos.

Diferentemente da inserção, a exclusão pode causar **propagação de mudanças em cascata** para cima na árvore, especialmente quando nós ficam com menos elementos do que o mínimo permitido.

Consolidações em Exclusão em Árvores B+

Quando um nó fica com menos do que o número mínimo de elementos após uma exclusão, existem duas estratégias principais para restaurar o balanceamento da árvore:



Redistribuição (Empréstimo)

Se um nó irmão adjacente tiver elementos suficientes, pode-se **redistribuir** as chaves entre os nós. Um elemento é "emprestado" do irmão através do nó pai, mantendo a ordenação das chaves.

- Mantém o número total de nós
- Requer atualização de chaves no nó pai
- Preserva melhor a distribuição dos dados

Mesclagem (Consolidação)

Se o nó irmão também estiver no mínimo, os dois nós são **mesclados** em um único nó. Esta operação reduz o número de nós na árvore e pode propagar para cima.

- Reduz o número de nós na árvore
- Remove uma chave do nó pai
- Pode causar mesclagens em cascata
- Em casos extremos, reduz a altura da árvore

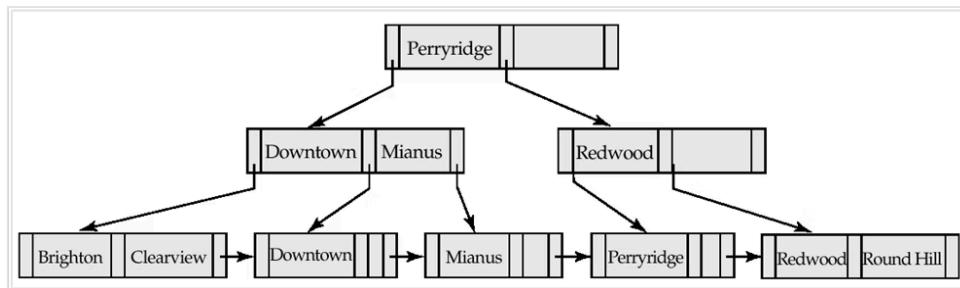
A escolha entre redistribuição e mesclagem depende da ocupação dos nós irmãos e tem impacto direto no desempenho futuro da árvore.

Exemplos de Exclusão em Árvore B+

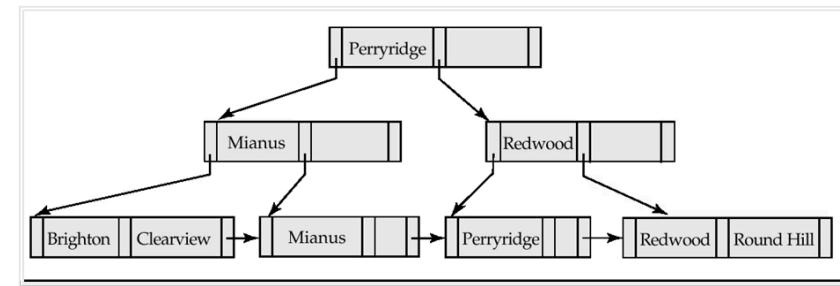
Exclusão Sem Propagação

No primeiro exemplo, analisamos a exclusão do nó folha contendo "**Downtown**". Este é um caso onde a exclusão não causa efeitos em cascata na estrutura da árvore.

Árvore Original



Após Exclusão de "Downtown"



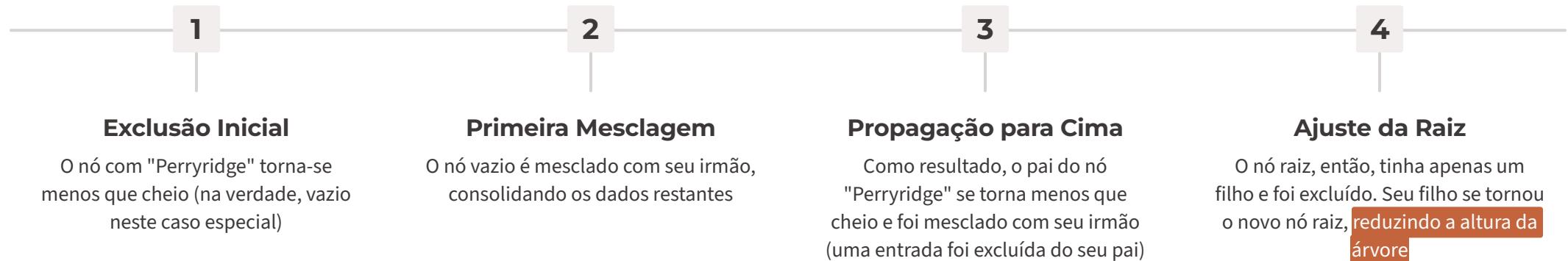
- ❑ **Observação Importante:** A remoção do nó folha contendo "Downtown" não resultou em seu pai tendo poucos ponteiros demais. Assim, as exclusões em cascata pararam com o pai do nó folha excluído, mantendo a integridade estrutural da árvore sem necessidade de operações adicionais.

Este é um cenário ideal de exclusão, onde a operação é localizada e não requer ajustes complexos na estrutura da árvore.

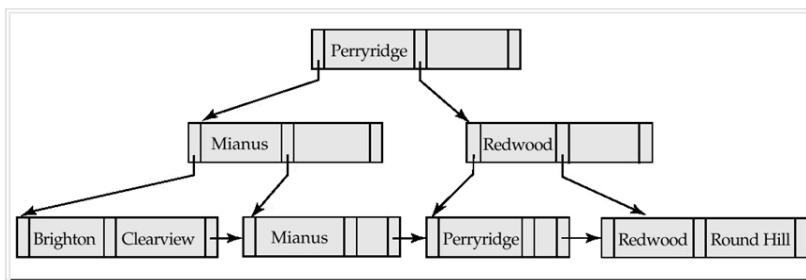
Exclusão com Mesclagem ao Irmão da Esquerda

Este exemplo demonstra um cenário mais complexo, onde a exclusão de "Perryridge" desencadeia uma série de mesclagens em cascata que afetam múltiplos níveis da árvore.

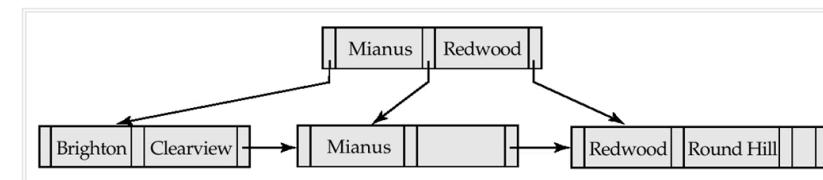
Sequência de Eventos



Estado Antes



Estado Depois



Exclusão com Empréstimo do Irmão à Esquerda

Este último exemplo ilustra a estratégia de redistribuição, onde em vez de mesclar nós, elementos são redistribuídos entre nós irmãos para manter o balanceamento.

Processo de Redistribution

Antes e depois da exclusão de "Perryridge"

Deficiência no Nó

O pai da folha contendo Perryridge tornou-se menos cheio após a exclusão

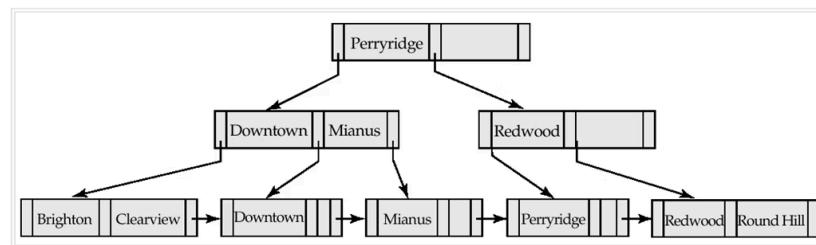
Empréstimo

O nó deficiente pediu emprestado um ponteiro do seu irmão da esquerda, que tinha elementos suficientes para compartilhar

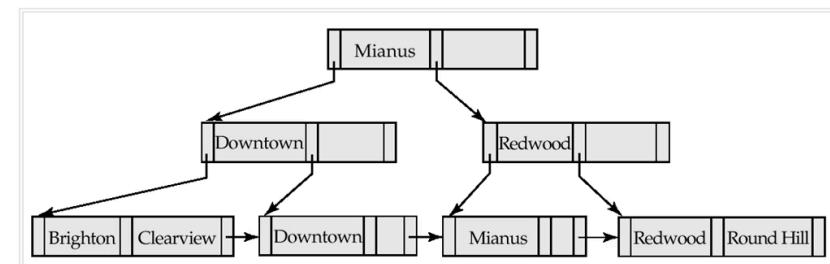
Atualização de Chaves

O valor de chave de busca no pai do pai muda como resultado da redistribuição, refletindo a nova organização dos dados

Configuração Inicial



Após Redistribution



A redistribuição é preferível à mesclagem quando possível, pois mantém uma distribuição mais uniforme dos dados e evita mudanças estruturais mais profundas na árvore.

