Resumo P1 LBD

Índices

- A indexação independe da organização dos registros no arquivo de dados
- São armazenados em arquivos auxiliares chamados arquivos de índice
- Classificação de arquivos índice:
    - Baseados em arquivos ordenados (índices de um só nível)
    - Baseados em vários níveis (estruturas de dados de árvore)
    - Outras estruturas.

- **Índice Primário**: quando o campo de indexação é uma chave e o arquivo de dados é ordenado por esse campo. (pode haver 1)
- **Índice de agrupamento ("cluster")**: quando o campo de indexação não é chave mas o arquivo de dados é ordenado por esse campo. (pode haver 1)
- **Índice secundário**: campo de indexação não é o de ordenação. (pode haver vários)

- Índices são da forma **<K, P>** onde **K é o campo** e **P é o endereço** do bloco.
- Existe um registro de índice para cada bloco do arquivo;
- K é a chave do registro âncora (índice esparso)
    - [Primeiro registro de cada bloco é registro âncora ou âncora do bloco]

"Fórmulas"

Fator de Bloco: **bfr = *floor(tamanhoBloco / tamanhoRegistro)***
Número de Blocos: **b = ceiling(numRegistros / bfr)**

tamanhoBloco => B
tamanhoRegistro => R
numRegistros => r

Custo de Busca:

Sem indice:
Busca Binária pode levar log(r) => log(numRegistros);

Com índice:
Busca Binária pode levar log(b) => log(numBlocos)

Exemplos:

Parâmetros gerais:

Tamanho do bloco: B = 1024 bytes

Alocação não espalhada

Parâmetros do arquivos de dados:

Número de registros: r = 30.000 registros

Tamanho de registro: R = 100 bytes

Calculando:

Fator de bloco: bfr = **floor(1024/10) = 100** registros por bloco

Número de blocos: b = **ceiling(30000/100) = 3000** blocos

Parâmetros gerais:

Tamanho do bloco: B = 1024 bytes

Alocação não espalhada

Parâmetros do arquivos de dados:

Chave primária = 9 bytes

Ponteiro = 6 bytes

Logo, tamanho do registro R = 15 bytes

Número de registros: r = 3000 registros

Calculando:

Fator de bloco: bfr = **floor(1024/15) = 68** registros por bloco

Número de blocos: b = **ceiling(3000/68) = 45** blocos

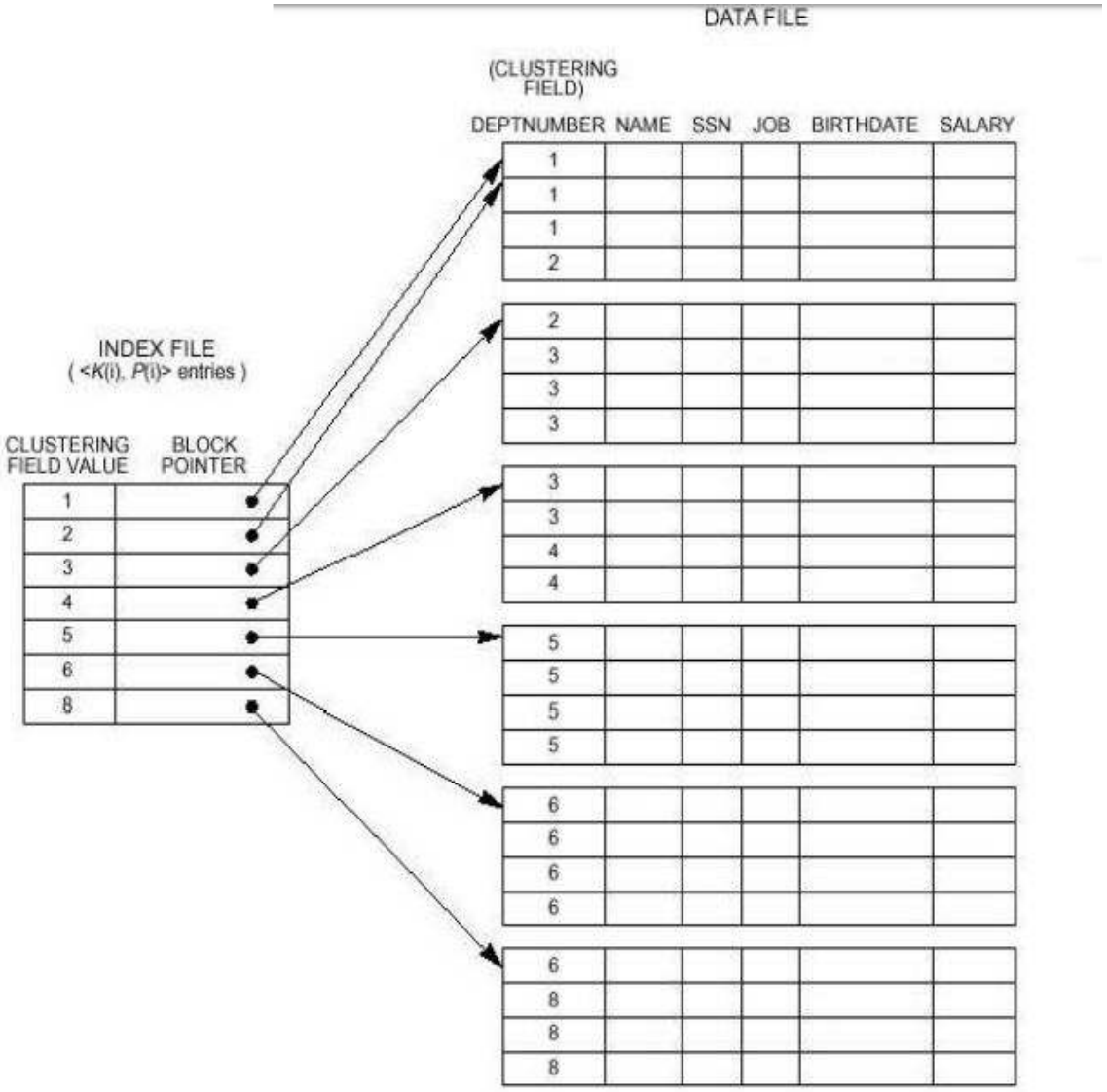
Analisando:

Sem índice, uma busca binária levaria **log(3000) = 12** acessos a blocos

Com índice, uma busca binária levaria **log(45) = 6** acessos a blocos de índice + 1 acesso ao arquivo de dados

Índice Cluster [esparso]

- Ordena o arquivo porém pode se repetir.
- Existe um registro de índice para cada valor diferente do campo de cluster no arquivo de dados *(Índice esparso)*
  Ex:



Índice secundário [denso]

Campo de indexação:

- Não ordena o arquivo de dados
- Pode ou não ser uma chave no arquivo de dados

No caso de ser chave:

- Existe uma entrada de índice para cada registro no arquivo de dados *(índice denso)*
- Provê maior ganho <u>com relação a uma busca sem índice</u> que o índice primário => usa busca seqüencial no arquivo de dados
- Precisa de maior espaço de armazenamento e o tempo de busca é maior que o índice primário

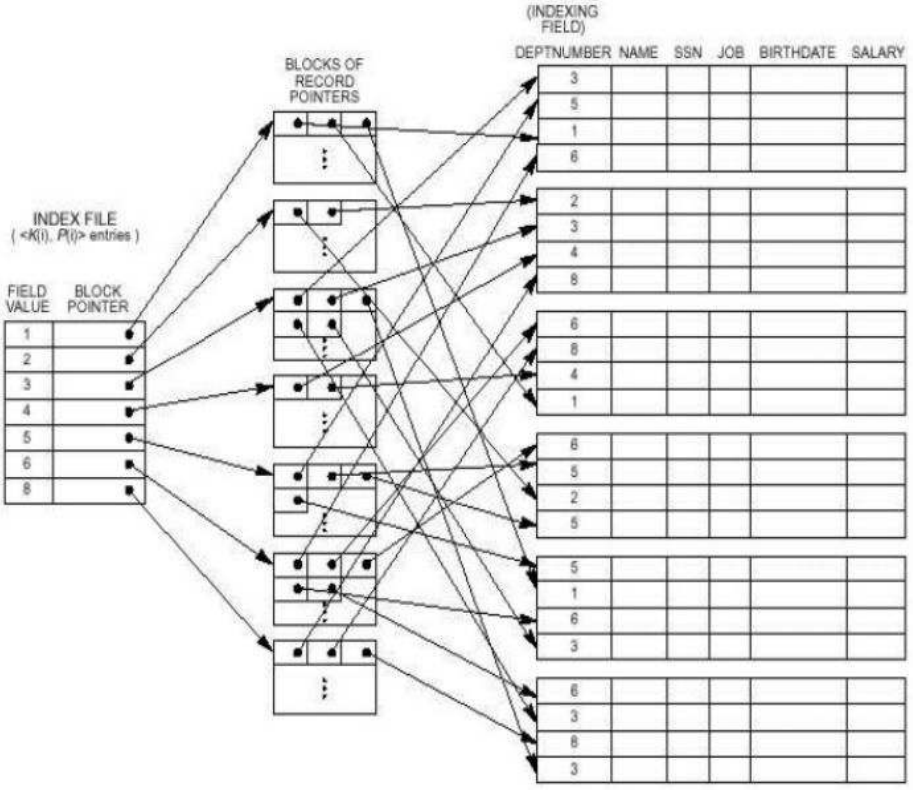
No caso de ser um campo não chave:

<u>Opção 1:</u> uma entrada de índice para cada registro do arquivo de dados

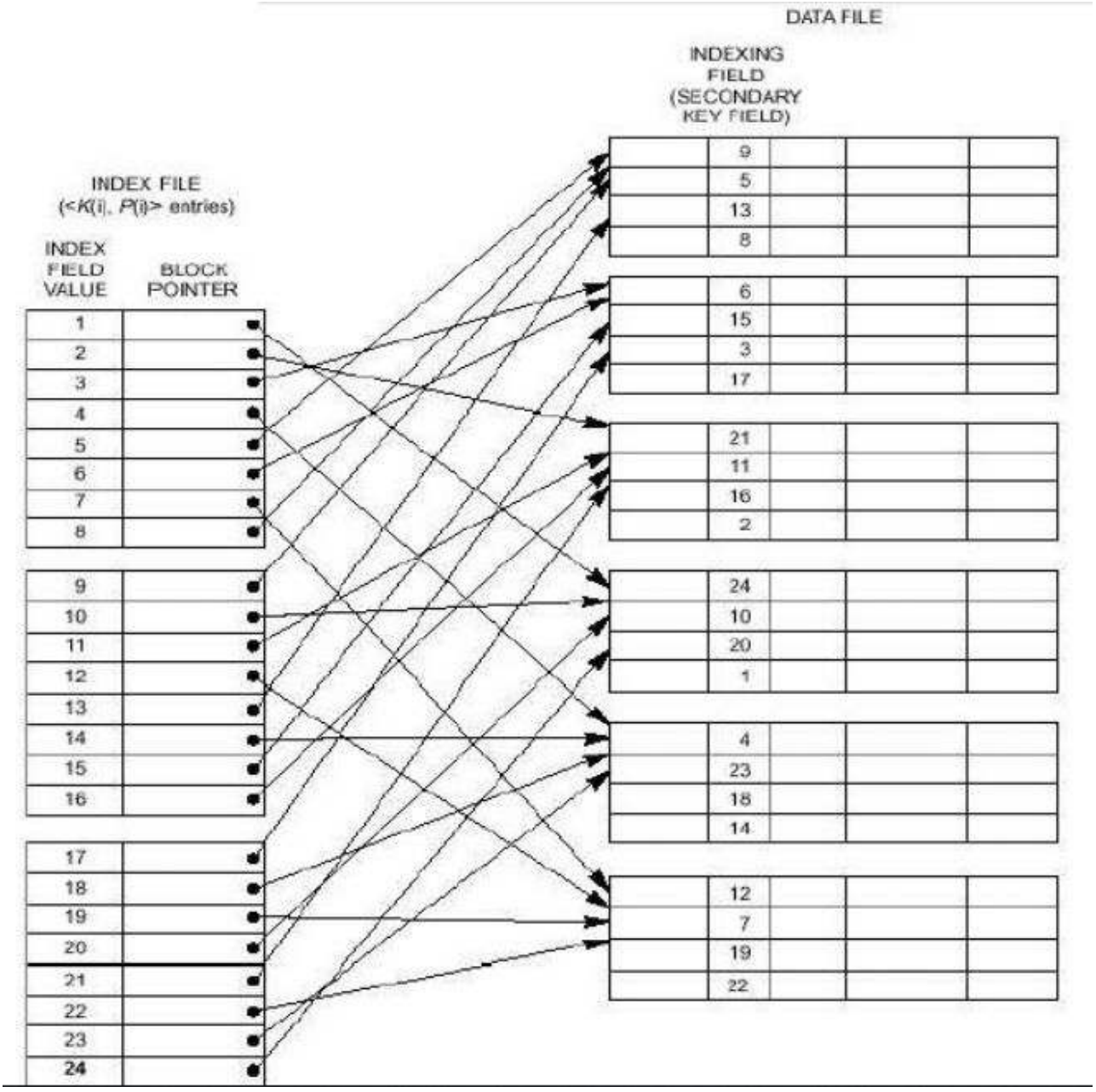
<u>Opção 2:</u> Vários ponteiros em cada entrada do índice; uma para cada posição do arquivo de dados onde o valor do campo de indexação ocorre

