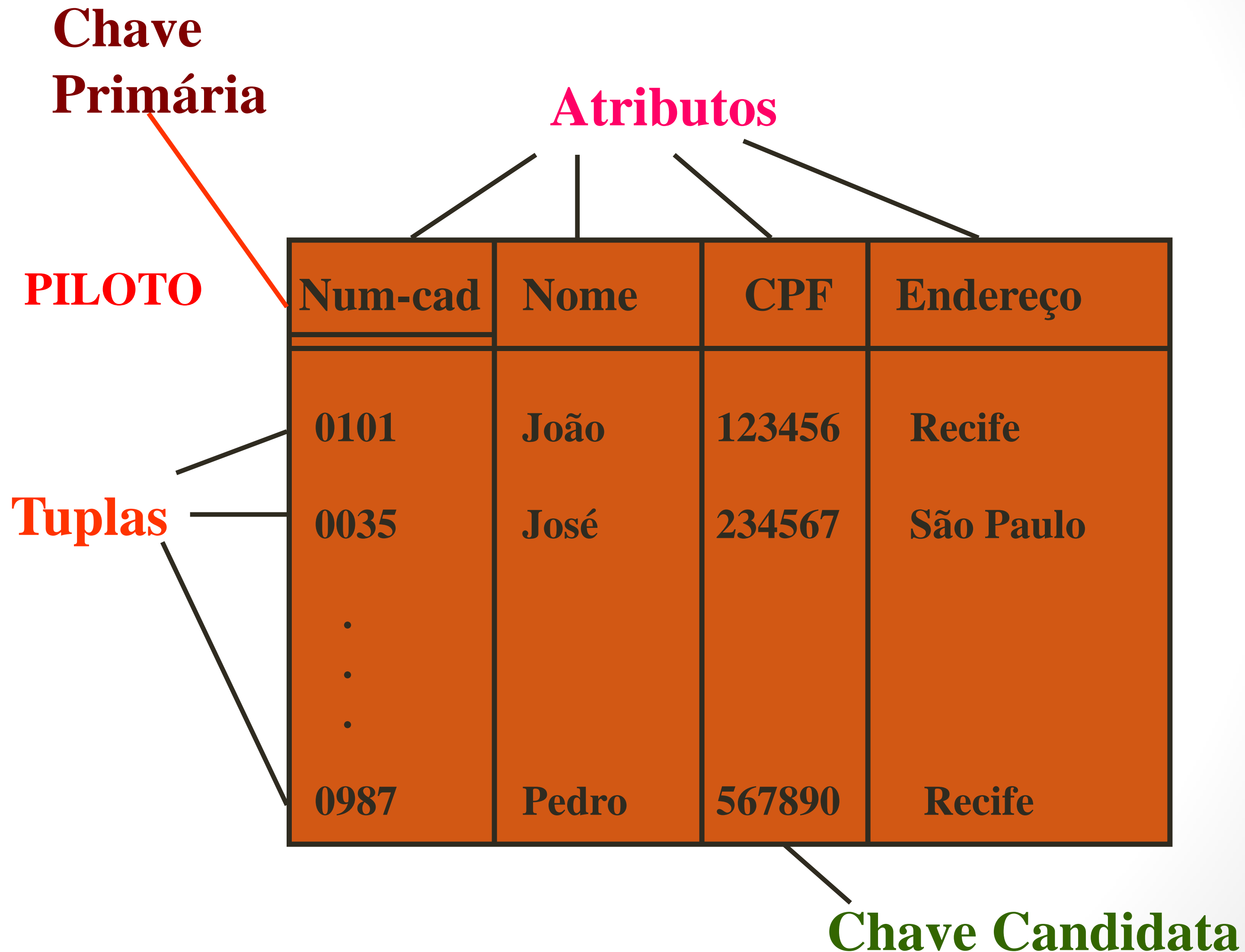
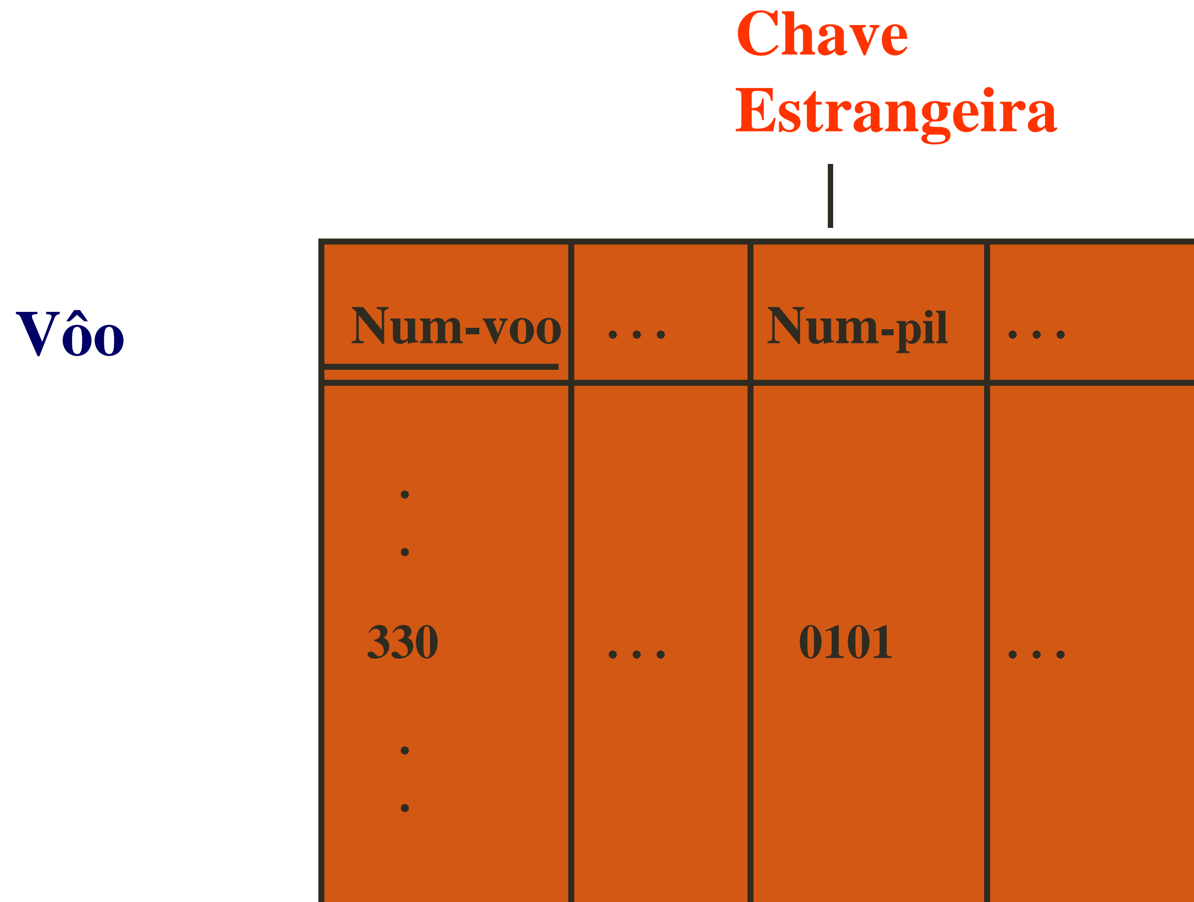
Modelo Relacional