Hash

Estruturas de hashing representam uma alternativa às árvores para indexação de dados. Enquanto árvores B+ organizam dados de forma ordenada, técnicas de hash permitem acesso direto através de funções matemáticas que mapeiam chaves para localizações físicas.

Esta seção explora como o hashing pode ser aplicado em sistemas de banco de dados, suas variantes e os desafios associados à sua implementação.

Hashing Estático

INDEXAÇÃO

01

Definição da Função de Hash

Uma função de hash h mapeia valores de chave de busca para um intervalo fixo de números inteiros correspondentes aos endereços dos baldes (buckets).

02

Alocação de Baldes

Os baldes são unidades de armazenamento, cada um capaz de conter múltiplos registros. O número de baldes é determinado na criação da estrutura e permanece fixo.

03

Distribuição de Registros

Cada registro é colocado no balde correspondente ao valor retornado pela função de hash aplicada à sua chave de busca.

O hashing estático oferece acesso direto em $O(1)$ no caso ideal, mas enfrenta desafios quando o número de registros cresce além da capacidade inicial planejada.

Exemplo de Organização de Arquivo de Hash

Configuração do Sistema

1 Arquivo de Conta Bancária

Considere um arquivo contendo informações de contas bancárias, onde cada registro inclui número da conta, nome da agência, saldo e outras informações relevantes.

2 Chave de Busca

Utilizamos **nome-agência** como chave de busca para demonstrar como diferentes agências são distribuídas pelos baldes do sistema de hash.

3 Função de Hash

A função calcula um valor numérico baseado nos caracteres do nome da agência, retornando um índice de balde através da operação módulo.

A escolha da função de hash e do número de baldes impacta diretamente a eficiência do sistema. Uma distribuição uniforme minimiza colisões e optimiza o desempenho das operações.

Visualização da Organização via Hash

Esta ilustração mostra a organização concreta de um arquivo de contas utilizando hashing com **nome-agência** como chave. Observe como registros de diferentes agências são distribuídos pelos baldes disponíveis.

Distribuição dos Registros

Cada balde pode conter múltiplos registros de contas. A distribuição depende do resultado da função de hash aplicada ao nome da agência.

Acesso aos Dados

Para localizar uma conta específica, calcula-se o hash do nome da agência, acessa-se o balde correspondente e busca-se linearmente dentro dele.

bucket 0	bucket 5
	A-102 Perryridge 400
	A-201 Perryridge 900
	A-218 Perryridge 700
bucket 1	bucket 6
bucket 2	bucket 7
	A-215 Mianus 700
bucket 3	bucket 8
A-217 Brighton 750	A-101 Downtown 500
A-305 Round Hill 350	A-110 Downtown 600
bucket 4	bucket 9
A-222 Redwood 700	

Funções de Hash

TEORIA

PRÁTICA

Pior Cenário

A pior função de hash possível mapeia todos os valores de chave para o mesmo balde, degradando o desempenho para tempo proporcional ao número total de registros - equivalente a uma busca linear.

Cenário Ideal

Uma função de hash ideal realiza **distribuição uniforme**, onde cada balde recebe aproximadamente o mesmo número de valores do conjunto de todas as chaves possíveis.

Implementação Prática

Funções de hash típicas operam sobre a **representação binária interna** da chave de busca. Este nível de abstração permite criar funções robustas e eficientes.

Exemplo para strings: Soma-se a representação binária de todos os caracteres na string e aplica-se a operação módulo pelo número de baldes. Esta abordagem simples frequentemente produz distribuições aceitáveis.

Outras técnicas incluem multiplicação por constantes especiais, divisão com análise de restos, ou combinações que aumentam a aleatoriedade da distribuição.

Tratamento de Estouros de Balde

Causas do Estouro

Baldes Insuficientes

Quando o número total de baldes é muito pequeno em relação ao volume de dados, mesmo com distribuição uniforme, baldes individuais podem exceder sua capacidade.

Distorção na Distribuição

Problemas surgem quando múltiplos registros compartilham o mesmo valor de chave, ou quando a função de hash produz distribuição não uniforme, concentrando registros em poucos baldes.

- ☐ **Realidade Prática:** Embora a probabilidade de estouro possa ser reduzida através de planejamento cuidadoso e boas funções de hash, ela não pode ser completamente eliminada em sistemas reais.

Para lidar com esta inevitabilidade, sistemas de hash implementam mecanismos de **baldes de estouro** que acomodam registros excedentes mantendo o acesso eficiente.

Encadeamento de Estouro

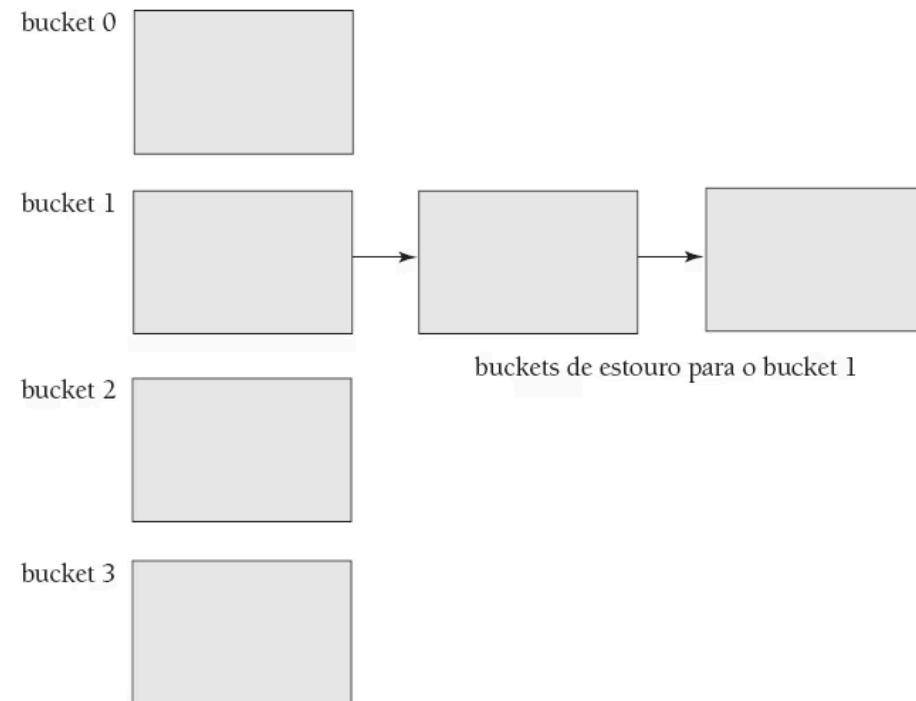
Hashing Fechado

A técnica de **encadeamento de estouro** conecta baldes adicionais em uma lista encadeada quando um balde primário atinge sua capacidade máxima.

Este esquema é conhecido como **hashing fechado** e é a abordagem preferida para aplicações de banco de dados devido à sua eficiência e simplicidade.

Alternativa: Hashing Aberto

Existe uma técnica chamada *hashing aberto* que não utiliza baldes de estouro, mas ela não é apropriada para aplicações de banco de dados devido às suas limitações em ambientes de alta concorrência.



