Otimização de Consultas em MySQL

BIANCA PEDROSA
2014

Otimização de Consultas

- ✓ Técnicas para melhorar desempenho do processamento de consultas em BD.
- ✓ Podem ser baseadas em:
  - Manipulações Algébricas
  - Uso de Índices
  - Estimativas baseadas em custos
- ✓ Neste curso veremos estratégias de otimização baseadas em índices!

Uso de Índices

- ✓ Indexação é a ferramenta mais importante para ter aumento de performance em consultas. Quando uma consulta demora a ser concluída, normalmente as tabelas envolvidas não possuem índices ou estes são mal criados.
- ✓ A adequação ou criação dos índices necessários resolve o problema na grande maioria das vezes. Claro que a criação de índices nem sempre resolve, pois a otimização de banco de dados nem sempre é simples.
- ✓ Entretanto, se você não usar índices, em muitos casos, estará desperdiçando seu tempo tentando aumentar a performance por outros meios. Use indexação como primeira alternativa para ter um ganho de performance e depois avalie que outras técnicas podem ser úteis.

Benefícios do uso de Índices

- ✓ Quando uma tabela não tem índices, os seus registros são desordenados e uma consulta terá que percorrer todos os registros. Se adicionarmos um índice, uma nova tabela é gerada.
- ✓ A quantidade de registros da tabela de índices é a mesma para a tabela original, com a diferença que os registros são ordenados. Isso implica que uma consulta "varre" a tabela para encontrar os registros que casem com a condição da consulta e a busca é cessada quando um valor imediatamente maior é encontrado.
- ✓ O uso de índices pode ainda ser mais valioso em consultas envolvendo joins ou múltiplas tabelas. Numa consulta em uma tabela, o número de valores que precisa ser examinado por coluna é o número de registros da tabela. Em consultas em múltiplas tabelas, o número de possíveis combinações cresce meteoricamente, já que resulta do produto do número de registros das tabelas.

Exemplo do Uso de Índices

- ✓ Suponha que existam três tabelas sem índices, t1, t2 e t3, cada uma contendo uma coluna, c1, c2 e c3, respectivamente, e cada coluna contendo 1.000 registros com dados de 1 a 1.000. A consulta para encontrar todas as combinações de registros que tenham o mesmo valor pode ser representada desta forma:

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

Exemplo do Uso de Índices

- ✓ O resultado dessa consulta deve ser uma tabela de 1.000 registros, cada um contendo três valores iguais. Se a consulta for executada sem o uso de índices, não teremos ideia de quais registros contêm quais dados sem percorrer todos eles. Conseqüentemente, o banco tenta todas as combinações possíveis para encontrar os registros que combinam com a condição da cláusula WHERE. O número de possíveis combinações é 1.000 x 1.000 x 1.000 = 1.000.000.000 (um bilhão!), que é um milhão de vezes maior do que o número de registros que combinaram com a condição. Ou seja, foi muito esforço desperdiçado.

MySQL usa índices de muitas maneiras:

- ✓ Aumentar a velocidade de pesquisa de registros que combinem com a condição da cláusula WHERE ou linhas que combinam com linhas de outras tabelas quando joins são usados.
- ✓ Para consultas que usam as funções MIN() e MAX(), o menor e o maior valores numa coluna indexada podem ser encontrados rapidamente.
- ✓ Realizar operações de ordenação e agrupamento rapidamente usando as cláusulas ORDER BY e GROUP BY.
- ✓ Ler todas as informações requeridas numa consulta. Isso acontece quando os campos retornados na consulta são os campos indexados na tabela.

Custos de Índices

- Índices aumentam a velocidade de consultas, mas diminuem a velocidade de inserções, atualizações e deleções. Isto é, índices tornam mais lentas as operações de escrita. Isso ocorre porque escrever um registro requer não apenas escrevê-lo, mas também mudar a ordenação dos índices. Quanto mais índices houver numa tabela, mais mudanças de ordenação necessitam ser feitas, degradando a performance.
- Um índice ocupa espaço de disco. Isso pode fazer a tabela exceder o seu tamanho limite mais rapidamente do que se não tivesse índices.

A Escolha de Índices 1/2

- ✓ Índices devem ser criados em colunas que usamos para pesquisa, ordenação ou agrupamento. Nunca em colunas que só são exibidas. Em outras palavras, as colunas candidatas são aquelas que aparecem na cláusula **WHERE**, **ORDER BY** ou **GROUP BY** e em **JOIN**s.
- ✓ O fato de uma coluna aparecer na lista de colunas que serão exibidas num SELECT não a descarta de ser uma coluna candidata, pois ela pode estar na listagem, mas também estar na cláusula WHERE, por exemplo.
- ✓ Colunas que aparecem em cláusulas join ou em expressões como col1 = col2 nas cláusulas WHERE são candidatas fortíssimas à criação de índice.

A Escolha de Índices 2/2

- ✓ Considere a cardinalidade da coluna. A cardinalidade da coluna é o número de valores distintos que ela contém. Índices funcionam melhor em colunas com um alto número de cardinalidade relativa ao número de registros da tabela, isto é, colunas que têm muitos valores únicos e poucos duplicados. Se uma coluna contém valores muito diferentes de idade, um índice irá diferenciar os registros rapidamente. Entretanto, não irá ajudar numa coluna que é usada para armazenar registros de gênero (sexo) e contém somente os valores 'M' ou 'F'.
- ✓ Crie índices com valores pequenos. Use tipos de dados o menor possível. Por exemplo, não use uma coluna BIGINT se MEDIUMINT suporta os dados que serão armazenados. Não use CHAR(100) se nenhum dos valores armazenados ultrapassa 25 caracteres. Valores pequenos melhoram o processamento de índices, pois podem ser comparados mais rapidamente e ocupam menos espaço de disco nos arquivos de índices.

A Escolha de Índices 2/2

- ✓ Considere a cardinalidade da coluna. A cardinalidade da coluna é o número de valores distintos que ela contém. Índices funcionam melhor em colunas com um alto número de cardinalidade relativa ao número de registros da tabela, isto é, colunas que têm muitos valores únicos e poucos duplicados. Se uma coluna contém valores muito diferentes de idade, um índice irá diferenciar os registros rapidamente. Entretanto, não irá ajudar numa coluna que é usada para armazenar registros de gênero (sexo) e contém somente os valores 'M' ou 'F'.
- ✓ Crie índices com valores pequenos. Use tipos de dados o menor possível. Por exemplo, não use uma coluna BIGINT se MEDIUMINT suporta os dados que serão armazenados. Não use CHAR(100) se nenhum dos valores armazenados ultrapassa 25 caracteres. Valores pequenos melhoram o processamento de índices, pois podem ser comparados mais rapidamente.

A Escolha dos Tipos de Dados

- ✓ Não use colunas grandes para dados pequenos. Se você está usando colunas de tamanho fixo, como CHAR, não especifique um tamanho grande desnecessariamente.
- ✓ Use colunas de tamanho fixo ao invés de colunas de tamanho variável, ou seja, prefira **CHAR** à **VARCHAR**. Isto ocupará mais espaço em disco, mas se você dispõe desse espaço extra, colunas de tamanho fixo são processadas mais rapidamente, especialmente para tabelas que recebem muitas modificações.
- ✓ Defina colunas como **NOT NULL**. Isto traz rapidez de processamento e requer menos armazenamento. Isto também pode simplificar consultas porque não é necessário checar por valores nulos.
- ✓ Considere o uso de colunas **ENUM**. Se você tem uma coluna string que terá baixa cardinalidade (número pequeno de valores distintos), considere o uso de colunas tipo ENUM. Colunas ENUM são processadas mais rapidamente, pois são representadas como numéricas, internamente.

A Escolha dos Tipos de Dados

- ✓ Use a função PROCEDURE ANALYSE(). Execute a função PROCEDURE ANALYSE() para verificar indicações de tipos de dados para as colunas de uma tabela.

SELECT * FROM tabela PROCEDURE ANALYSE();

- ✓ Serão sugeridos tipos de dados para cada coluna da tabela. Baseado na saída da função, você pode perceber que sua tabela precisa ser modificada para tirar proveito de tipos de dados mais eficientes.

A Escolha dos Tipos de Dados

- ✓ Use a função PROCEDURE ANALYSE(). Execute a função PROCEDURE ANALYSE() para verificar indicações de tipos de dados para as colunas de uma tabela.

  SELECT * FROM tabela PROCEDURE ANALYSE();

- ✓ Serão sugeridos tipos de dados para cada coluna da tabela. Baseado na saída da função, você pode perceber que sua tabela precisa ser modificada para tirar proveito de tipos de dados mais eficientes.

Optimize table

- Use a função **OPTIMIZE TABLE** para tabelas que são sujeitas à fragmentação. Tabelas que são modificadas com grande frequência, especialmente aquelas que possuem colunas de tamanho variável, são sujeitas à fragmentação. **OPTIMIZE TABLE** pode ser usada apenas em tabelas **MyISAM** e **BDB**, mas desfragmentam apenas as tabelas **MyISAM**. O método de desfragmentação que funciona com qualquer tipo de tabela é fazer um **dump** da tabela com **mysqldump** e excluir e recriar as tabelas.

OPTMIZE TABLE ALUNO

- Coloque colunas BLOB e TEXT numa tabela separada. Sob algumas circunstâncias, pode fazer sentido mover estas colunas para uma tabela secundária, se você pretende converter a tabela para registros de tamanho fixo nas outras colunas. Isto reduzirá a fragmentação na tabela primária e permitirá que você tire proveito dos benefícios de performance de tabelas de coluna de tamanho fixo. Isto também permite que você execute consultas SELECT * na tabela primária sem sobrecarregar o servidor com o grande tamanho dos campos BLOB ou TEXT.