Josino Rodrigues

Modelo Relacional



Modelo Relacional



Álgebra Relacional

- Junção: produz todas as combinações de tuplas de R1 e R2 que satisfazem a condição de junção
  - JUNC R1 <condição de junção> R2

Num-cad	Nome	CPF	Endereço
0101 0035 0987	João José Pedro	123456 234567 567890	Recife São Paulo Recife

Num-voo	. . .	Num-pil	. . .
. . 330 . . .	. . .	0101	. . .

Num-cad	Nome	CPF	Endereço	Num-voo	. . .	Num-pil	. . .
0101	João	123456	Recife	330	. . .	0101	. . .

Teoria das dependências Normalização

- No projeto de um banco de dados devemos:
  - Identificar dados
  - Fazer com estes dados representem eficientemente o mundo real
- Como proceder? Por intuição?

Decomposição **Modelo relacional** **Normalização**

☞ **O processo de identificar e estruturar dados**

Normalização

- Normalização
  - Método que permite identificar a existência de problemas potenciais (anomalias de atualização) no projeto de um BD relacional
  - Converte progressivamente uma tabela em tabelas de grau menor até que pouca ou nenhuma redundância de dados exista

Normalização

- Se a normalização é bem sucedida:
  - O espaço de armazenamento dos dados diminui
  - A tabela pode ser atualizada com maior eficiência
  - A descrição do BD será imediata

Teoria das Dependências

- Dependência Funcional

  Sejam R(A1, A2, ..., An) e X, Y contidos em {A1,A2,..., An} diz-se que existe uma Dependência Funcional (DF) de X para Y ($X \rightarrow Y$) se somente se, em R, a um valor de X corresponde um e um só valor de Y.

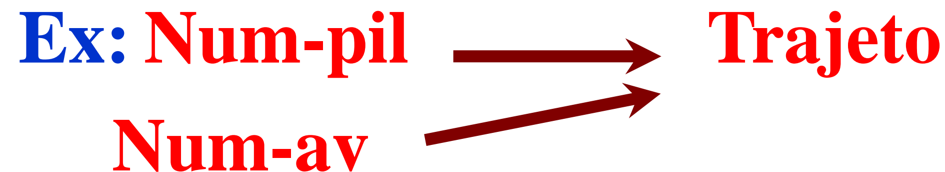
  Ex: Num-cad $\rightarrow$ Nome

- DF Total: Se $X \rightarrow Y$ e $Y \rightarrow X$ ($X \leftrightarrow Y$)

  Ex: Num-cad $\leftrightarrow$ CPF

Teoria das Dependências

- DF Plena: quando um atributo é dependente de dois (ou mais) outros.



Teoria das Dependências

- Chave Primária

  Um atributo A (ou uma coleção de atributos) é a chave primária de uma relação R, se:

  - 1. Todos os atributos de R são funcionalmente dependentes de A
  - 2. Nenhum subconjunto de atributos de A também tem a propriedade 1.

Normalização

- Definição:

  Uma relação está na Primeira Forma Normal (1NF) se todos os atributos que a compõem são atômicos(simples e indivisíveis).

Piloto

Num-cad	Nome	CPF	Salário	Diploma	Descrição
0010	José	123456	5.000,00	D1	Helicópteros
0010	José	123456	5.000,00	D2	Aviões a jato
0015	João	234567	3.000,00	D3	Bi-motor
0020	Manuel	345678	8.000,00	D1	Helicópteros
0020	Manuel	345678	8.000,00	D2	Aviões a jato
0020	Manuel	345678	8.000,00	D4	Concorde
0018	José	987654	4.000,00	D2	Aviões a jato

Normalização

- Conseqüências da Normalização:
  - Extensão da chave primária
  - Dependência Funcional de parte da chave primária
  - Anomalias de atualização:
    - atualização: de todas as tuplas com mesmo valor de atributo