A ilustração mostra como baldes de estouro são encadeados: quando um balde primário fica cheio, um novo balde é alocado e linkado, formando uma cadeia. Buscas seguem esta cadeia até encontrar o registro desejado ou determinar sua ausência.

Índices de Hash

INDEXAÇÃO SECUNDÁRIA

O hashing não se limita apenas à organização direta de arquivos - ele também pode criar poderosas [estruturas de índice](#) que aceleram o acesso aos dados.

Estrutura do Índice

Um índice de hash organiza as chaves de busca juntamente com seus ponteiros de registro associados em uma estrutura de arquivo de hash dedicada.

Natureza Secundária

Índices de hash são estritamente índices secundários. Se o arquivo já usa hashing para organização primária com a mesma chave, um índice hash adicional seria redundante.

Terminologia

Na prática, usamos o termo "índice de hash" de forma ampla para nos referir tanto às estruturas de índice secundárias quanto aos arquivos organizados diretamente via hash, apesar da distinção técnica.

Esta flexibilidade terminológica reflete o uso comum na literatura de banco de dados, onde o contexto geralmente deixa claro qual tipo de estrutura está sendo discutido.

Exemplo de Índice de Hash

Esta visualização demonstra a estrutura completa de um índice de hash aplicado a um conjunto de dados. Observe a separação clara entre a [estrutura de índice](#) e os [dados efetivos](#).

Camada de Índice

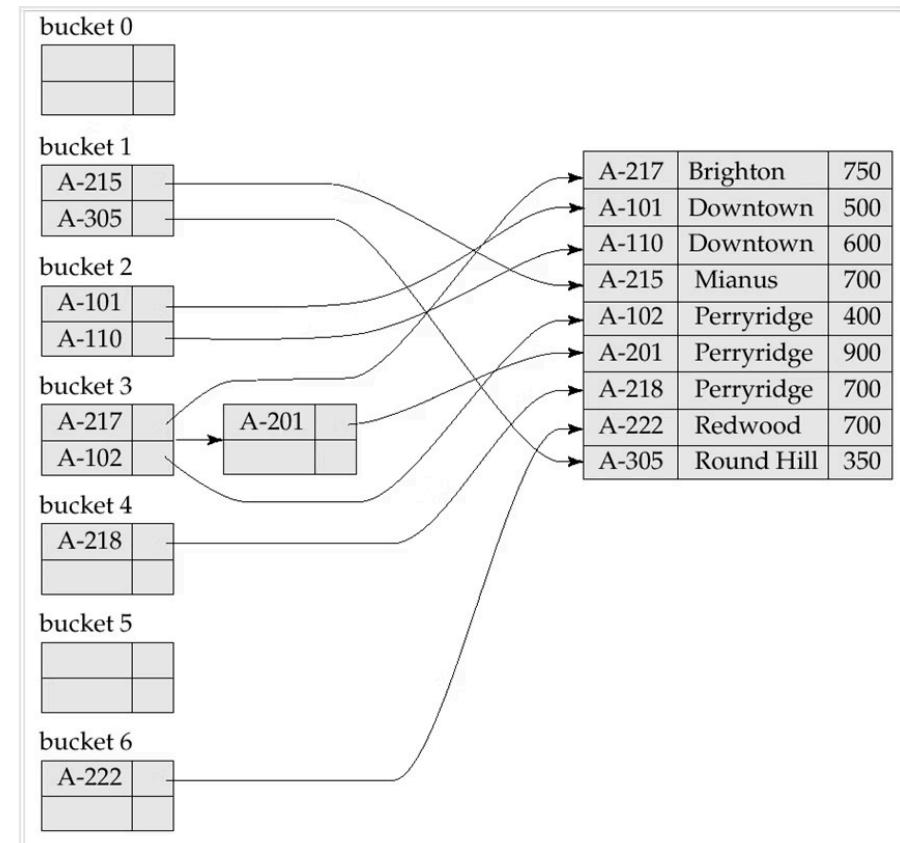
Os baldes contêm pares de (chave, ponteiro), onde cada ponteiro referencia a localização física do registro completo no arquivo de dados.

Camada de Dados

Os registros completos residem separadamente, permitindo que o índice permaneça compacto e eficiente para buscas rápidas.

Vantagem da Separação

Esta arquitetura em duas camadas permite múltiplos índices sobre diferentes atributos sem duplicar os dados completos.



Deficiências do Hashing Estático

LIMITAÇÕES

1

Tamanho Fixo de Baldes

O número de baldes é determinado na criação da estrutura e não pode ser facilmente alterado. À medida que o banco de dados cresce, a taxa de colisões aumenta progressivamente.

2

Degradação de Desempenho

Com o crescimento do arquivo, cadeias de estouro se tornam longas, degradando o tempo de busca de $O(1)$ ideal para algo próximo de busca sequencial em casos extremos.

3

Desperdício de Espaço

Se configurado para crescimento futuro, muitos baldes permanecem vazios ou subutilizados inicialmente, desperdiçando espaço de armazenamento precioso.

4

Reorganização Custosa

Para acomodar crescimento significativo, todo o arquivo precisa ser reorganizado com um novo número de baldes - uma operação extremamente custosa que pode requerer inatividade do sistema.

Estas limitações motivaram o desenvolvimento de técnicas de hashing dinâmico que adaptam automaticamente o número de baldes conforme o arquivo cresce ou encolhe.

Hashing Dinâmico

Adaptação ao Crescimento

Técnicas de hashing dinâmico resolvem as limitações do hashing estático permitindo que a estrutura **cresça e encolha dinamicamente** conforme necessário, sem requerer reorganização completa do arquivo.



Hash Extensível

Utiliza um diretório de ponteiros que pode dobrar de tamanho, permitindo crescimento incremental sem mover todos os dados.



Hash Linear

Expande gradualmente um balde por vez em uma sequência linear, evitando reorganizações em larga escala.

- Vantagem Principal:** Ambas as técnicas mantêm desempenho próximo de $O(1)$ mesmo com crescimento significativo do banco de dados, enquanto minimizam reorganizações custosas.

Nas próximas seções, exploraremos o [hash extensível](#) em detalhes, uma das técnicas dinâmicas mais populares em sistemas de banco de dados modernos.

Estrutura Geral do Hash Extensível

O hash extensível introduz uma camada de indireção através de um **diretório** que mapeia prefixos de hash para baldes de dados. Esta arquitetura permite expansão eficiente.

Diretório de Ponteiros

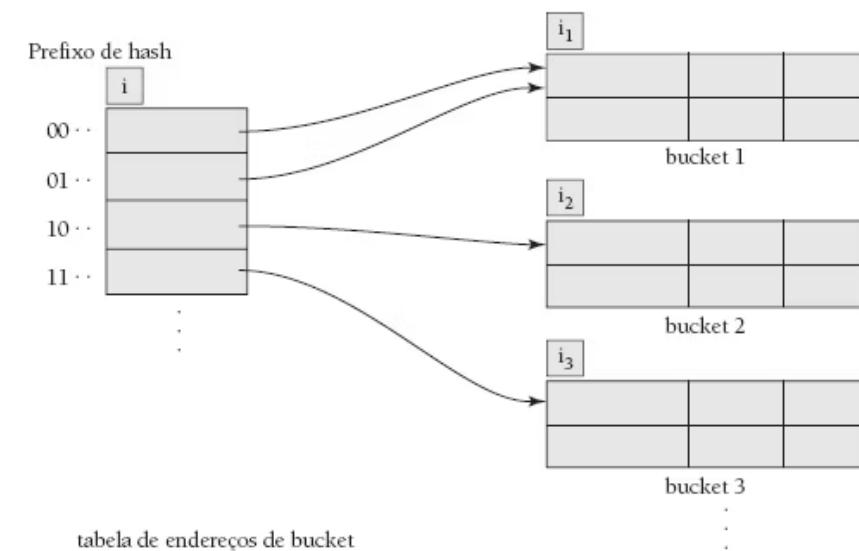
Uma tabela que usa os primeiros i bits do valor de hash para indexar ponteiros para baldes. O tamanho do diretório é 2^i .