O Otimizador de Consultas

- Quando executamos uma consulta que seleciona registros, o MySQL a analisa a fim de verificar se uma otimização pode ser feita para processar a consulta mais rapidamente. O Otimizador tira proveito de índices, obviamente, mas outras informações também são usadas. Por exemplo, se executamos a consulta seguinte, o MySQL a executará muito rápido, não importando o quão grande a tabela seja:

SELECT * FROM tabela WHERE 0;

- Neste caso, o MySQL verifica a cláusula WHERE, conclui que não há possibilidade de um registro combinar e não se preocupa em procurar na tabela. Podemos verificar isso pelo uso do comando EXPLAIN, que mostra informações sobre como a consulta será executada.

mysql> explain select ra,nome from aluno order by ra;

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aluno	index	*NULL*	PRIMARY	9	*NULL*	7	

Como o Otimizador funciona

- ✓ O Otimizador tem muitos objetivos, mas seus alvos preliminares são os usos de índices possíveis e o uso do índice mais restritivo para eliminar ao máximo número de registros na pesquisa.
- ✓ Depois disso, o objetivo na execução do comando SELECT é encontrar registros, não rejeitá-los. O exemplo mostra uma consulta que testa duas colunas, cada uma com um índice:

```
SELECT col3 FROM tabela
WHERE col1 = 'algum valor' AND col2 = 'algum outro valor';
```

- ✓ Suponhamos também que o teste na coluna col1 combine 900 registros, o teste da col2 combine 300 registros e, juntas, combinem 30 registros. Se pesquisarmos pela col1, 870 registros serão rejeitados. Se pesquisarmos pela col2, 270 registros serão rejeitados e menos processamento de disco será necessário. Como resultado, o otimizador testará a col2 primeiro.

Dicas de Otimização para MySQL

1. Comparar colunas que tenham o mesmo tipo de dado. Quando usamos colunas indexadas em comparações, devemos usar colunas de mesmo tipo. Tipos de dados idênticos trazem melhor ganho de performance do que tipos similares. Por exemplo, INT é diferente de BIGINT. CHAR(10) é considerado o mesmo que CHAR(10) ou VARCHAR(10), mas diferente de CHAR(12) ou VARCHAR(12). Se as colunas que estamos comparando tiverem tipos diferentes, podemos usar ALTER TABLE para modificar um deles para que os tipos combinem.

2. Mantenha colunas separadas em comparações. Se usamos uma coluna numa chamada de função ou como parte de um termo mais complexo na expressão aritmética, o MySQL não pode usar índices porque ele deve calcular o valor da expressão para cada registro. Algumas vezes, isto é inevitável, mas muitas vezes podemos reescrever a consulta para tornar a coluna separada. A cláusula WHERE seguinte ilustra como isso funciona. Elas são equivalentes aritmeticamente, mas muito diferentes em termos de otimização:

WHERE col < 4 / 2; (recomendado)
WHERE col * 2 < 4;

Dicas de Otimização em MySQL

3. Não use o caracter de porcentagem (%) no início de um padrão LIKE.
4. Auxilie o otimizador a fazer melhores estimativas de efetividade dos índices. Por padrão, quando comparamos valores em colunas indexadas com uma constante, o otimizador supõe que os valores chaves estão distribuídos uniformemente dentro do índice. O otimizador também fará uma rápida checagem do índice para estimar quantas entradas serão usadas para determinar se o índice será usado para comparações constantes.
5. Use EXPLAIN para verificar a operação do otimizador. O comando EXPLAIN nos mostra se os índices estão sendo usados. Esta informação é muito útil quando tentamos diferentes maneiras de escrever um comando ou checamos se a adição de índices fará a diferença na eficiência da execução da consulta.

Dicas de Otimização em MySQL

6. Dê dicas ao otimizador. Normalmente, o otimizador se considera livre para determinar a ordem na qual as tabelas serão percorridas para retornar os dados mais rapidamente. Na ocasião, o otimizador nem sempre fará a melhor escolha. Se observarmos que isto está acontecendo, podemos forçar a escolha do otimizador usando a palavra chave STRAIGHT_JOIN. Isto força que as tabelas serão ligadas na ordem chamada na cláusula FROM. Também podemos usar FORCE INDEX, USE INDEX ou IGNORE INDEX para mostrar quais índices preferir.

7. Tire proveito de áreas em que o otimizador é mais completo. O MySQL pode realizar joins e subconsultas, mas o suporte a subconsultas é mais recente. Conseqüentemente, o otimizador está mais ajustado para joins do que subconsultas, em alguns casos.

Dicas de Otimização em MySQL

8. Teste formas alternativas de consultas, mas realize mais de um teste. Quando testamos formas alternativas de consultas, devemos testar diversas vezes cada forma. Se testarmos duas alternativas somente uma vez cada, freqüentemente a segunda forma será mais eficiente pois as informações da primeira consulta ainda estão no cache e não é necessário uma nova leitura em disco. Também há influência do uso do servidor naquele determinado momento.

9. Evite a sobrecarga do uso de auto-conversão de tipos. O MySQL executa automaticamente a conversão de tipos (*casting*), mas se podemos evitar conversões, melhoramos a performance. Por exemplo, se a coluna col_num é tipo INT, cada uma das consultas abaixo retornará o mesmo resultado, mas a segunda consulta envolve conversão automática. Assim, a primeira consulta é mais eficiente.

```
SELECT * FROM tabela WHERE col_num = 4;

SELECT * FROM tabela WHERE col_num = '4';
```

Dicas de Otimização em MySQL

10. Evite DISTINCT quando desnecessário, pois causa uma operação de ordenação
11. Evite Consultas Aninhadas usando **IN, =ALL, =ANY, =SOME,** pois frequentemente índices não são usados em sub-consultas
12. Prefira condições de junção sobre colunas numéricas. Evite junções sobre cadeias de caracteres
13. Transforme a condição NOT em uma expressão afirmativa
14. Consultas com múltiplas condições de seleção conectadas por **OR** devem ser substituídas por uma união de consultas, se existir índice no atributo a ser utilizado

```
SELECT nome,salario,idade
FROM Empregado
Where idade>45 OR Salario<50000
```

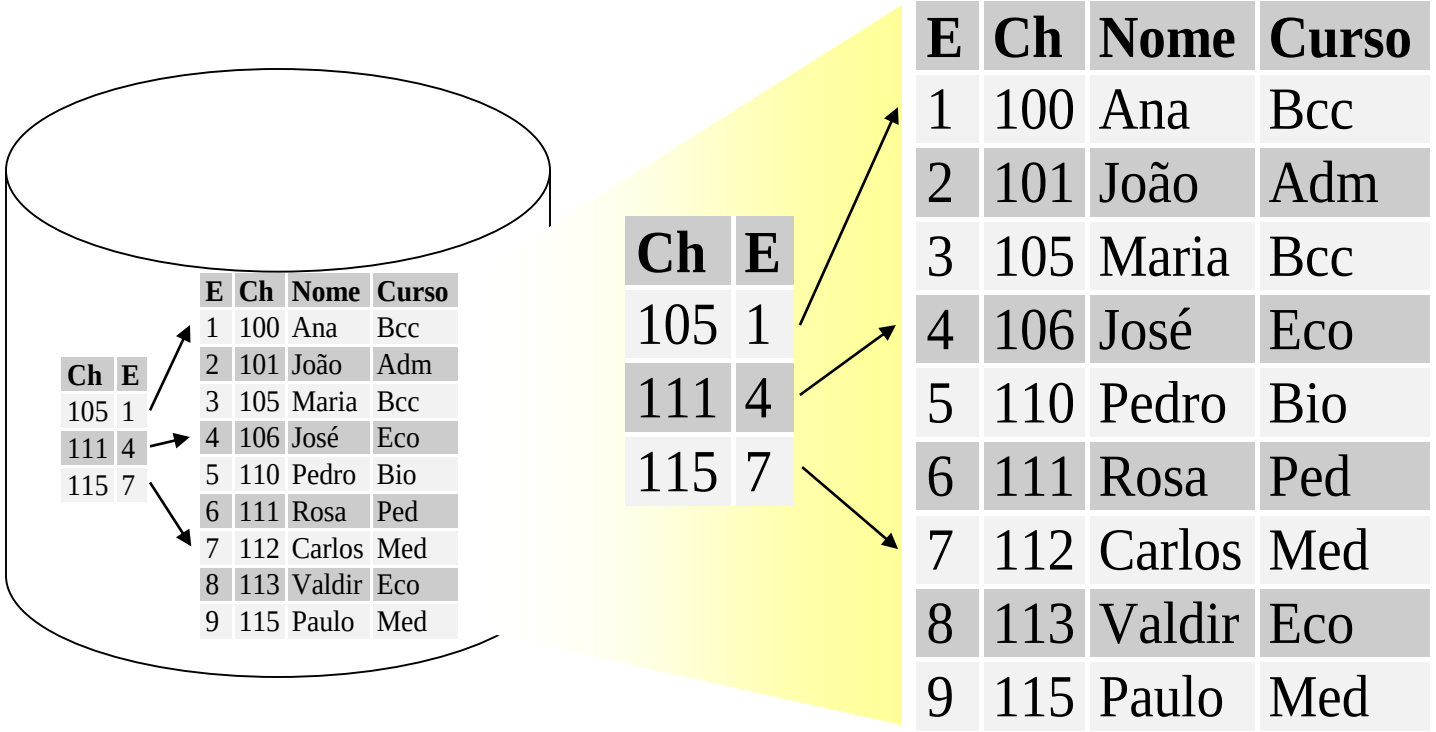
```
SELECT nome,salario,idade
FROM Empregado
Where idade>45      UNION
SELECT nome,salario,idade
FROM Empregado
Where Salario<50000
```