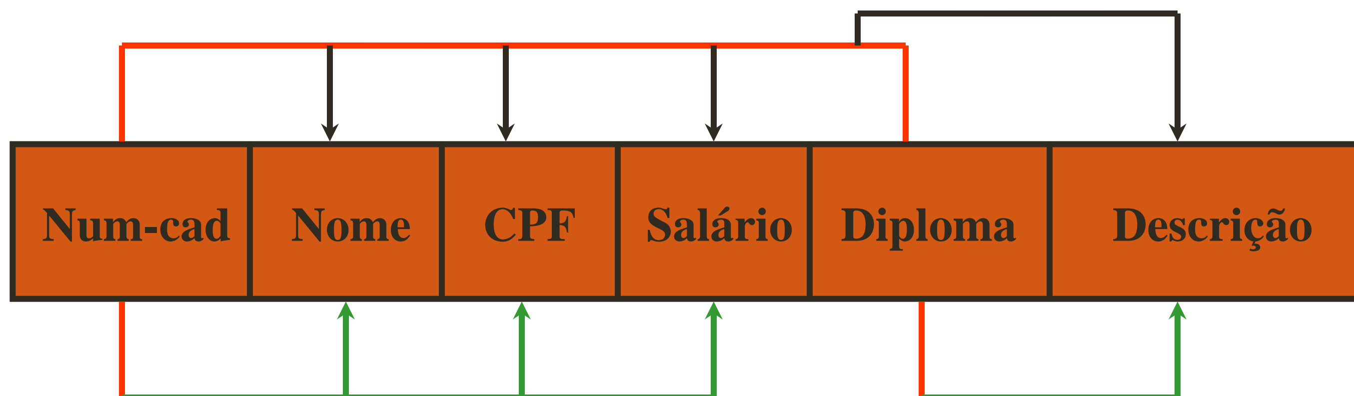
Normalização

- Anomalias (Cont.)
  - inconsistência: se a atualização não for feita em todos os níveis
  - inclusões: de um item que não tem correspondente para os outros campos da chave primária
  - remoções: de um item da chave provoca a remoção de informações adicionais

Normalização

- Definição:

  Uma relação está na Segunda Forma Normal (2NF) se ela está na 1NF e todo atributo não-chave é plenamente dependente da chave primária.



Normalização

Como corrigir:

1. Para cada subconjunto de atributos que compõem a chave primária, criar uma relação com este subconjunto como chave primária
2. Colocar cada um dos outros atributos com o subconjunto mínimo do qual ele depende

Normalização

Relações criadas:

(Num-cad, Nome, CPF, Salário)

(Diploma, Descrição)

(Num-cad, Diploma)

3. Dar nome às novas relações, por exemplo: Piloto, Diploma e Formação (respectivamente).

Normalização

Piloto

Num-cad	**Nome**	**CPF**	**Salário**
0010	José	123456	5.000,00
0015	João	234567	3.000,00
0020	Manuel	345678	8.000,00
0018	José	987654	4.000,00

Formação

Num-cad	**Diploma**
0010	D1
0010	D2
0015	D3
0020	D1
0020	D2
0020	D4
0018	D2

Diploma

Diploma	**Descrição**
D1	Helicópteros
D2	Aviões a jato
D3	Bi-motor
D4	Concorde

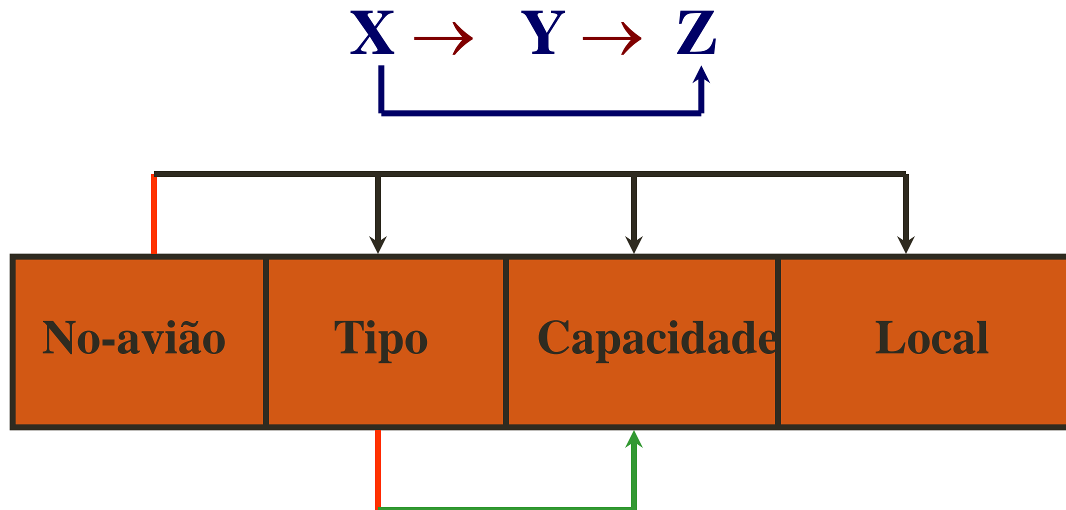
OBS: Anomalias foram eliminadas - Não houve perda de informação

Teoria das Dependências

- Dependência Transitiva:

  Ocorre quando Y depende de X e Z depende de Y.

  Logo, Z também depende de X.



Normalização

- Definição:

  Uma relação está na Terceira Forma Normal (3NF) se ela está na 2NF e nenhum atributo não-chave é transitivamente dependente da chave primária.

- Definição:

  Uma relação está na Forma Normal de Boyce/Codd (BCNF) se todo determinante é uma chave candidata.

Normalização

- Como corrigir?

  Para cada determinante que não é chave candidata, remover da relação os atributos que dependem dele para criar uma nova relação onde o determinante será chave primária.

Normalização

Avião

No-av	Tipo	Capacidade	Local
101	A320	320	Rio
104	B727	250	S.Paulo
105	DC10	350	Rio
103	B727	250	Recife
110	B727	250	Rio

Avião1

No-av	Tipo	Local
101	A320	Rio
104	B727	S.Paulo
105	DC10	Rio
103	B727	Recife
110	B727	Rio

Tipo-av

Tipo	Capacidade
A320	320
B727	250
DC10	350

Normalização

- Relações com mais de uma chave candidata

Considere a relação:

ADP

Aluno	Disc	Prof.
Maria	BD	Fernando
Maria	ES	Paulo
José	BD	Fernando
José	ES	André

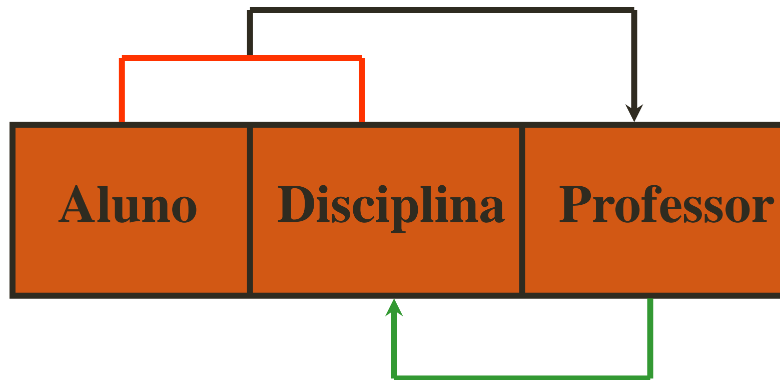
e as regras:

- para cada disciplina, cada estudante tem um único professor

- cada professor ensina uma única disciplina

- cada disciplina é ensinada por vários professores

Normalização



A decomposição seria:

AP

Aluno	Professor
Maria	Fernando
Maria	Paulo
José	Fernando
José	André

PD

Professor	Disciplina
Fernando	BD
Paulo	ES
André	ES

Indexação e Hashing

Introdução

Mecanismos de indexação são utilizados para acelerar o acesso aos dados desejados.