Baldes de Dados

Contêineres que armazenam os registros reais. Múltiplas entradas do diretório podem apontar para o mesmo balde.

Profundidade Global e Local

A profundidade global (λ) determina bits usados no diretório. A profundidade local de cada balde indica quantos bits efetivamente o distinguem.



Uso da Estrutura de Hash Extensível

Operações de Busca e Inserção

01

Aplicar Função de Hash

Calcular o valor de hash completo da chave de busca usando uma função de hash que produz uma sequência de bits.

02

Consultar Diretório

Usar os primeiros i bits do hash como índice no diretório para localizar o ponteiro do balde apropriado.

03

Acessar Balde

Seguir o ponteiro até o balde de dados e procurar linearmente pelo registro desejado dentro dele.

04

Inserir ou Retornar

Para inserções, adicionar o registro ao balde se houver espaço. Para buscas, retornar o registro encontrado ou indicar ausência.

A beleza do hash extensível está na sua simplicidade de operação: a maioria das buscas requer apenas dois acessos a disco - um para o diretório e outro para o balde de dados.

Atualizações na Estrutura de Hash Extensível

INSCRIÇÃO

DIVISÃO

Inserção Simples

Quando há espaço disponível no balde apropriado, a inserção é direta: o registro é simplesmente adicionado ao balde identificado pelos bits de hash.

Divisão de Balde

Quando um balde está cheio, ele precisa ser dividido. Se a profundidade local do balde é menor que a profundidade global, apenas o balde é dividido.

Divisão Local

Incrementa-se a profundidade local, cria-se um novo balde, e redistribuem-se os registros baseado em um bit adicional do hash.



Expansão do Diretório

Se profundidade local = global, o diretório dobra de tamanho primeiro, depois o balde é dividido.

Esta estratégia de divisão incremental evita reorganização completa, mantendo o desempenho mesmo durante crescimento intenso do banco de dados.

Continuação: Atualizações no Hash Extensível

Exclusões e Coalescência



Remoção de Registros

Exclusões localizam o registro através do processo normal de busca e o removem do balde. Se o balde ficar vazio, pode ser elegível para coalescência com seu balde "irmão".

Coalescência de Baldes

Dois baldes que compartilham o mesmo prefixo de hash (exceto o último bit) podem ser fundidos em um único balde se o total de registros couber. Isso reduz o espaço utilizado.

Contração do Diretório

Se todas as profundidades locais se tornam menores que a profundidade global, o diretório pode ser reduzido pela metade. Esta operação é menos comum mas importante para liberar memória.

Consideração de Desempenho: Embora coalescência e contração do diretório sejam possíveis, muitas implementações as adiam ou evitam durante operações de alta concorrência para minimizar bloqueios e maximizar throughput.

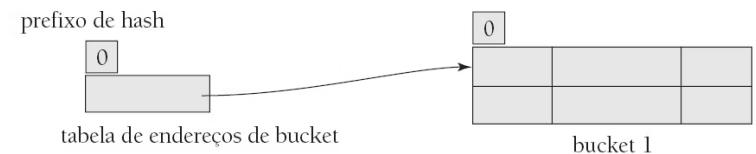
O hash extensível representa um equilíbrio elegante entre **flexibilidade**, **desempenho** e **uso eficiente de espaço**, tornando-o uma escolha popular em sistemas de gerenciamento de banco de dados modernos.

Uso da Estrutura de Hash Extensível: Exemplo Prático

Vamos explorar como funciona uma estrutura de hash extensível por meio de um exemplo passo a passo. Começaremos com uma configuração inicial simples e acompanharemos como a estrutura evolui à medida que novos registros são inseridos.

A estrutura inicial possui um **tamanho de balde igual a 2**, o que significa que cada balde pode armazenar no máximo dois registros antes de precisar ser dividido. Esta capacidade limitada permite que vejamos claramente como o hash extensível se adapta dinamicamente ao crescimento dos dados.

<i>nome_agência</i>	$h(\textit{nome_agência})$
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



Primeira Expansão: Inserindo Brighton e Downtown

Após a inserção de um registro de **Brighton** e dois registros de **Downtown**, observamos a primeira transformação significativa na estrutura de hash. Como o balde atingiu sua capacidade máxima, o sistema precisa se reorganizar para acomodar os novos dados.

Note como a profundidade local dos baldes começa a variar (indicada pelos valores 0 e 1). Isso demonstra a característica dinâmica do hash extensível: nem todos os baldes precisam ter a mesma profundidade simultaneamente, permitindo uma expansão mais eficiente e localizada.

01

Estado Inicial

Estrutura com capacidade limitada

02

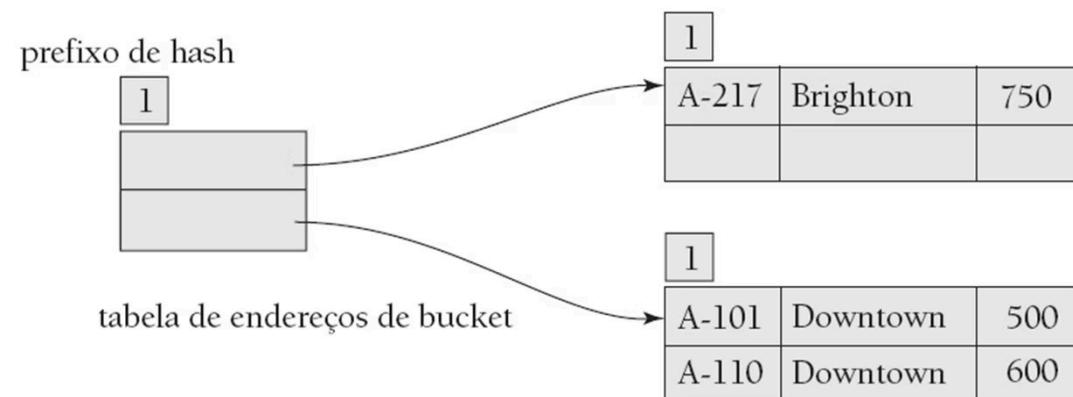
Inserção de Dados

Brighton e Downtown adicionados

03

Reorganização

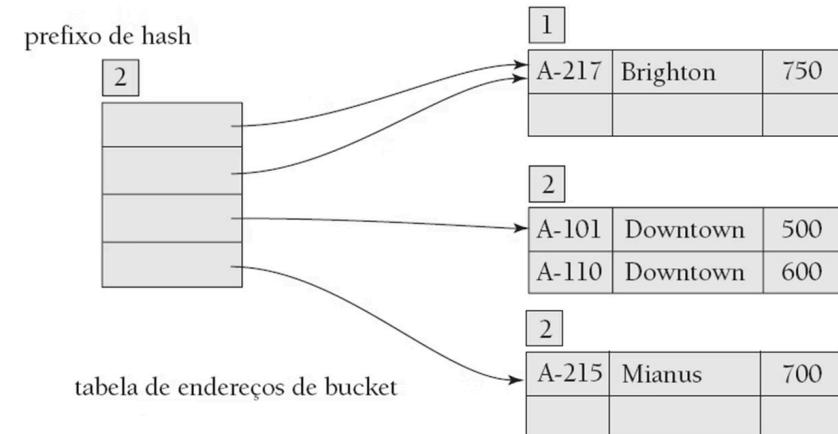
Baldes ajustados com profundidades 0 e 1



Segunda Expansão: Adicionando Mianus

A inserção do registro de **Mianus** desencadeia uma nova expansão na estrutura de hash. Observe como a tabela de diretório agora precisa crescer para acomodar mais ponteiros, refletindo o aumento na profundidade global da estrutura.

As profundidades locais agora variam entre 0, 00, 01, 10 e 11, demonstrando como o hash extensível utiliza progressivamente mais bits do valor hash para determinar o posicionamento dos registros. Esta expansão gradual é muito mais eficiente do que reorganizar toda a estrutura de uma só vez.