<u>Opção 3:</u> Cada entrada do índice aponta para um bloco de registros que contém ponteiros a blocos do arquivo de dados (Veja Fig.)

- Provê uma ordenação lógica dos registros pelo campo de indexação.



Ex:


DATA FILE
INDEXING FIELD (SECONDARY KEY FIELD)
INDEX FILE (<K(i), P(i)> entries)
INDEX FIELD VALUE
BLOCK POINTER
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
9
5
13
8
6
15
3
17
21
11
16
2
24
10
20
1
4
23
18
14
12
7
19
22

Exemplo índice secundário:

Parâmetros gerais:

Tamanho do bloco: B = 1024 bytes

Alocação não espalhada

Parâmetros do arquivos de dados:

Chave primária = 9 bytes

Ponteiro = 6 bytes

Logo, tamanho do registro Ri = 15 bytes

Porém, apenas o Registro é R = 100

Número de registros: r = 30000 registros

Calculando:

bfri = floor(B/R) = **floor(1024/15) = 68** regs por bloco

bfr (sem indice) = floor(1024/100) = 10 regs por bloco

bi = ceiling(r/bfr) = **ceiling(30000/68) = 442** blocos

b (sem indice) = 30000/10 = 3000 blocos

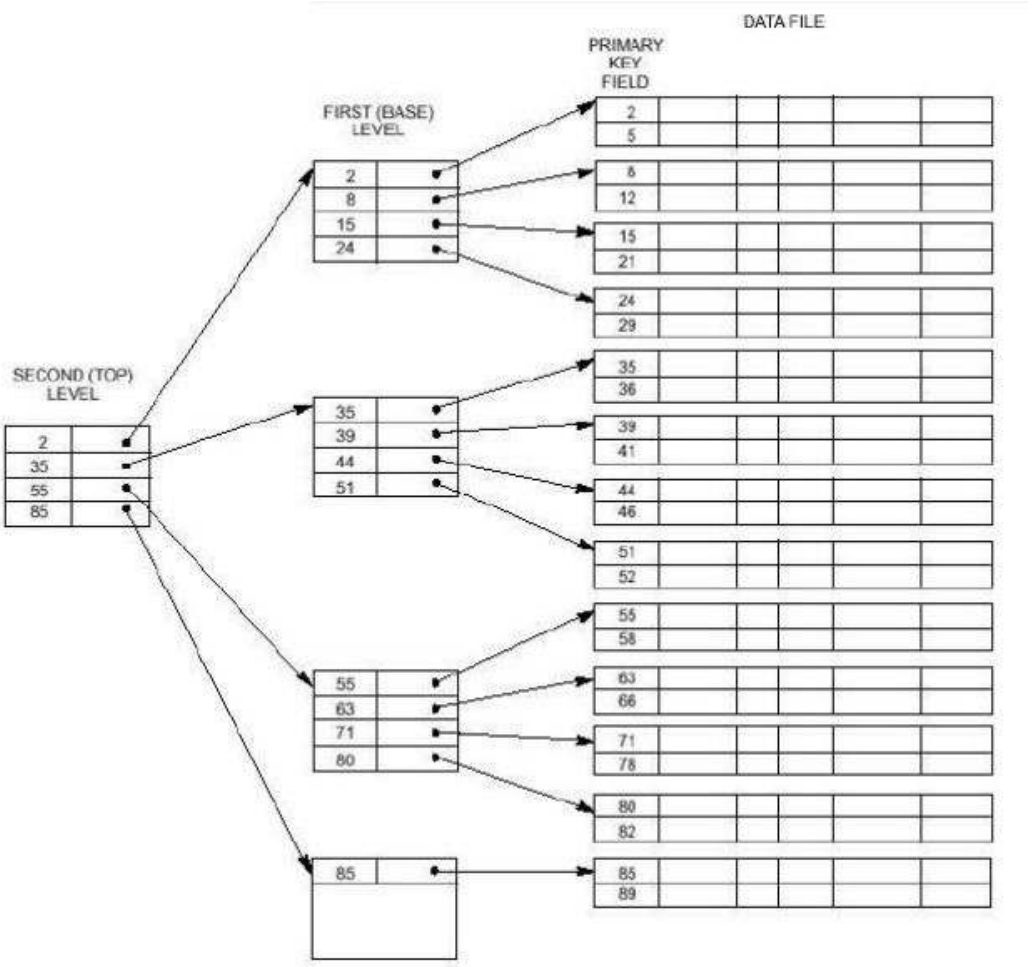
Analisando:

Sem índice, temos que o **bfr = 10 regs/bloco** e **b = 3000 blocos**, e uma busca levaria cerca de **ceil(3000/2) = 1500 acessos a blocos**

Porém, com índice, temos um custo de **log(2, 442) = 9 acessos a blocos de índice + 1 acesso ao arquivo de dados**

Índices Multinível

- Busca reduzir o número de acesso a blocos do índice de **log(2, bi) para log(fo, b)**.
- **Fan-out (fo):** Fator de bloco do arquivo de índice; fo > 2
- Pode ser construído sobre qualquer tipo: primário, secundário ou de cluster
- Se o índice for esparso é necessário acessar o bloco do arquivo de dados para determinar a existência de um registro.



Ex:
Considerando o ex anterior:

Parâmetros gerais:

Tamanho do bloco: B = 1024 bytes

Parâmetros do arquivos de dados:

Ri = 15 bytes

Calculando:

bi = ceiling(r/bfr) = **ceiling(30000/68) = 442** blocos

Fan-Out = bfr = **floor(1024/15) = 68** entradas de índices por bloco

Estrutura dos índices:

- Nível 1: b1 = 442 blocos
- Nível 2: b2 = **ceil(b1/fo) = ceil(442/68) = 7 blocos**
- Nível 3: b3 = **ceil(b2/fo) = ceil(7/68) = 1 bloco**
- Número de níveis t = ceil(log(fo, r)) = **ceil(log(68, 30000)) = 3**
- Número de acessos: **t + 1 = 4**

Inserção/remoção em índices multiníveis:

Remoção:

Se o registro for o único registro no arquivo com um valor específico do campo de indexação, o valor deve ser removido do índice.

Remoção de índices de um só nível:

- **Densos:** remoção do registro índice é similar a do registro do arquivo
- **Esparsos:** Procura-se valor seguinte do campo índice no arquivo e substitua entrada no índice. Caso contrário ela é removida.

Inserção:

Inserção de índices de um só nível:

- Busca o lugar de inserção
- **Densos:** se não aparece no índice, insira-o.
- **Esparsos:** índice tem uma entrada para cada bloco, precisa ser mudado só no caso da criação de um novo bloco.

Processamento de Consultas

Merge-sort

Ordenação externa por merge-sort:

- M = tamanho da memória (em blocos ou páginas)

Pseudocódigo encurtado:

(para cada relação (tabelinha))

Lê M blocos da relação em memória

Ordena eles

Escreve eles no próximo run

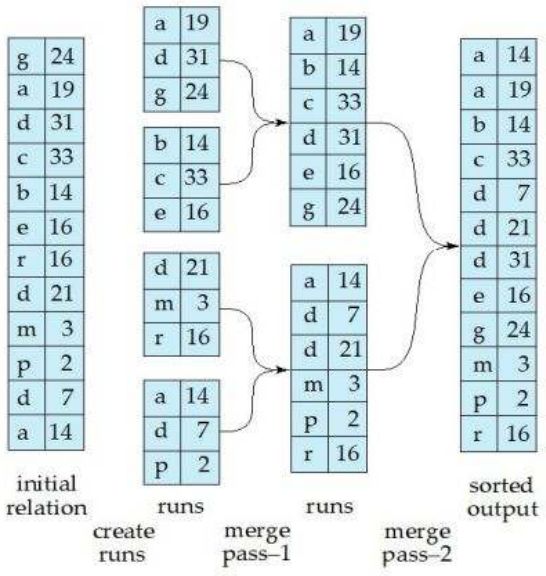
- Um passo reduz o número de runs por um fator de M - 1, e cria runs maiores pelo mesmo fator.
  - Ex: Se **M=11**, e existirem **90 runs**, **um passo** reduz o número de runs **para 9**, num fator de 10 o tamanho inicial de runs
  -
- Número total de passos de intercalação requeridos: $\lceil \log_{M-1}(b_r/M) \rceil$.