Um arquivo de índice é utilizado neste processo. Consiste de registros chamados de entradas de índice na forma:

CHAVE BUSCA	PONTEIRO

Chave de Busca – atributo ou conjunto de atributos usados para procurar registros em um arquivo.

Arquivos de índices são tipicamente muito menores do que o arquivo original.

Há Dois tipos básicos de índices:

- **Índices Ordenados**: as chaves de busca são armazenadas de forma ordenada
- **Índices Hash**: as chaves de busca são distribuídas uniformemente em *buckets* e acessadas através de uma função *hash*.

Como Avaliar a Utilização de Índices

Suporte eficiente aos tipos de acesso. O acesso pode dar-se tanto para:

- registros com um certo valor de atributo
- ou registros com um valor de atributo dentro de uma faixa específica de valores.

Tempo de acesso;

Tempo de inserção;

Tempo de exclusão;

Overhead de espaço;

Índices Ordenados

Em um índice ordenado, as entradas de índice são armazenadas ordenadamente pelo valor da chave de busca.

Índice Primário: em um arquivo ordenado seqüencialmente, o índice cuja chave de busca especifica a ordem seqüencial do arquivo.

- Também chamado *clustering index*
- A chave de busca de um índice primário é normalmente (mas não necessariamente) a chave primária.

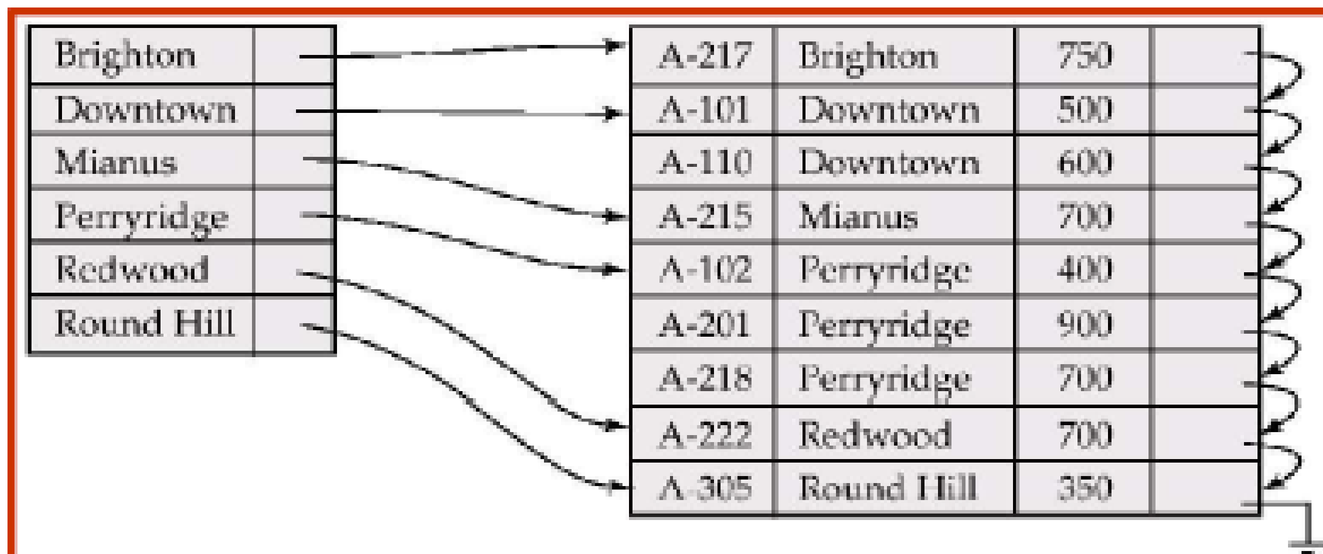
Índice Secundário: um índice cuja chave de busca especifica uma ordem diferente da ordem seqüencial do arquivo. Também chamado *non-clustering index*.

Um arquivo de índice seqüencial é tão somente um arquivo seqüencial ordenado com um índice primário.

Índices Ordenados *(continuação)*

Há dois tipos de índices ordenados que podem ser utilizados:

Índice Denso — existe um registro de índice para cada valor da chave de busca no arquivo.



Índices Ordenados *(continuação)*

Índice Esparso: contém registros de índice de apenas alguns valores da chave de busca.

- Aplicável somente quando os registros são ordenados seqüencialmente pela chave de busca

Para localizar um registro com valor de chave de busca *K*:

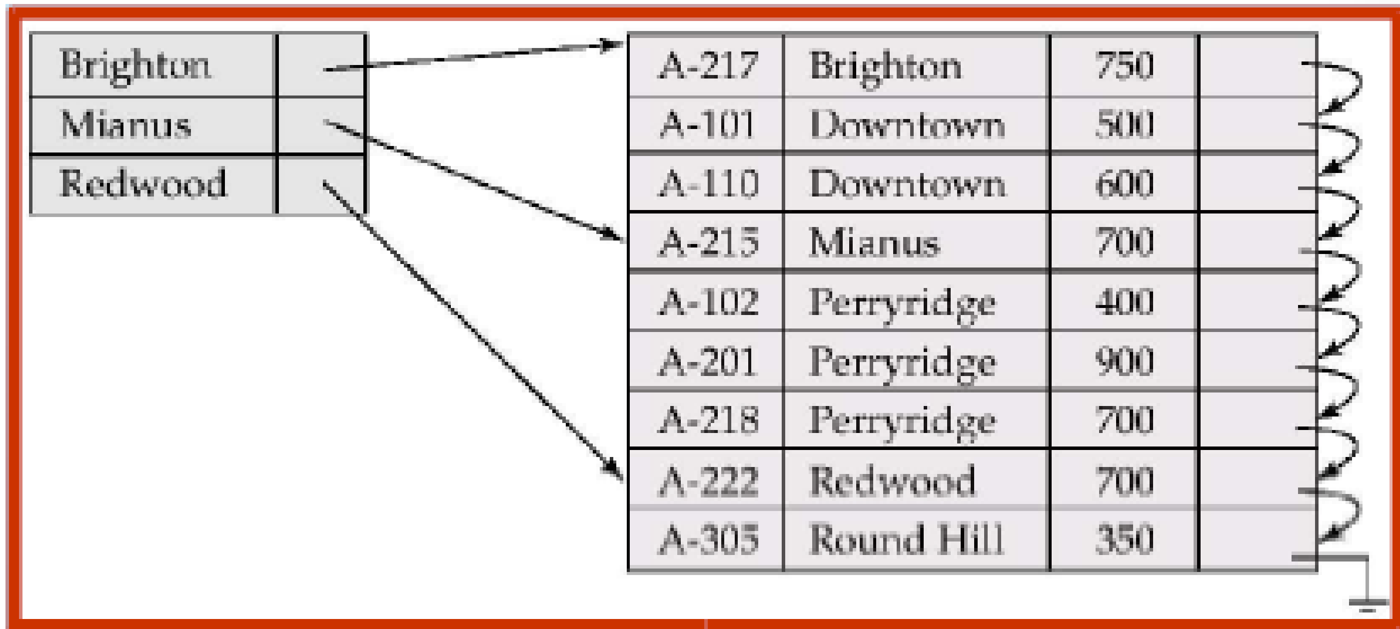
- Buscar o registro de índice com maior valor da chave de busca < *K*
- Pesquisar o arquivo seqüencialmente iniciando naquele registro para o qual o registro de índice aponta

Possui menor overhead de espaço e de manutenção para inserções e exclusões.

Em geral, mais lento do que o índice denso para localizar registros.

Índices Ordenados *(continuação)*

Exemplo da estrutura de índices esparsos:



Índices Ordenados *(continuação)*

Índices de Níveis Múltiplos: São índices como dois ou mais níveis utilizadas para otimizar o armazenamento dos arquivos de índices.

Se o índice primário não cabe na memória, o acesso se torna caro.

Para reduzir o número de acessos de disco para os registros de índice, tratar a chave primária mantida em disco como um arquivo seqüencial e construir um registro esparso sobre ela.

índice externo – um índice esparso do índice primário

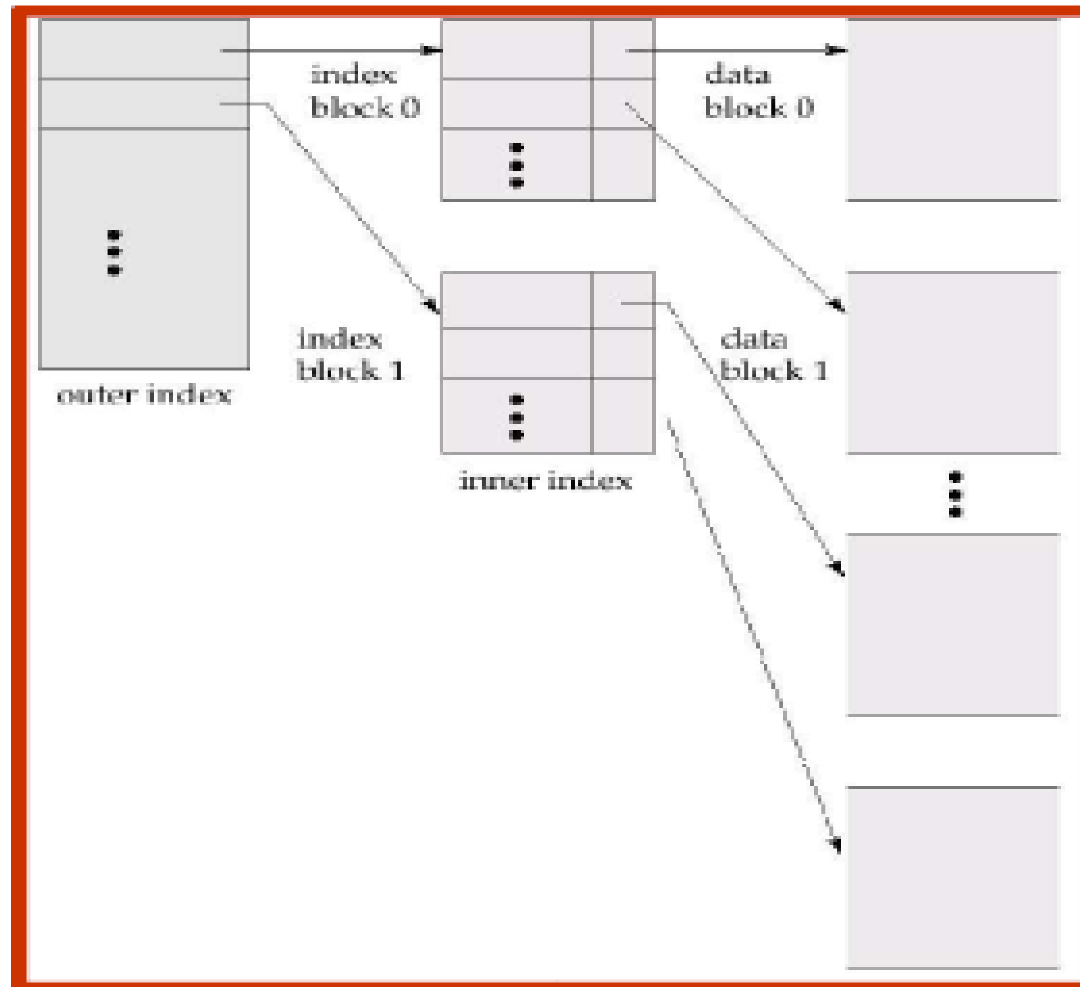
índice interno – o arquivo de índice primário

Se mesmo o índice externo for muito grande para caber na memória principal, outro nível de índice pode ser criado, e assim por diante.