Indexação em BD

Índices

- ✓ O que são índices?
  - Estruturas adicionais que associamos ao arquivo de dados para localizar mais diretamente os registros desejados

Índices



Ch	E
105	1
111	4
115	7

E	Ch	Nome	Curso
1	100	Ana	Bcc
2	101	João	Adm
3	105	Maria	Bcc
4	106	José	Eco
5	110	Pedro	Bio
6	111	Rosa	Ped
7	112	Carlos	Med
8	113	Valdir	Eco
9	115	Paulo	Med

Ch	E
105	1
111	4
115	7

E	Ch	Nome	Curso
1	100	Ana	Bcc
2	101	João	Adm
3	105	Maria	Bcc
4	106	José	Eco
5	110	Pedro	Bio
6	111	Rosa	Ped
7	112	Carlos	Med
8	113	Valdir	Eco
9	115	Paulo	Med

Índices

- ✓ Tipos?
  - Ordenados
    - baseiam-se na ordenação de valores
  - Hash
    - baseiam-se na distribuição uniforme de valores por meio de uma faixa de buckets

Critérios para escolhas de Índices

- ✓ tempo de acesso
- ✓ tempo de inserção
- ✓ tempo de exclusão
- ✓ sobrecarga de espaço

Índices ordenados

- ✓ Vantagens:
  - Índices possuem registros de tamanho fixo
  - Índices são menores do que arquivo de dados.
  - Podem ser percorridos por **Busca Binária**

Algoritmo Busca Binária

```
inicio<- 1; fim <- nro_blocos;
Enquanto  ( inicio <= fim )
{         corrente <- ( inicio+ fim) div 2;
 se (k = campo_chave)
         então retorne campo_chave
 se (K< campo_chave)
         então fim  <- corrente – 1
 senão inicio <- corrente+1
}
```

Tipos de Índices ordenados

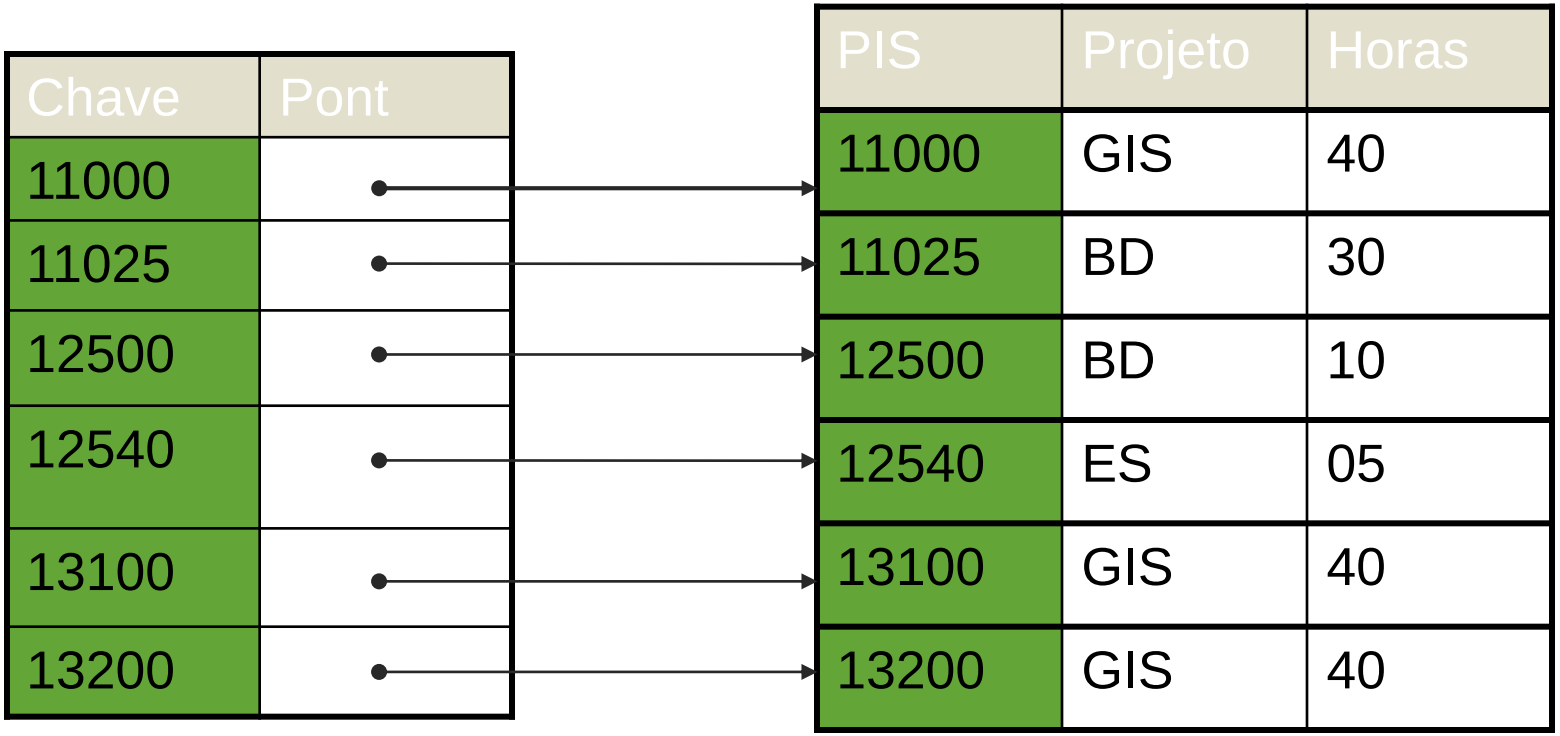
Densos

- Aplicam-se a arquivos não ordenados
- Possuem um registro para cada valor diferente no campo chave

Esparsos

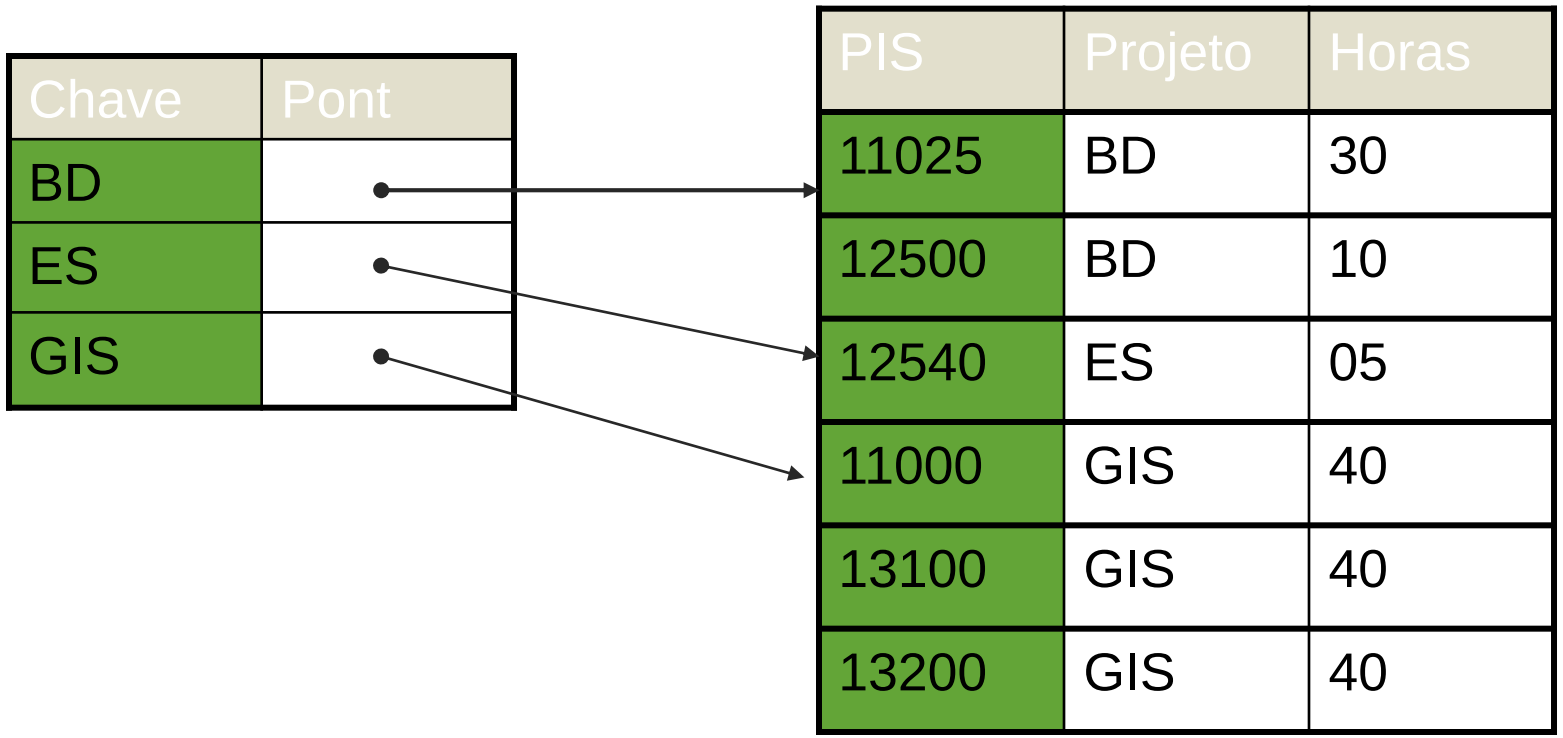
- Aplicam-se apenas a arquivos ordenados
- Possuem um registro para cada bloco

Índices Ordenados Densos



Existe um registro de índice para cada registro de dado.
Geralmente, índices densos são definidos sobre atributos chave.
O arquivo de dados não precisa estar ordenado.