Características da Expansão

- Profundidade global aumentada
- Novo diretório de ponteiros criado
- Apenas baldes necessários são divididos

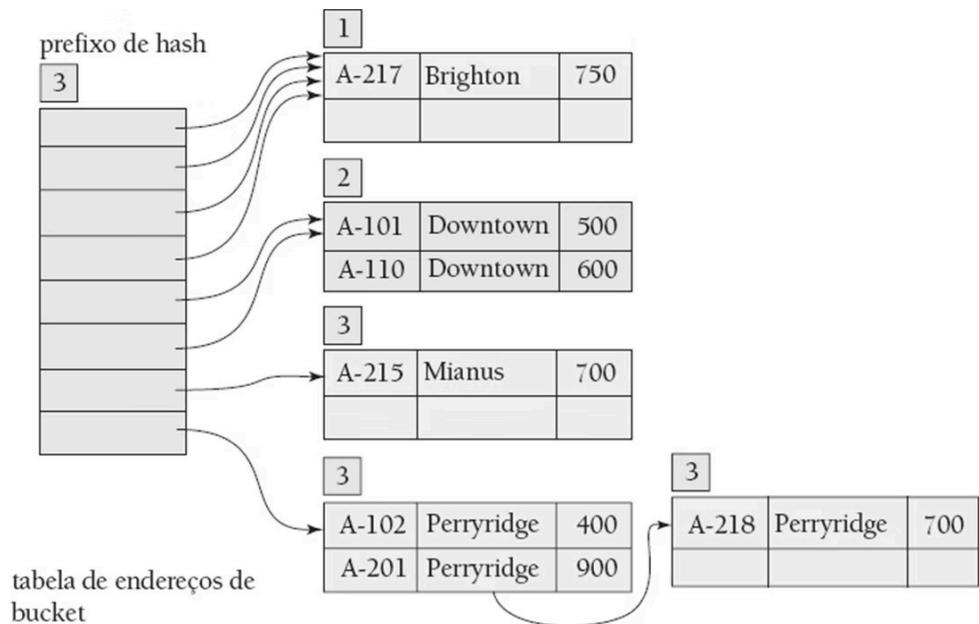
Vantagens do Processo

- Expansão localizada e eficiente
- Sem reorganização completa
- Adaptação dinâmica ao crescimento

Terceira Expansão: Múltiplos Registros de Perryridge

Com a inserção de **três registros de Perryridge**, a estrutura demonstra sua capacidade de lidar com múltiplos registros que compartilham o mesmo valor hash inicial. A complexidade da estrutura aumenta significativamente, com profundidades variando de 0 até 111.

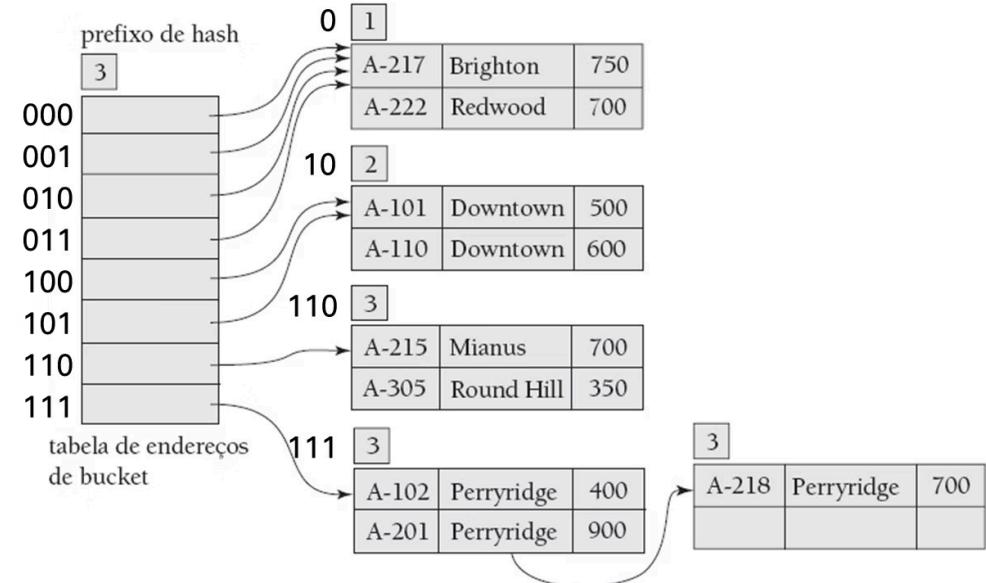
Este exemplo ilustra um cenário realista onde muitos registros podem mapear para localizações próximas. O hash extensível gerencia essa situação criando subdivisões adicionais apenas onde necessário, mantendo outros baldes inalterados e economizando espaço.



Estado Final: Redwood e Round Hill

Após adicionar os registros de **Redwood** e **Round Hill**, chegamos ao estado final da nossa estrutura de hash extensível. A organização complexa resultante mostra como a estrutura se adaptou organicamente ao padrão de inserção dos dados.

Cada balde na estrutura final mantém sua profundidade local específica, e a tabela de diretório mapeia eficientemente para os baldes apropriados. Esta configuração final representa um equilíbrio entre a necessidade de armazenar todos os registros e a minimização do espaço desperdiçado.



- ❑ **Ponto-Chave:** A estrutura final demonstra como o hash extensível cresce de forma assimétrica, expandindo apenas as partes necessárias enquanto mantém outras áreas compactas. Isso resulta em uso eficiente de espaço e desempenho otimizado.

Hashing Extensível: Análise Crítica

Benefícios do Hashing Extensível

- **Desempenho consistente:** O desempenho não diminui à medida que o arquivo cresce, mantendo tempos de busca previsíveis mesmo com milhões de registros
- **Sobrecarga mínima de espaço:** A estrutura cresce de forma incremental e localizada, evitando desperdício excessivo de memória
- **Adaptação dinâmica:** Ajusta-se automaticamente ao volume de dados sem necessidade de reorganização completa

Desvantagens e Limitações

- **Nível extra de indireção:** Para encontrar um registro, é necessário primeiro consultar a tabela de endereços de balde, adicionando um passo extra ao processo de busca
- **Crescimento da tabela de endereços:** Em sistemas muito grandes, a própria tabela de endereços pode se tornar maior que a memória disponível, exigindo o uso de estruturas auxiliares em árvore
- **Custo de redimensionamento:** Mudar o tamanho da tabela de endereços é uma operação dispendiosa que pode impactar temporariamente o desempenho do sistema

Comparação: Indexação Ordenada vs. Hashing

A escolha entre indexação ordenada e hashing não é uma decisão simples - cada técnica tem seu lugar dependendo do tipo de consulta que será executado com mais frequência no sistema.

Quando Usar Hashing

O hashing é **geralmente superior** na recuperação de registros com um **valor específico da chave**. Se sua aplicação executa principalmente buscas exatas (por exemplo, "encontre o cliente com CPF = 12345678900"), o hashing oferece tempo de acesso praticamente constante O(1).

Casos de uso ideais: sistemas de login, consultas de identificadores únicos, caches de dados

Quando Usar Índices Ordenados

Se as **consultas por intervalo** são comuns em seu sistema, os índices ordenados são claramente preferíveis. Eles permitem percorrer eficientemente ranges de valores (por exemplo, "todos os produtos com preço entre R\$ 100 e R\$ 500").

Casos de uso ideais: relatórios analíticos, consultas de data/hora, buscas por faixas de valores

Consultas por Igualdade e por Intervalo

Entender a diferença fundamental entre consultas por igualdade e por intervalo é crucial para escolher a estrutura de índice apropriada para seu banco de dados.



Consultas por Igualdade

Exemplo: saldo = 1000

Buscam um valor exato e específico. O hashing é ideal porque calcula diretamente a posição do registro através da função hash, resultando em acesso quase instantâneo.

Desempenho: $O(1)$ com hashing

Consultas por Intervalo

Exemplo: saldo BETWEEN 500 AND 2000

Buscam múltiplos valores dentro de um range. Índices ordenados são superiores porque mantêm os dados organizados sequencialmente, permitindo varredura eficiente.