Os índices em todos os níveis devem ser atualizados quando houver inserção ou remoção neste arquivo.

Índices Ordenados *(continuação)*

Exemplo da estrutura de índices esparsos de dois níveis:



Atualização de Índices

Exclusão

Se o registro excluído era o único registro no arquivo com o seu valor particular de chave de busca, a chave de busca é também removida do índice.

Exclusão em índice de nível único:

> **Índices densos** – a exclusão da chave de busca é similar à exclusão do registro do arquivo.
>
> **Índices esparsos** – se existir no índice uma entrada para a chave de busca, ela será excluída pela substituição da entrada no índice com o próximo valor de chave de busca do arquivo (na ordem da chave de busca). Se o próximo valor da chave de busca já tiver uma entrada de índice, a entrada será excluída ao invés de ser substituída.

Atualização de Índices *(continuação)*

Inserção

Inserção em índice de nível único:

Executa uma busca usando o valor da chave de busca do registro a ser inserido.

Índices densos – se o valor da chave de busca não aparecer no índice, deve-se inseri-lo.

Índices esparsos – se o índice guarda uma entrada para cada bloco do arquivo, não será necessário mudar o índice, a menos que um novo bloco seja criado. Neste caso, o primeiro valor de chave de busca que aparece no novo bloco será inserido no índice.

Os algoritmos de inserção multi-nível (e também os de exclusão) são simples extensões dos algoritmos de nível único

Índices Secundários

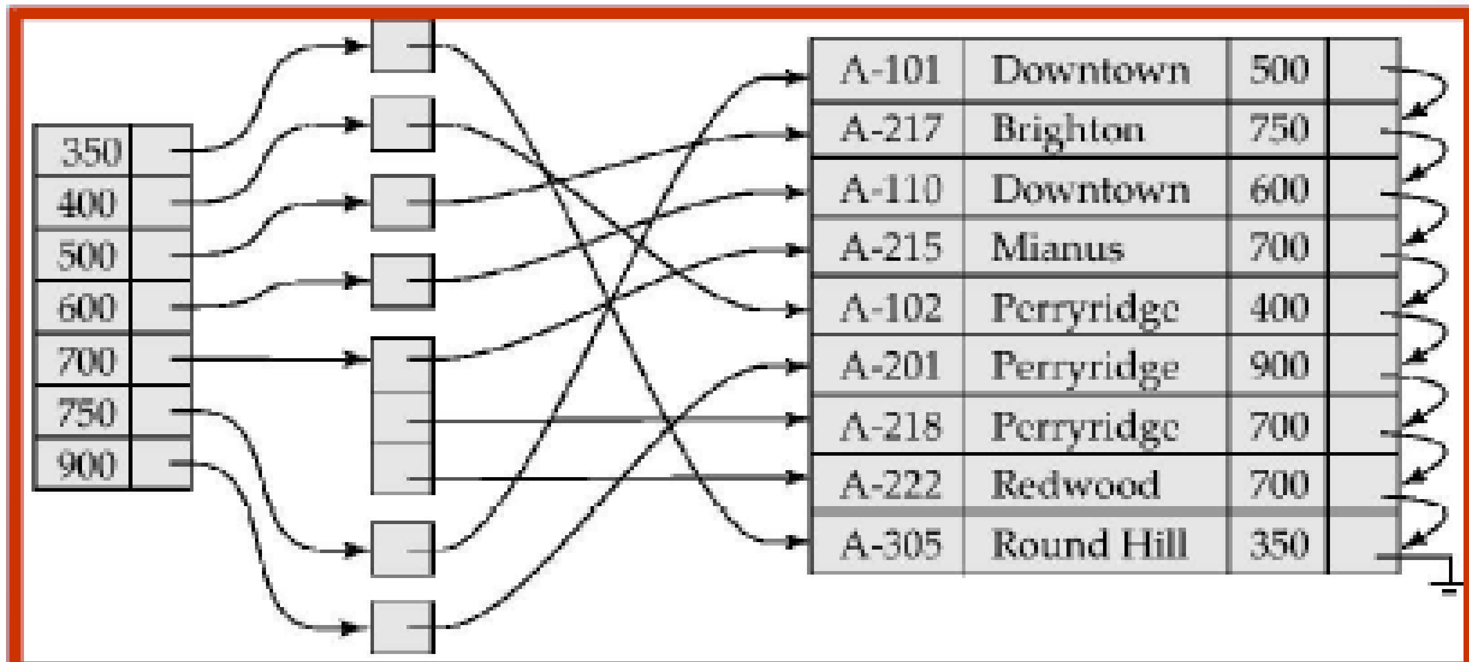
Freqüentemente, deseja-se buscar todos os registros cujos valores em um determinado campo (que não é a chave de busca) satisfazem alguma condição.

> Exemplo 1: em um BD de *contas* armazenado seqüencialmente pelo número da conta, deseja-se encontrar todas as contas de uma determinada agência
>
> Exemplo 2: mesmo que acima, mas onde se quer buscar todas as contas com um determinado saldo ou faixa de saldos

Pode-se ter um índice secundário com um registro de índice para cada valor de chave de busca. Os registros de índice apontam para um *bucket* que contém ponteiros para todos os registros reais que possuem aquele valor específico de chave de busca.

Índices Secundários *(continuação)*

Exemplo da estrutura de índices esparsos de dois níveis:



Índices Primários e Secundários

Índices secundários têm que ser densos.

Índices oferecem benefícios substanciais na busca por registros.

Quando um arquivo é modificado, cada índice no arquivo deve ser atualizado.

A atualização de índices impõe overhead sobre a modificação do banco de dados.

Buscas seqüenciais usando a chave primária são eficientes, mas buscas seqüenciais usando um índice secundário são caras

- Cada acesso a registro, por exemplo, pode buscar um novo bloco do disco

Arquivo de Índices B^+-Tree

Índices B^+-tree são uma alternativa para arquivos indexados seqüencialmente.