Índices Ordenados Esparsos



Não existe um registro de índice para cada registro de dado.
Geralmente, índices esparsos são definidos sobre atributos não chave e requerem que o arquivo de dados esteja ordenado

Índices Primários

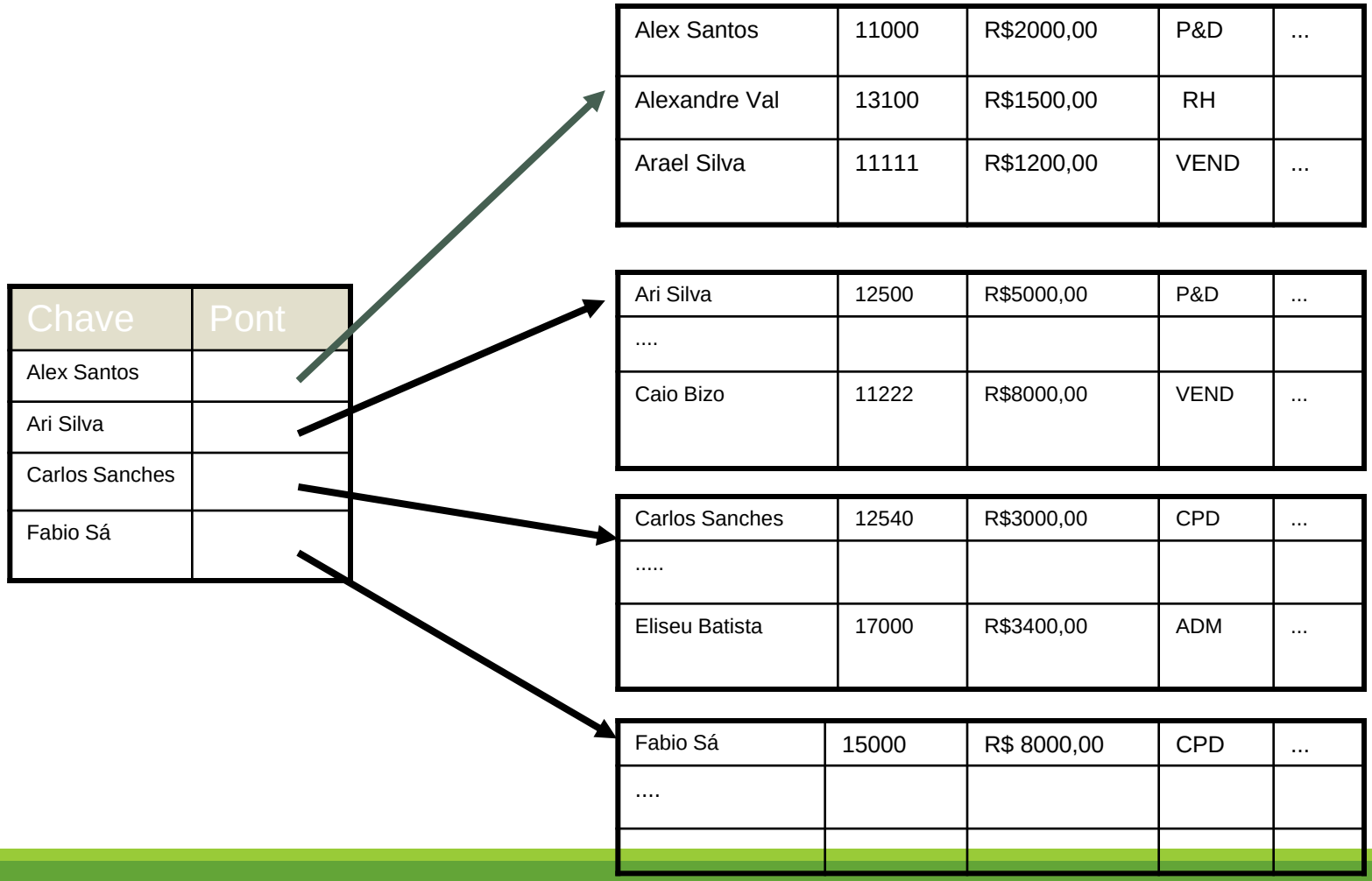
Definido sobre a chave primária de um arquivo de dados ordenado por esta mesma chave

Índice esparso (um registro p/ cada bloco)

Cada registro tem o valor do campo da chave primária do 1º registro do bloco e um ponteiro para este bloco.

Para inserir um registro na sua posição correta no arquivo de dados é necessário, além de mover os registros para abrir espaço para o novo registro, alterar alguns registros âncora (primeiro registro de um bloco)

Índices Primários



Se o arquivo de dados já está ordenado, para que utilizar um índice primário ?

Para que usar índice primário?

- ✓ Características do arquivo de dados:
  - Número total de registro = 30000
  - Tamanho do registro = 100 bytes
  - Bloco de disco = 1024 bytes
  - Fator de bloco = nro de registros existente em cada bloco de disco
    - FB= tam. do bloco/tam. do registro
    - FB= 1024/100 =10 registros/bloco
  - Número de blocos necessários para armazenar o arquivo todo
    - NB= número total de registros / fator de bloco
    - NB= 30000/10= 3000 blocos.
  - Busca binária = 12 acessos a blocos (log2 3000=12)

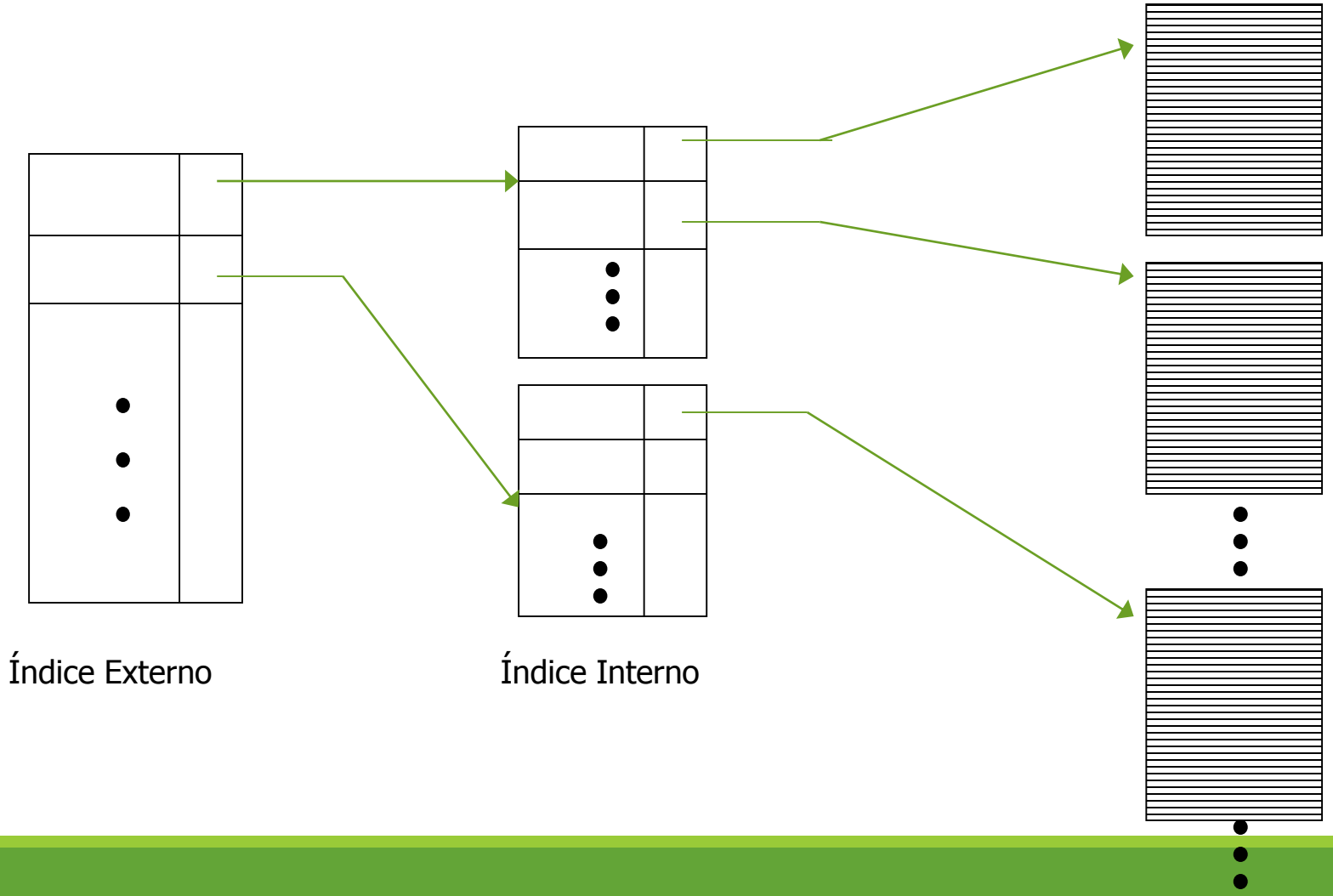
Para que usar índice primário?