Desempenho: $O(\log n + k)$ com índice B-tree

Fatores de Decisão

- Tipo de consulta mais frequente no sistema
- Padrão de acesso aos dados pelos usuários
- Distribuição dos valores no atributo indexado

Índices de Mapa de Bits

Chegamos agora a uma técnica especializada e extremamente eficiente para um tipo específico de dado: os **índices de mapa de bits**. Esta estrutura revolucionou como consultamos dados categóricos em grandes bancos de dados.



Dados Categóricos

Ideais para atributos com poucos valores distintos: gênero, estado, categoria de produto, status



Representação Binária

Cada valor possível tem um vetor de bits onde 1 indica presença e 0 indica ausência



Eficiência Extrema

Operações bit a bit permitem consultas complexas com múltiplos atributos de forma ultra-rápida

Fundamentos dos Índices de Mapa de Bits

Os índices de mapa de bits são projetados especificamente para a **consulta eficiente de dados categóricos discretizados**. Sua elegância está na simplicidade: cada registro é representado por sua posição numérica sequencial, começando do zero.

01

Numeração Sequencial

Os registros são numerados de 0 a n-1, permitindo acesso direto por posição

02

Acesso Rápido

Dado um número n, deve ser fácil recuperar o registro correspondente

03

Otimização

Particularmente eficiente quando registros têm tamanho fixo

Atributos Aplicáveis

O mapa de bits funciona melhor com atributos que assumem uma **quantidade relativamente pequena de valores distintos**:

- **Demográficos:** gênero, país, estado, cidade
- **Categóricos:** departamento, tipo de produto, status
- **Discretizados:** faixas de renda (0-9999, 10000-19999, 20000-50000, 50000+)
- **Booleanos:** ativo/inativo, público/privado

Estrutura e Funcionamento dos Mapas de Bits

Em sua forma mais simples, um índice de mapa de bits sobre um atributo mantém **um mapa de bits separado para cada valor distinto** do atributo. Cada mapa de bits tem tantos bits quanto o número total de registros na relação.

número de registro	nome	sexo	endereço	nível_renda	Mapas de bits para sexo	Mapas de bits para nível_renda
0	John	m	Perryridge	L1	m 10010	L1 10100
1	Diana	f	Brooklyn	L2	f 01101	L2 01000
2	Mary	f	Jonestown	L1		L3 00001
3	Peter	m	Brooklyn	L4		L4 00010
4	Kathy	f	Perryridge	L3		L5 00000

1

Criação do Bitmap

Para cada valor v do atributo, cria-se um vetor de bits

2

Mapeamento de Registros

Cada posição no vetor corresponde a um registro específico

3

Marcação de Valores

Bit = 1 se o registro tem valor v; bit = 0 caso contrário

- Exemplo Prático: Para um atributo "Gênero" com 1 milhão de registros, teríamos dois bitmaps (um para "M" e outro para "F"), cada um com 1 milhão de bits. Se o registro na posição 42 é do gênero masculino, então o bit 42 no bitmap "M" seria 1, e o bit 42 no bitmap "F" seria 0.

Consultas com Mapas de Bits: Operações Booleanas

A verdadeira potência dos índices de mapa de bits se revela quando executamos **consultas sobre múltiplos atributos**. As operações bit a bit permitem combinar condições de forma extremamente eficiente.

Interseção (AND)

Combina dois mapas aplicando AND bit a bit

$$100110 \text{ AND } 110011 = 100010$$

Útil para condições que devem ser satisfeitas **simultaneamente**

União (OR)

Combina dois mapas aplicando OR bit a bit

$$100110 \text{ OR } 110011 = 110111$$

Útil para condições onde **qualquer uma** pode ser satisfeita

Complementação (NOT)

Inverte todos os bits do mapa

$$\text{NOT } 100110 = 011001$$

Útil para consultas de **negação**

Exemplo: Consulta Combinada

Encontrar homens com nível de receita L1:

Bitmap "Homem": 10010

Bitmap "Receita L1": 10100

Resultado: $10010 \text{ AND } 10100 = 10000$

Vantagens das Operações

- Processamento em nível de hardware (CPU)
- Múltiplos bits processados simultaneamente
- Contagem rápida sem recuperar registros

Eficiência de Espaço dos Mapas de Bits

Uma das características mais impressionantes dos índices de mapa de bits é sua **extraordinária economia de espaço**, especialmente quando comparados ao tamanho da relação original.

1/800

Razão de Compressão

Se um registro tem 100 bytes, o espaço para um único mapa de bits é apenas 1/800 do espaço usado pela relação

1%

Espaço Total

Com 8 valores distintos, todos os mapas de bits juntos ocupam apenas 1% do tamanho da relação original

Considerações Especiais

Para manter a integridade dos índices, algumas questões operacionais precisam ser endereçadas:

- **Exclusão de registros:** Não basta zerar os bits - é necessário marcar explicitamente que aquela posição está vazia
- **Mapa de bits de existência:** Um bitmap adicional anota se existe um registro válido em cada posição
- **Compactação:** Periodicamente, pode ser necessário reorganizar para recuperar espaço de registros deletados

Dica Prática: Em data warehouses com bilhões de registros e atributos categóricos, mapas de bits podem reduzir consultas que levariam minutos para poucos segundos, consumindo uma fração do espaço de um índice B-tree tradicional.

Índice Composto: Múltiplos Atributos

Um **índice composto** é ordenado pelo primeiro atributo e, em seguida, pelos demais atributos em sequência. Esta estrutura é ideal quando consultas frequentemente filtram ou ordenam por múltiplas colunas.

Índice Composto (Agência, Saldo)

Agência	Saldo
Brighton	750
Downtown	500
Downtown	600
Mianus	700
Perryridge	400
Perryridge	900
Redwood	700
Round Hill	350

Tabela Original (Conta)

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

Ordenação primária por agência, secundária por saldo

Ordenação Hierárquica

Registros são primeiramente agrupados pelo atributo primário (Agência), depois ordenados pelo secundário (Saldo) dentro de cada grupo

Consultas Otimizadas

Excelente para queries como "encontre todas as contas da agência X com saldo > Y"

Criação de Índices em SQL

Índices são estruturas fundamentais que otimizam o desempenho de consultas em bancos de dados relacionais. Eles funcionam como catálogos que permitem acesso rápido aos dados, reduzindo significativamente o tempo de busca em tabelas grandes.

Definição de Índice em SQL

Um índice em SQL é uma estrutura de dados auxiliar que melhora a velocidade de recuperação de registros em uma tabela. A criação de índices segue uma sintaxe específica e oferece diferentes opções de configuração.



Sintaxe Básica

Para criar um índice, utilize o comando:

```
CREATE INDEX nome_index  
ON tabela(coluna1, coluna2)
```

Exemplo prático:

```
CREATE INDEX  
emp_pnome_unome  
ON empregado(pnome, unome)
```



Índices Únicos

Use CREATE UNIQUE INDEX para especificar indiretamente e impor a condição de que a chave de busca é uma chave candidata.

Observação: Não é realmente necessário se a restrição de integridade UNIQUE da SQL for admitida no esquema da tabela.



Remoção de Índices

Para remover um índice existente do banco de dados, utilize:

```
DROP INDEX nome_index
```

Esta operação é irreversível e remove completamente a estrutura do índice, liberando espaço em disco.

Consultas Envolvendo Múltiplos Atributos

Índices são ferramentas poderosas para otimizar consultas que envolvem múltiplos atributos. A escolha da estratégia correta pode impactar dramaticamente o desempenho das operações de busca.

Exemplo de consulta multi-atributo:

```
SELECT pnome  
FROM empregado  
WHERE dno = 5 AND salario = 30000
```

Estratégias de Processamento com Índices Isolados

01

Índice em DNO

Usar o índice sobre **dno** para encontrar registros onde **dno = 5**, depois testar cada registro encontrado para verificar se **salario = 30000**

03

Interseção de Ponteiros

Usar o índice **dno** para encontrar ponteiros de todos os registros do departamento 5. Simultaneamente, usar índice sobre **salario = 30000**. Apanhar a interseção dos dois conjuntos de ponteiros obtidos

02

Índice em Salário

Usar o índice sobre **salario** para encontrar registros onde **salario = 30000**, depois testar cada registro para verificar se **dno = 5**

04

Índice Composto

Utilizar um índice composto sobre **(dno, salario)** para busca direta e eficiente dos registros que satisfazem ambas as condições

Índices Compostos

Índices compostos são estruturas que combinam múltiplos atributos em uma única chave de busca, oferecendo desempenho superior para consultas que filtram por vários campos simultaneamente.