Desvantagem de arquivos indexados seqüencialmente: degradação de desempenho à medida em que o arquivo cresce, uma vez que muitos blocos de *overflow* são criados. É necessária a reorganização periódica do arquivo inteiro.

Vantagem dos arquivos de índice B^+-tree: reorganiza-se automaticamente face a inserções e exclusões, com mudanças pequenas e locais. Não é necessária a reorganização do arquivo inteiro para manter desempenho.

Desvantagem das B^+-trees: overhead extra para inserções e exclusões, overhead de espaço.

Vantagens das B^+-trees se sobrepõem às desvantagens, e elas são usadas extensivamente.

Estrutura de Nó de uma B^+-Tree

Nó típico:

Ki são os valores da chave de busca

Pi são ponteiros para os filhos (para nodos não-folha) ou ponteiros para registros ou *buckets* de registros (para nodos folha).

As chaves de busca em um nodo são ordenadas

$$K1 < K2 < K3 < . . . < Kn-1$$

Hashing Estático

Um *bucket* é uma unidade de armazenamento contendo um ou mais registros (um *bucket* é tipicamente um arquivo de disco).

Em uma organização de arquivo *hash* nós obtemos o *bucket* de um registro diretamente do seu valor de chave de busca usando uma função *hash*.

Uma função *hash h* é uma função do conjunto de todos os valores de chave de busca *K* para o conjunto de todos os endereços de *bucket B.*

Uma função *hash* é usada para localizar registros para acesso, inserção ou remoção.

Registros com valores diferentes de chave de busca podem ser mapeados para o mesmo *bucket*; assim, o *bucket* inteiro tem que ser varrido seqüencialmente para localizar um registro.

Hashing Estático *(continuação)*

Organização hash do arquivo *conta*, usando *nome-agencia* como chave, conforme figura a ser apresentada no próximo slide.

Há 10 buckets na representação.

A representação binária do *i*-ésimo caracter é assumida como sendo o inteiro *i.*

A função hash retorna a soma das representações binárias dos caracteres módulo 10.

Exemplo:

h(Perryridge) = 5

h(Round Hill) = 3

h(Brighton) = 3

Hashing Estático ***(continuação)***

Exemplo da organização de um arquivo hash.

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

A Função *Hash*

A pior função *hash* mapeia todos os valores de chave de busca para o mesmo *bucket*; isso torna o tempo de acesso proporcional ao número de valores de chave de busca no arquivo.

Uma função *hash* ideal é uniforme, ou seja, a cada *bucket* é atribuído o mesmo número de valores de chave de busca a partir do conjunto de todos os valores possíveis.

Uma função *hash* ideal é randômica, de tal forma que cada *bucket* terá o mesmo número de registros atribuídos a ele, independentemente da distribuição real de valores de chave de busca no arquivo.

Funções de *hash* típicas executam cálculos sobre a representação binária interna da chave de busca.

> Por exemplo, para uma chave de busca do tipo string, as representações binárias de todos os caracteres na string podem ser adicionadas, e o módulo da soma do número de *buckets* pode ser retornado.

Gerenciamento de *Overflow* de *Bucket*

Overflow de *Bucket* pode ocorrer por causa de:

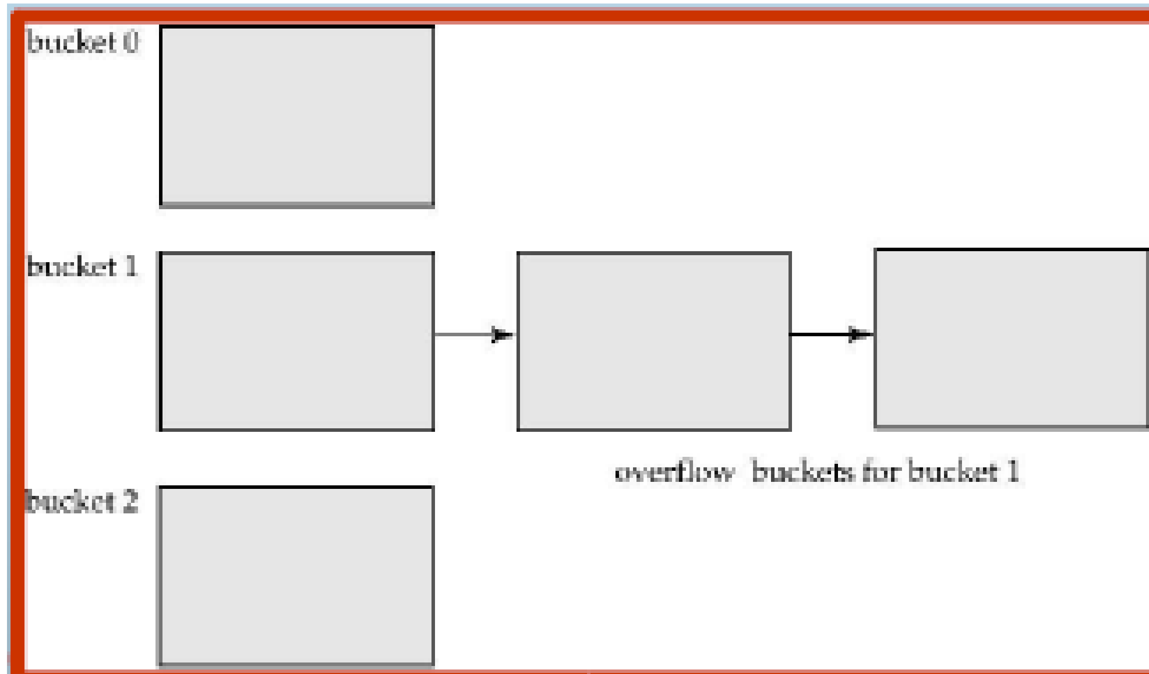
- *Buckets* insuficientes
- Desvios na distribuição de registros. Isso pode ocorrer devido a duas razões:
  - Múltiplos registros têm o mesmo valor de chave de busca
  - A função de *hash* escolhida produz distribuições não uniformes de valores de chave

Embora a probabilidade de *overflow* de *bucket* poder ser reduzida, ela não pode ser eliminada

Este problema é gerenciado pelo uso de *overflow buckets.*

Gerenciamento de *Overflow* de *Bucket* *(continuação)*

Cadeia de *Overflow* – os *overflow buckets* de um dado *bucket* são encadeados em uma lista ligada.