- ✓ Características do arquivo de índice primário:
  - Número total de registros = 3000 (1 POR BLOCO)
  - Bloco de disco = 1024 bytes
  - Tamanho do registro do arquivo de índice = 15 bytes, por exemplo (campo chave de 9 bytes e ponteiro para bloco de disco de 6 bytes)
  - Fator de bloco do arquivo de índice é de 1024/15=68 registros/bloco.
  - Número de blocos necessários para armazenar o arquivo todo
  - Número total de registros / fator de bloco = 3000/68= 45 blocos
  - Busca binária = log 2 45 = 6 acessos a blocos

Vantagem de Índices Primários

- ✓ Redução expressiva do número de acessos a disco
- ✓ No exemplo anterior, sem a utilização do índice seriam necessários 12 acessos a bloco, com o índice é possível resolver com 7 acessos (6 acessos no índice e 1 no arquivo de dados).

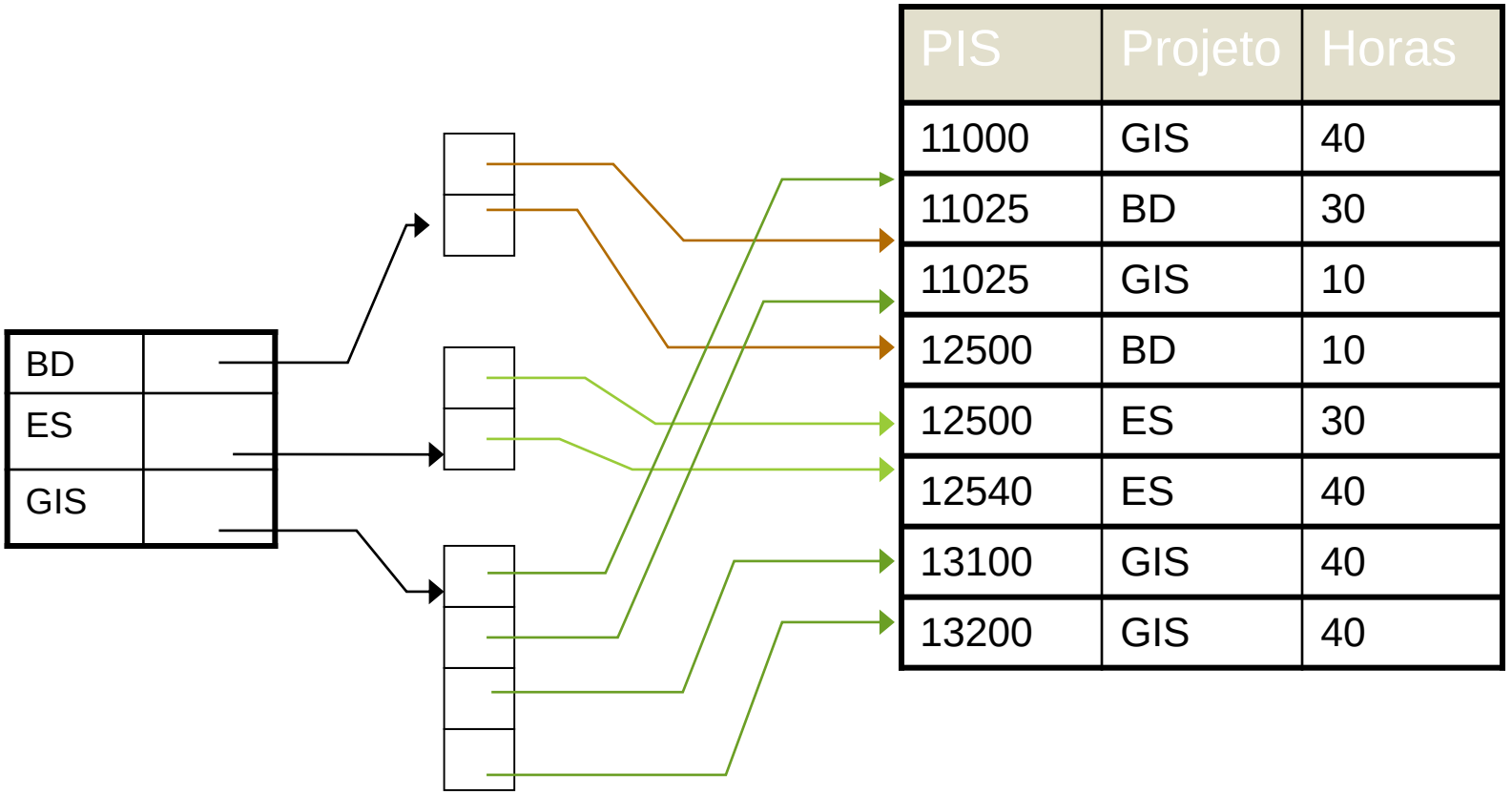
Índices Multi-níveis



Índices Secundários

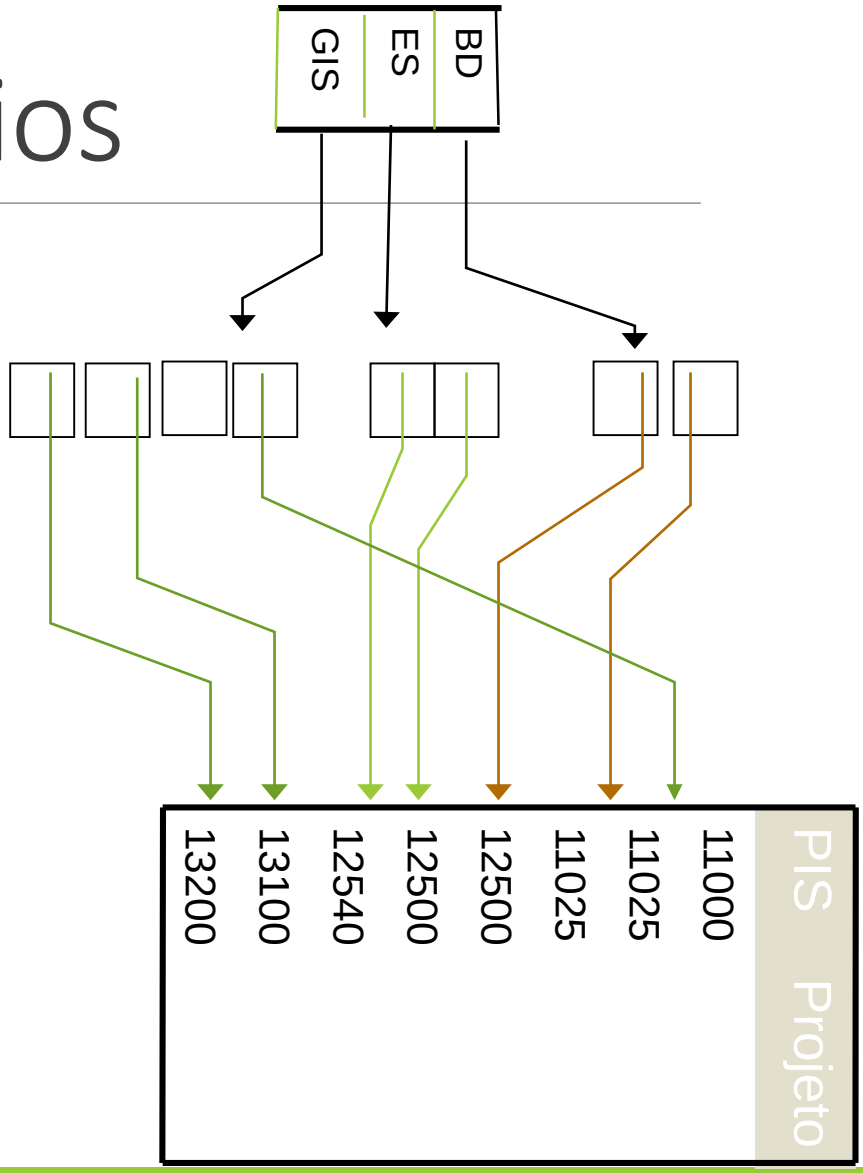
- ✓ composto por 2 campos:
  - 1º consiste em um campo não ordenado do arquivo de dados, chamado **campo de indexação**, e o
  - 2º consiste em um ponteiro para um bloco de disco ou registro.
- ✓ Se o índice secundário é definido em um **campo chave - chave secundária -** haverá um registro de índice para cada registro do arquivo, o que o caracterizará como **índice denso**.