Cenários de Uso e Eficiência

Condições Exatas Múltiplas

Suponha que temos um índice composto **(dno, salario)**

Para a cláusula WHERE:

```
WHERE dno = 5 AND salario = 30000
```

O índice sobre a chave de busca combinada apanha apenas registros que satisfazem ambas as condições. O uso de índices separados é menos eficiente — podemos apanhar muitos registros (ou ponteiros) que satisfazem apenas uma das condições.

Primeiro Atributo Fixo

Também pode tratar de modo eficiente quando o primeiro atributo está fixo:

```
WHERE dno = 5 AND salario < 30000
```

Neste caso, o índice pode navegar eficientemente pelos valores de salário dentro do departamento específico.

Limitação: Primeiro Valor em Intervalo

Mas pode **não tratar de modo eficiente** quando o primeiro valor está em intervalo:

```
WHERE dno <= 5 AND salario = 30000
```

Pode trazer muitos registros que satisfazem a primeira restrição, mas não a segunda condição, resultando em desempenho degradado.

Índice que Cobre uma Consulta

Um índice **cobre uma consulta** quando todos os atributos necessários pela consulta estão presentes no próprio índice. Isso permite que o banco de dados responda à consulta sem jamais acessar a tabela principal - uma otimização poderosa que pode **reduzir drasticamente o tempo de resposta**.

Índice Contém Tudo

Todos os atributos usados na cláusula SELECT, WHERE e ORDER BY estão no índice

Sem Acesso ao Arquivo

Não é necessário acessar o arquivo de dados principal - resposta vem direto do índice

Exemplo Conceitual

Índice: (nomeAgencia, saldo)

Consulta coberta:

```
SELECT nomeAgencia, saldo  
FROM Conta  
WHERE nomeAgencia = 'Downtown'  
ORDER BY saldo;
```

Benefícios Principais

- Redução drástica de I/O:** Menos acessos a disco
- Desempenho superior:** Pode ser 10x a 100x mais rápido
- Menor contenção:** Menos bloqueios de tabela
- Cache eficiente:** Índices geralmente cabem em memória

Todos os atributos necessários estão no índice!

- Conclusão:** A escolha correta de índices - sejam eles hash, ordenados, bitmap ou compostos - pode transformar completamente o desempenho de um banco de dados. Analise seus padrões de consulta, entenda suas estruturas de dados, e selecione a ferramenta certa para cada situação.

Filtros de Bloom

Um **filtro de Bloom** é uma estrutura de dados probabilística usada para verificar a pertinência de um valor em um conjunto. Esta estrutura elegante pode retornar verdadeiro (com baixa probabilidade) mesmo se um elemento não estiver presente, mas nunca retorna falso quando um elemento está realmente presente. Esta propriedade o torna ideal para filtrar conjuntos irrelevantes de forma eficiente.

Estrutura Fundamental

- A estrutura de dados principal é um único bitmap
- Para um conjunto com n elementos, o tamanho típico do bitmap é $10n$ bits
- Utiliza múltiplas funções hash independentes para reduzir colisões

Função Hash Única

Para cada elemento s no conjunto S , calcule $h(s)$ e defina o bit $h(s)$

Para consultar um elemento v , calcule $h(v)$ e verifique se o bit está definido

Problema: Colisões

Chance significativa de falso positivo devido à colisão de hash

10% de chance com 10^n bits

Filtros de Bloom: Múltiplas Funções Hash

A ideia central do filtro de Bloom é reduzir falsos positivos usando múltiplas funções hash $hi()$ para $i=1..k$. Esta abordagem aumenta dramaticamente a precisão do filtro ao custo de computação adicional mínima.

01

Inserção de Elementos

Para cada elemento s no conjunto S , e para cada i , calcule $hi(s)$ e defina o bit $hi(s)$

02

Consulta de Elementos

Para consultar um elemento v , para cada i , calcule $hi(v)$ e verifique se o bit $hi(v)$ está definido

03

Decisão de Pertinência

Se o bit $hi(v)$ estiver definido para cada i , reporte v como presente no conjunto. Caso contrário, reporte v como ausente



Resultado Impressionante: Com $10n$ bits e $k=7$ funções hash, a taxa de falso positivo reduz para **1%** em vez de 10% com $k=1$

Índices Otimizados para Escrita

O Desafio das Árvores B+

O desempenho das árvores B+ pode ser insatisfatório para cargas de trabalho intensivas em escrita. Cada inserção pode exigir uma operação de I/O por folha, assumindo que todos os nós internos estão em memória.

<100

Inserções/segundo

Por disco magnético

1

Sobrescrita de Página

Por inserção em flash

Duas Abordagens para Reduzir Custo de Escritas

Log-Structured Merge Tree (LSM)

Utiliza estrutura em camadas com mesclagem periódica para otimizar operações de escrita sequencial

Buffer Tree

Implementa buffers em nós internos para acumular inserções antes de propagá-las aos níveis inferiores

Árvore Log Structured Merge (LSM)

A árvore LSM é uma estrutura de dados sofisticada que otimiza operações de inserção por meio de uma hierarquia de árvores com fusões periódicas. Vamos considerar apenas inserções e consultas inicialmente para entender o conceito fundamental.

Nível L0: Árvore em Memória

Os registros são inseridos primeiro em uma árvore na memória (árvore L0). Esta estrutura mantém os dados mais recentes e permite inserções extremamente rápidas.

Nível L1: Primeira Persistência

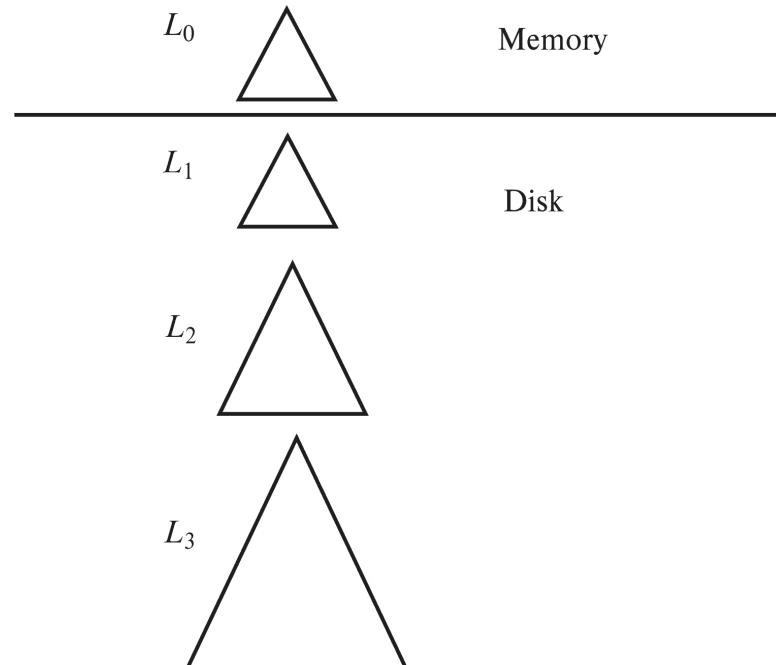
Quando a árvore em memória está cheia, os registros são movidos para o disco (árvore L1). Uma árvore B+ é construída usando construção bottom-up, mesclando a árvore L1 existente com os registros da árvore L0.

Níveis Superiores: Crescimento Escalável

Quando a árvore L1 excede um limite, ela é mesclada na árvore L2, e assim por diante para mais níveis. O limite de tamanho para a árvore L_{i+1} é k vezes o limite de tamanho para a árvore L_i .