Índice *Hash*

Hashing pode ser usado não apenas para a organização do arquivo, mas também para a criação da estrutura de índices.

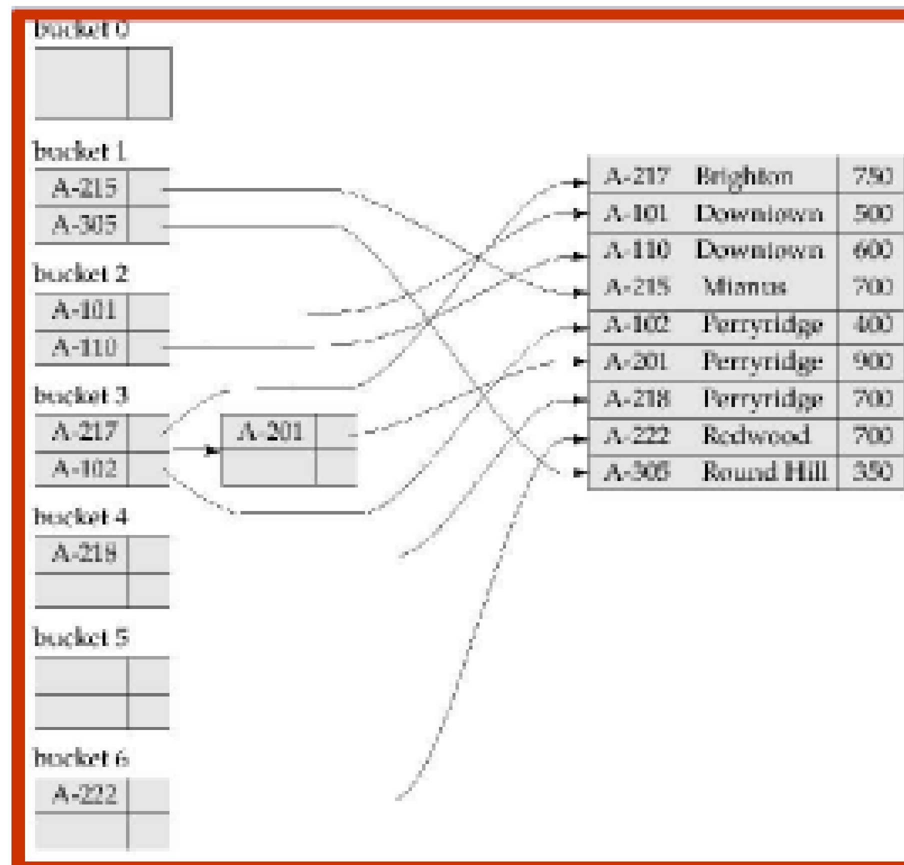
Um índice *hash* organiza as chaves de busca com os seus ponteiros de registro associados em uma estrutura de arquivos hash.

Estritamente falando, índices *hash* são sempre índices secundários:

> se o próprio arquivo é organizado usando *hashing*, um índice primário separado usando a mesma chave de busca é desnecessário.
>
> Entretanto, usa-se o termo índice *hash* para referir tanto a estruturas de índice secundárias e arquivos organizados usando *hashing*.

Índice *Hash* *(continuação)*

Exemplo de índice *Hash:*



Deficiências do *Hashing* Estático

No *hashing* estático, a função *h* mapeia os valores da chave de busca para um conjunto fixo *B* de endereços de bucket.

- Bancos de dados crescem com o tempo. Se o número inicial de *buckets* for muito pequeno, a performance será degradada devido a muitos *overflows*.
- Se o tamanho do arquivo em algum ponto no futuro for antecipado e o número de *buckets* alocado de acordo, uma quantidade significativa de espaço será desperdiçada inicialmente.
- Se o banco de dados encolhe, novamente haverá desperdício de espaço.
- Uma opção é a reorganização periódica do arquivo com uma nova função *hash*, mas isso é muito caro.

Estes problemas podem ser evitados pelo uso de técnicas que permitem que o número de *buckets* seja modificado dinamicamente.

Deficiências do *Hashing* Estático

No *hashing* estático, a função *h* mapeia os valores da chave de busca para um conjunto fixo *B* de endereços de bucket.

- Bancos de dados crescem com o tempo. Se o número inicial de *buckets* for muito pequeno, a performance será degradada devido a muitos *overflows*.
- Se o tamanho do arquivo em algum ponto no futuro for antecipado e o número de *buckets* alocado de acordo, uma quantidade significativa de espaço será desperdiçada inicialmente.
- Se o banco de dados encolhe, novamente haverá desperdício de espaço.
- Uma opção é a reorganização periódica do arquivo com uma nova função *hash*, mas isso é muito caro.

Estes problemas podem ser evitados pelo uso de técnicas que permitem que o número de *buckets* seja modificado dinamicamente.

Definição de Índices em SQL

Um índice é criado através da seguinte instrução

create index <index-name> or <relation-name> (<attribute-list>)

Para apagar um índice usa-se:

drop index <index-name>

Acesso por Chave Múltipla

Uso de índices múltiplos para certos tipos de consulta.Exemplo:

select *account-number*
from *account*
where *branch-name* = "Perryridge" **and** *balance* =1000

Possíveis estratégias para processar a consulta usando índices sobre atributos únicos:

1. Usar índice sobre *branch-name* para encontrar todos os registros pertencente à agência "Perryridge" e examinar cada um dos registros para verificar se saldo é igual a $1000.

2. Usar índice sobre *balance* para encontrar contas com saldos de $1000; testart *branch-name* = "Perryridge".

3. Usar índice *branch-name* para encontrar ponteiros para todos os registros pertencentes à agência Perryridge. De maneira semelhante, usar índices sobre *balance*. Tomar a interseção de ambos os conjuntos de ponteiros obtidos.