Índices Secundários



PIS	Projeto	Horas
11000	GIS	40
11025	BD	30
11025	GIS	10
12500	BD	10
12500	ES	30
12540	ES	40
13100	GIS	40
13200	GIS	40

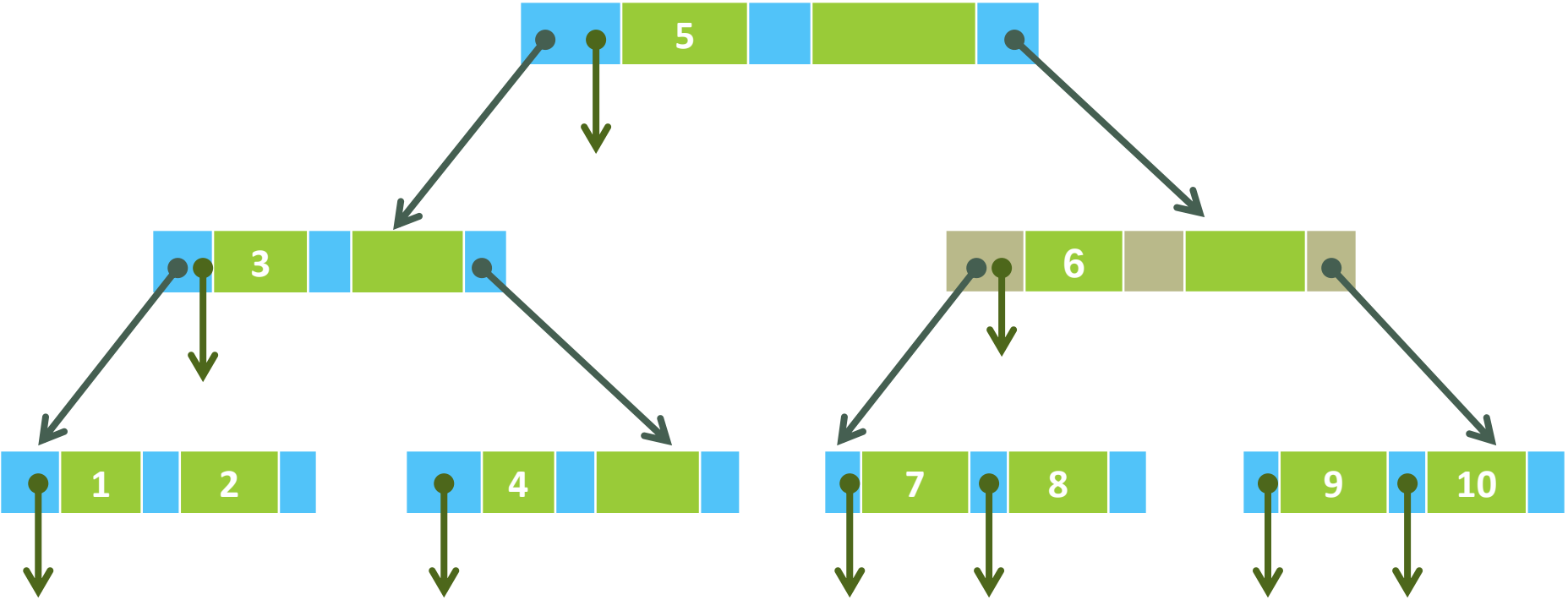
Índices Secundários



Implementação de Índices Secundários

ÁRVORES B

Árvore B



Ponteiro para dados

Ponteiro para nó de árvore

Árvore B

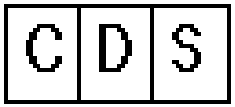
É uma árvore homogênea (nós iguais)

Todos os nós armazenam ponteiros para os dados

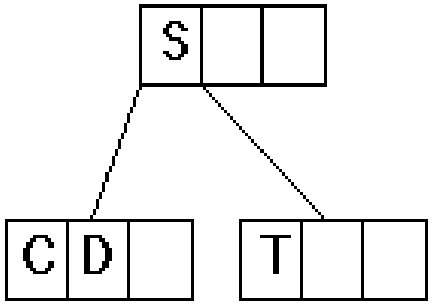
Valores Chaves aparecem 1 única vez

B-tree - Inserção

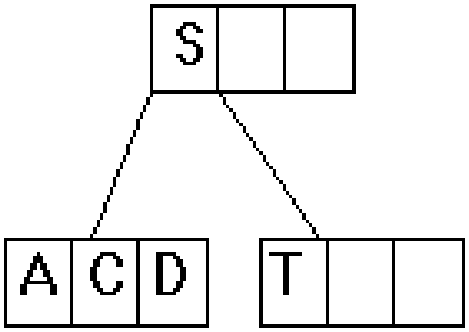
C S D T A M P I B W N G U R K E H O L J Y Q Z F X V



Inserção de C,S e D dentro da página inicial.)



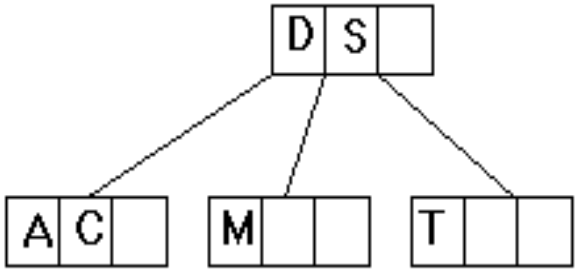
Inserção de T força o *split* e o *promotion* de S.



Adição de A.

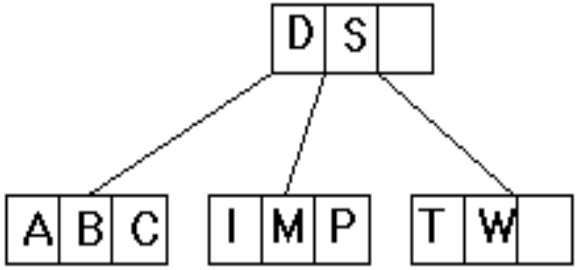
B-tree - Inserção

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

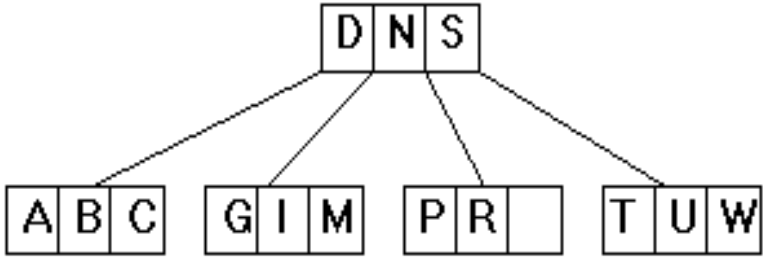


Inserção de M força outro split e o promotion de D.

Inserção de P, I, B,e W nas páginas existentes.

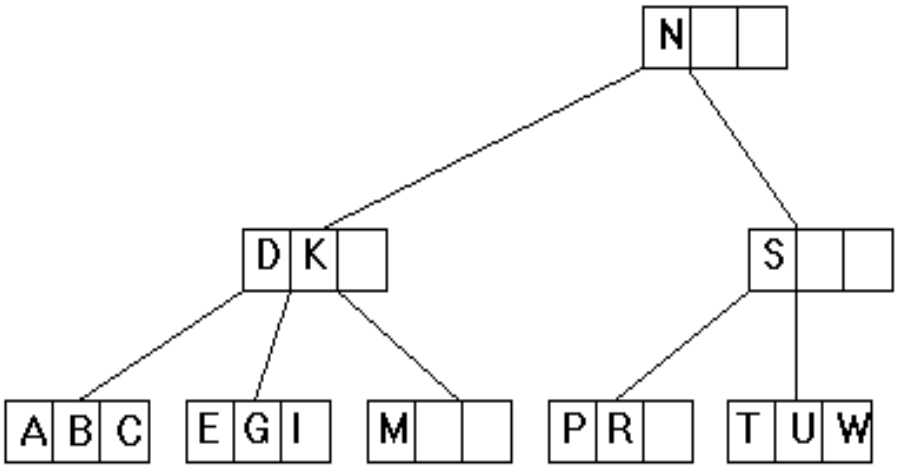


Inserção de N precisa de outro split seguido por promotion de N (G, U e R).



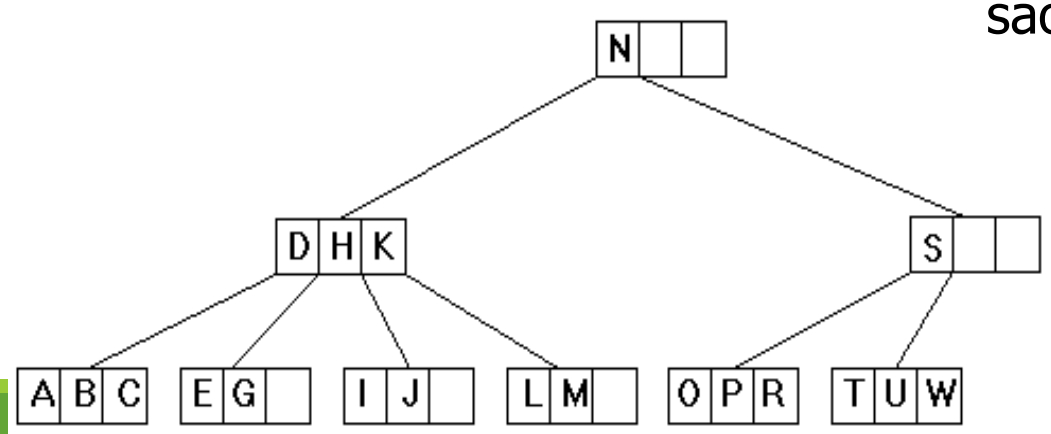
B-tree - Inserção

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V



- Inserção de k resulta no spliting no nível folha, seguido pelo promotion de k. Isto resulta no split da raiz. N é promovido para ser a nova raiz. E é posto como nó folha