Estrutura Visual da LSM Tree

A imagem ilustra a organização hierárquica das árvores LSM, mostrando como os dados fluem da memória para níveis progressivamente maiores no disco.



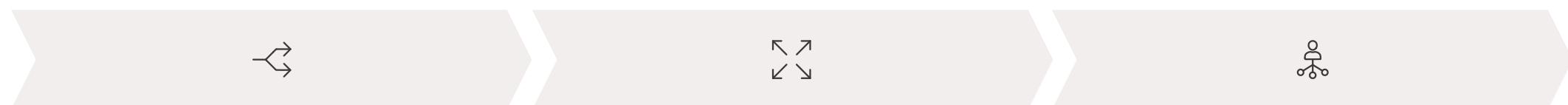
Otimizações da LSM Tree

Rolling Merge: Mesclagem Contínua

Uma otimização importante da LSM tree é a técnica de **rolling merge**, que realiza a mesclagem de forma incremental e contínua, em vez de executar grandes mesclagens periódicas. Isso distribui o custo de I/O ao longo do tempo e evita picos de latência.

Implementação Particionada

As árvores LSM e Stepped Merge são frequentemente implementadas em uma relação particionada para melhorar a escalabilidade e o paralelismo:



Particionamento

Cada partição tem seu tamanho definido para um máximo específico

Divisão Dinâmica

Partições são divididas quando excedem o tamanho máximo configurado

Distribuição

As partições são distribuídas entre múltiplas máquinas para processamento paralelo

- Esta abordagem particionada é fundamental para sistemas de BigData, permitindo escalabilidade horizontal e alta disponibilidade em ambientes distribuídos.

Índice Stepped Merge

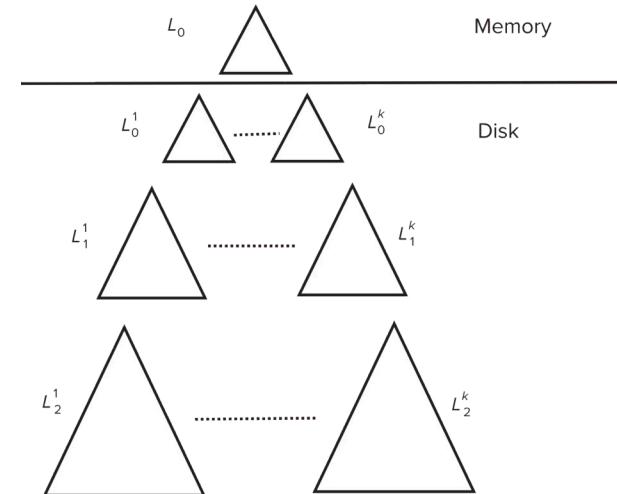
O **índice Stepped-merge** é uma variante da árvore LSM que mantém k árvores em cada nível no disco, oferecendo um equilíbrio diferente entre custo de escrita e custo de leitura.

Mecânica de Funcionamento

Quando todas as k árvores existem em um nível, elas são mescladas em uma única árvore do próximo nível. Esta abordagem reduz o custo de escrita em comparação com a árvore LSM tradicional, pois as mesclagens ocorrem com menos frequência.

Desafio: Custo de Consultas

As consultas se tornam ainda mais caras, pois muitas árvores precisam ser consultadas para encontrar um único registro. Este é o principal trade-off da abordagem stepped merge.



Otimização para Consultas Pontuais

01

Computar Filtro de Bloom

Calcule um filtro de Bloom para cada árvore e armazene-o na memória

02

Verificação Rápida

Consulte uma árvore apenas se o filtro de Bloom retornar um resultado positivo

03

Redução de I/O

Esta otimização elimina consultas desnecessárias a árvores que definitivamente não contêm o registro procurado

LSM Trees: Operações Avançadas

Gerenciamento de Deleções

A deleção em árvores LSM é tratada de forma elegante através da adição de entradas especiais de "delete" (tombstone). Esta abordagem evita a necessidade de localizar e remover imediatamente o registro original.

Processo de Deleção

Quando um registro é deletado, uma entrada especial "delete" é inserida no sistema. As consultas encontrarão tanto a entrada original quanto a entrada de deleção, e devem retornar apenas as entradas que não têm uma entrada de deleção correspondente.

Mesclagem e Limpeza

Durante a mesclagem de árvores, se encontrarmos uma entrada de deleção correspondente a uma entrada original, ambas são descartadas, liberando espaço de armazenamento.

Operações de Atualização

- As atualizações são tratadas como uma combinação de **inserção + deleção**, mantendo a simplicidade do modelo de dados e aproveitando os mecanismos existentes.

Aplicações Modernas

As árvores LSM foram originalmente introduzidas para índices baseados em disco, mas provaram ser extremamente úteis para minimizar apagamentos em índices baseados em memória flash.



Sistemas BigData

Google BigTable e Apache Cassandra usam variantes stepped-merge de árvores LSM



Bancos de Dados NoSQL

MongoDB implementa LSM trees para otimização de armazenamento



Storage Engines

SQLite4, LevelDB e MyRocks (MySQL) utilizam árvores LSM para alto desempenho

Buffer Tree

Conceito Fundamental

A **Buffer Tree** é uma alternativa elegante à árvore LSM que mantém a estrutura tradicional da árvore B+ enquanto otimiza as operações de escrita.

Ideia Central

Cada nó interno da árvore B+ possui um buffer para armazenar inserções. As inserções são movidas para níveis inferiores apenas quando o buffer está cheio. Com um buffer grande, muitos registros são movidos para o nível inferior de cada vez, e o custo de I/O por registro diminui proporcionalmente.

Benefícios da Buffer Tree

1

Menor Overhead em Consultas

Consultas são mais eficientes do que em árvores LSM, pois seguem uma única estrutura hierárquica

2

Flexibilidade Estrutural

Pode ser usado com qualquer estrutura de índice em árvore, não apenas B+

3

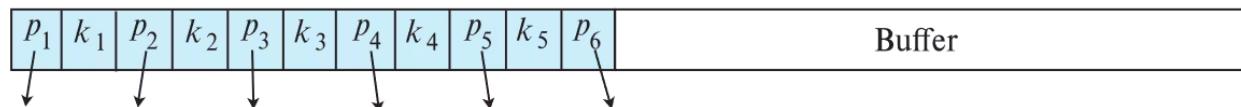
Implementação Prática

Usado em índices PostgreSQL Generalized Search Tree (GiST)

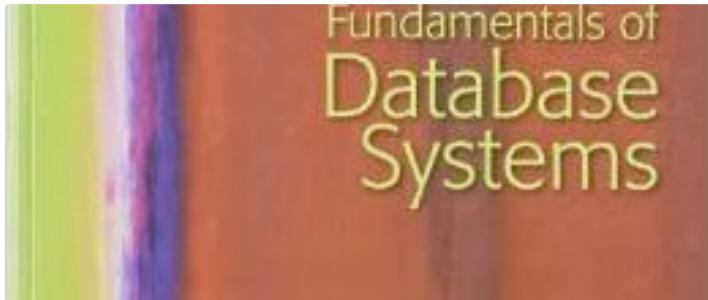
Desvantagem

- Trade-off:** A Buffer Tree realiza mais operações de I/O aleatórias do que a árvore LSM, o que pode impactar o desempenho em certos cenários de carga de trabalho.

Internal node



Referências



Elmasri & Navathe

Fundamentals of Database Systems

Pearson, 2016

Referência abrangente sobre fundamentos de sistemas de bancos de dados, cobrindo aspectos teóricos e práticos.

Korth, Sudarshan & Silberschatz

Database System Concepts

McGraw-Hill, 2019

Texto fundamental que serviu como base para a maioria dos exemplos apresentados nesta apresentação.

Özsu & Valduriez

Principles of Distributed Database Systems

Springer Nature, 2019

Obra especializada em sistemas de bancos de dados distribuídos, essencial para compreensão avançada.

- Nota:** Os conceitos e exemplos apresentados baseiam-se principalmente na literatura clássica de sistemas de bancos de dados, em especial *Database System Concepts* e *Fundamentals of Database Systems*.