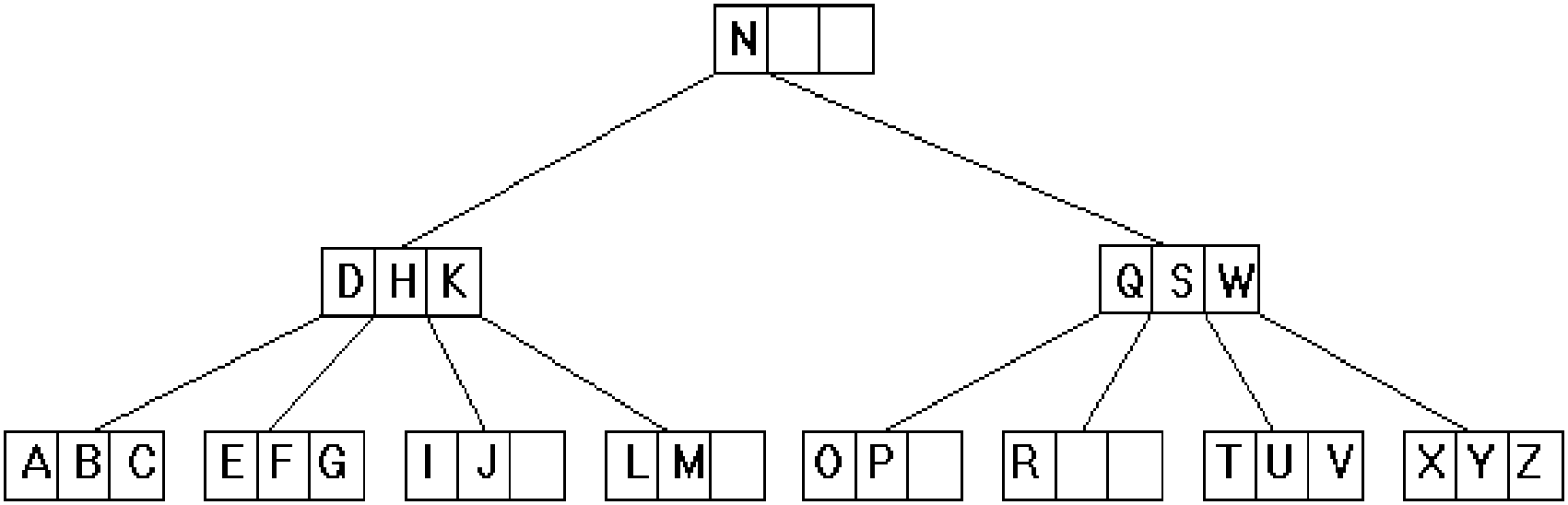
- Inserção de H resulta em split no nó folha. H é promovido. O, L e J são adicionados.



B-tree - Inserção

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Inserção de Y e Q força mais dois splits nos nós folhas. O restante das letras são adicionados.



Qual a vantagem índice secundário?

- ✓ Características do arquivo de dados:
  - Número total de registro = 30000
  - Tamanho do registro = 100 bytes
  - Bloco de disco = 1024 bytes
  - Fator de bloco = nro de registros existente em cada bloco de disco
  - FB= tam. do bloco/tam. do registro = 1024/100 =10 registros/bloco
  - Número de blocos necessários para armazenar o arquivo todo
  - Número total de registros / fator de bloco = 30000/10= 3000 blocos.
  - Busca seqüencial = 1500 acessos a blocos

Qual a vantagem do índice secundário?

- ✓ Características do arquivo índice secundário:
  - Número total de registros = 30000
  - Bloco de disco = 1024 bytes
  - Tamanho de uma chave de procura (valor+ponteiro) do nó da árvore b+ = 20 bytes
  - Nro de chaves no nó = 1024/20 = 51 chaves/no
  - Pesquisa árvore B^+ = $\log_{25}$ 300 = 3 acessos na árvore + 1 acesso no disco

HASHING

HASH

- ✓ O que é *Hashing* ?
  - A função *hash*, h(k), transforma uma chave k num endereço.
  - É similar a Indexação pois associa a chave ao endereço relativo do registro.
- ✓ Diferenças:
  - com *hashing* os endereços parecem ser aleatórios;
  - com *hashing* duas chaves podem levar ao mesmo endereço (a colisão deve ser tratada)

Tratamento de Colisões

Solução Ideal: algoritmo que não produz colisão. Muito difícil para mais de 500 chaves.

Espalhamento dos registros: Busca de um algoritmo que distribui os registros relativamente por igual.

Utilização de mais memória: é relativamente fácil achar um bom algoritmo quando se pode desperdiçar muito espaço.

Utilização de mais de um registro por endereço: técnica chamada de *bucket*.

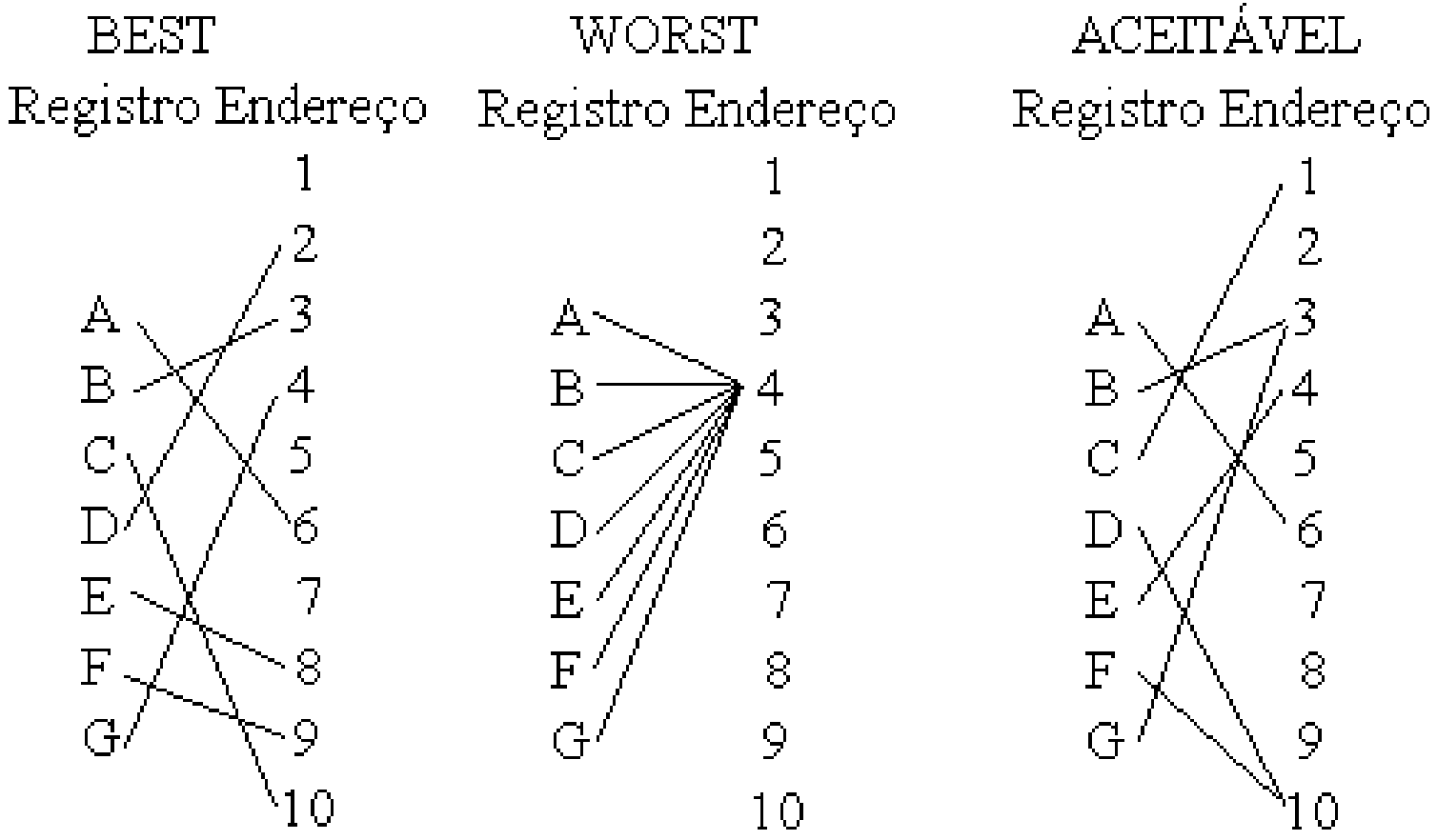
Algoritmo Simples de *Hashing*

1. Represente a chave numericamente.
2. Pique o número em pedaços e some;
3. Ajuste a soma dividindo por um número primo;
3. Divida pelo espaço de endereçamento para obter endereço final.

Distribuição dos registros entre endereços

- ✓ A função hash perfeita seria aquela que não produzisse nenhum sinônimo para um dado conjunto de chaves, em um dado endereçamento.
- ✓ A pior função seria aquela que , para qualquer chave, geraria sempre o mesmo endereço (todas as chaves seriam sinônimas).
- ✓ Uma função aceitável gera poucos sinônimos.

Distribuição de registros nos endereços



Alguns outros métodos *hashing*

examinar as chaves ?

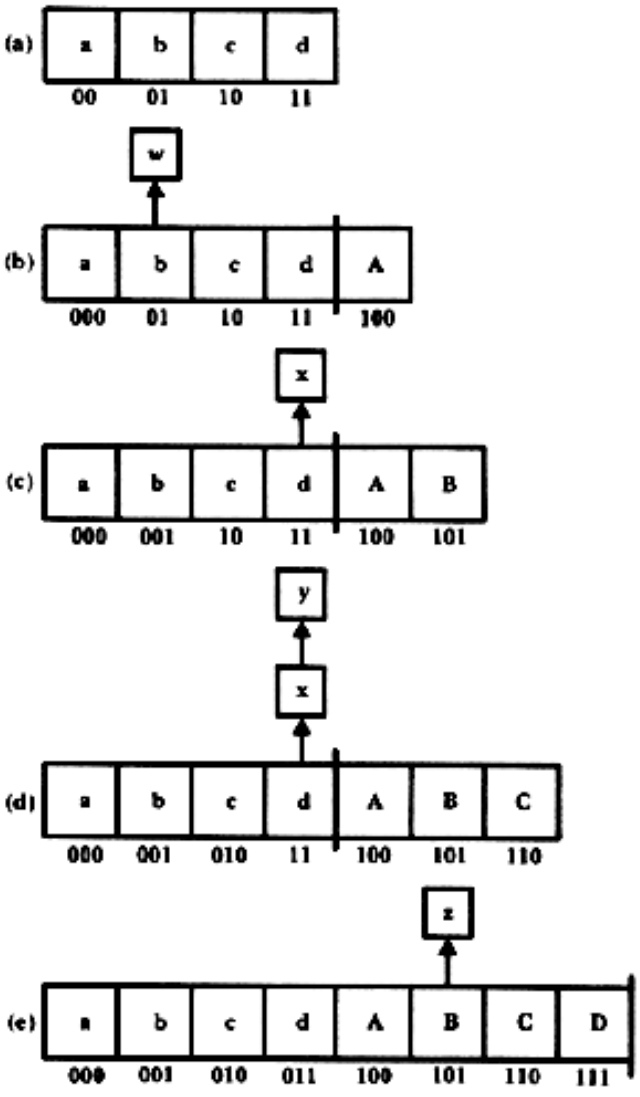
separar e somar partes da chave apenas

dividir a chave por primos

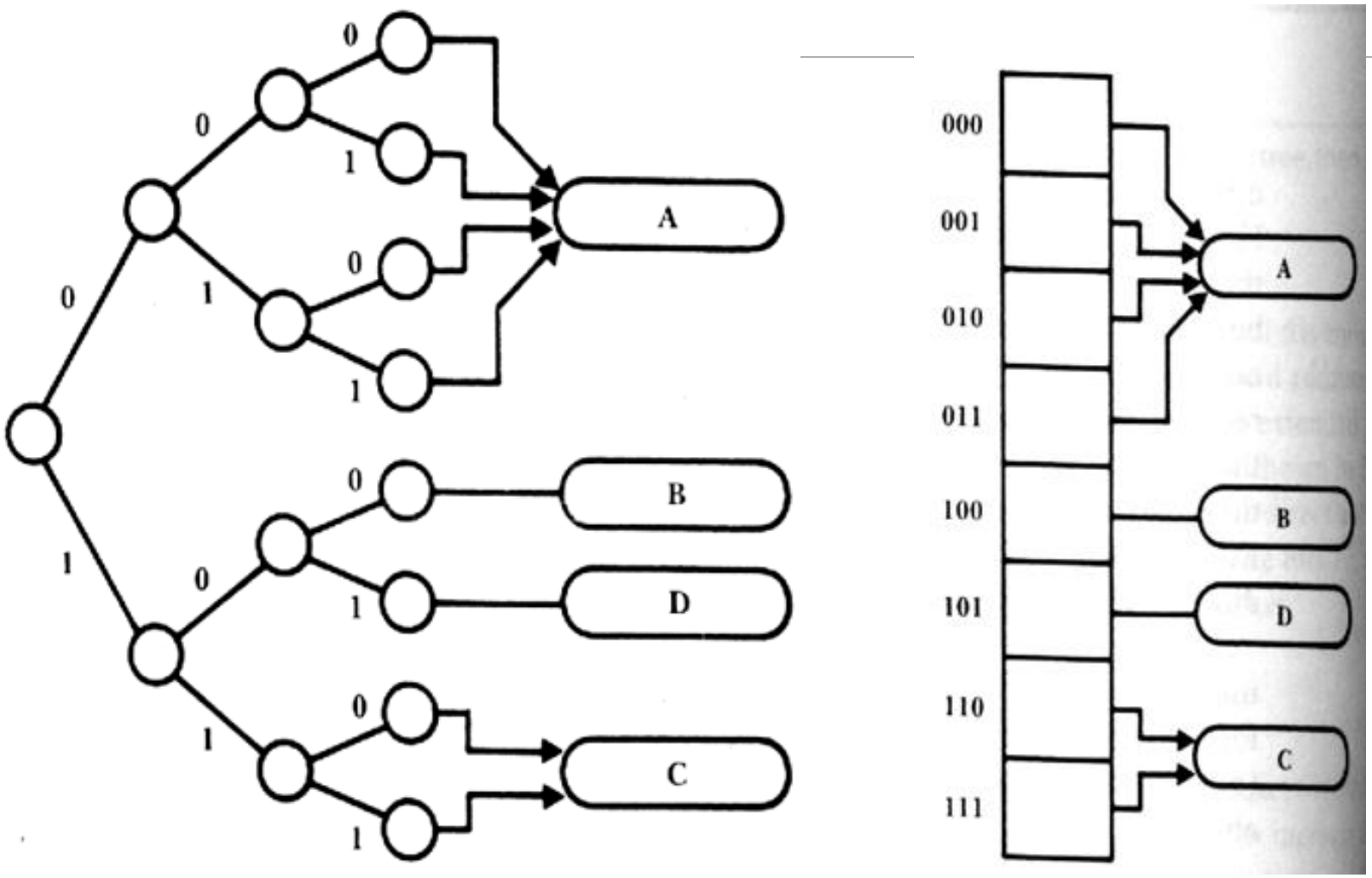
elevar a chave ao quadrado e considerar os dígitos do meio

mudar de base e dividir pelo espaço de endereçamento (considerando o módulo)

Hashing I


(a)
a b c d
00 01 10 11
w
(b)
a b c d A
000 01 10 11 100
x
(c)
a b c d A B
000 001 10 11 100 101
y
x
(d)
a b c d A B C
000 001 010 11 100 101 110
z
(e)
a b c d A B C D
000 001 010 011 100 101 110 111

Hashing Extensível



Exercício

Para criar índices em SQL use o comando:

- **CREATE INDEX nome_indice ON tabela(coluna)**

Para mostrar o plano de acesso do SQL para processar um consulta use:

- **EXPLAIN** consulta

Exemplo:

Create index ra_idx on Aluno(ra)

Explain select ra, nome from aluno force index(ra)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aluno	index	*NULL*	ra_idx	9	*NULL*	7	

Exercício

Refazer o Banco de Dados acadêmico implementando as dicas de otimização contidas nestes slides.

Tipos de Tabelas MYSQL*

- ✓ Tabelas com suporte para transações, as chamadas tabelas seguras (TST)
- ✓ Tabelas sem suporte para transações, as chamadas Tabelas Não Seguras (TNS)

* Mais detalhes no manual do MySQL(capítulo 7)

Tabelas Seguras

- ✓ Mais segura. Mesmo se o MySQL falhar ou se você tiver problemas com hardware, você pode ter os seus dados de volta, ou através de recuperação automática ou de um backup ou do log de transação.
- ✓ Você pode combinar muitas instruções e aceitar todas de uma vez com o comando **COMMIT**.
- ✓ Você pode executar um **ROLLBACK** para ignorar suas mudanças (se você não estiver rodando em modo auto-commit).
- ✓ Se uma atualização falhar, todas as suas mudanças serão restauradas. Pode fornecer melhor concorrência se a tabela obter muitas atualizações concorrentes com leituras.
- ✓ Tabelas InnoDB e BDB

Tabelas Não Seguras

- ✓ Muito mais rápida e não há nenhuma sobrecarga de transação.
- ✓ Usará menos espaço em disco já que não há nenhuma sobrecarga de transação.
- ✓ Usará menos memória para as atualizações.
- ✓ MyIsam (padrão), Merge e Heap

Como MYSQL utiliza índices

- ✓ Os índices são utilizados para encontrar registros com um valor específico de uma coluna rapidamente. Sem um índice o MySQL tem de iniciar com o primeiro registro e depois ler através de toda a tabela até que ele encontre os registros relevantes. Quanto maior a tabela, maior será o custo. Se a tabela possui um índice para as colunas em questão, o MySQL pode rapidamente obter uma posição para procurar no meio do arquivo de dados sem ter que varrer todos os registros.
- ✓ Se uma tabela possui 1000 registros, isto é pelo menos 100 vezes mais rápido do que ler todos os registros sequencialmente. Note que se você precisar acessar quase todos os 1000 registros, seria mais rápido acessá-los sequencialmente porque evitaria acessos ao disco.
- ✓ Todos os índices do MySQL (PRIMARY, UNIQUE e INDEX) são armazenados em **árvores B.** Strings são automaticamente compactadas nos espaços finais e prefixados.

Como o MySQL utiliza índices

- ✓ Para encontrar rapidamente os registros que coincidam com uma cláusula WHERE.
- ✓ Para recuperar registros de outras tabelas ao realizar *joins*.
- ✓ Para encontrar o valor MAX() ou MIN() para uma coluna indexada espeifica. Isto é otimizado por um preprocessador que confere
- ✓ Para ordenar ou agrupar uma tabela se a ordenação ou agrupamento for feito em um prefixo mais à esquerda de uma chave útil
- ✓ Em alguns casos uma consulta pode ser otimizada para recuperar valores sem consultar o arquivo de dados