  Onde M é o tamanho da memória (tamanho do bloco na primeira run) e b = número de blocos
- Acessos a disco para a criação do run inicial assim como em cada passo é 2b (número de blocos de registros)
- Sendo assim, o número total de acessos para a ordenação externa:

$$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

Ex:

Fo = 1 e
M = 3



Junção de laços aninhados

Para calcular a junção theta $r \theta s$
for each tupla t_r **in** r **do begin**
for each tupla t_s **in** s **do begin**
verifique par (t_r, t_s) para ver se eles satisfazem a condição de junção θ
Se eles cumprem faça, adicione $t_r \bullet t_s$ ao resultado.
end
end

- **r** é chamada de **relação externa** e **s é a interna**
- Não precisa de índices e pode ser usada com qualquer tipo de condição junção
- É custoso porque examina todo par de tuplas nas duas relações.
- No **pior caso**, se a memória somente pode conter um bloco de cada relação, o custo estimado é **nr * bs + br** onde:
  nr = número de registros de r [externa]
  bs = número de blocos de s [interna]
  br = número de blocos de r [externa]
- Se a menor relação cabe totalmente na memória, use ela como a relação **interna**. Reduz o custo para br + bs acessos a disco.

Ex:
Número de registros de **cliente: 10,000** | **dono-conta: 5000**
Número de blocos de **cliente: 400** | **dono-conta: 100**

Assumindo o pior caso de disponibilidade de memória o estimativo de custo é:

- **5000 * 400 + 100 = 2.000.100** acessos a disco com **dono-conta** como **relação externa**

- **10000 * 100 + 400 = 1.000.400** acessos a disco com **cliente** como a **relação externa**
- Se a menor relação (dono-conta) se ajusta totalmente na memória, o estimativo de custo será de 500 acessos a disco

Junção de laços aninhados <u>por bloco</u>

- Variação da junção de laços aninhados na qual todo bloco da relação interna é emparelhada com todo bloco da relação externa

```
for each bloco B_r de r do begin
    for each bloco B_s de s do begin
        for each tuple t_r em B_r do begin
            for each tuple t_s em B_s do begin
                Verifique se (t_r,t_s) satisfaz a condição
                    de junção
                se eles cumprem, adicione t_r • t_s ao
                        resultado
            end
        end
    end
```

- Estimativo do pior caso: **br * bs + br** acessos a blocos, onde:
  br: número de blocos da relação externa
  bs: número de blocos da relação interna
- Melhor caso: **br + bs** acessos a blocos
- Melhoras para os algoritmos de laços aninhados e laços aninhados por blocos:
  - No laço aninhado por blocos, use **M - 2 blocos** de disco como unidade de bloqueio da **relação externa**, onde **M = tamanho da memória em blocos**; use os dois blocos restantes como memória intermediária para a relação interna e a saída

    Custo = $\lceil b_r / (M-2) \rceil * b_s + b_r$
  - Se os atributos da equi-join formam uma chave da relação interna, o laço interno finaliza apenas seja achado o primeiro casamento
  - Use índice sobre a relação interna se existir <u>(próximo tópico)</u>

Junção de laços aninhados indexada

- Busca em índices pode substituir a exploração no arquivo se:
    - A junção é uma equi-join ou junção natural
      E
    - Um índice é disponível sobre o atributo de junção da relação interna

- Para cada tupla tr na relação **externa r**, use o índice para procurar tuplas em **s** que satisfaçam a condição de **junção com a tupla tr**.
- Pior caso: buffer tem espaço para unicamente **uma página de r**, e, para cada **tupla de r**, nós fazemos uma busca no **índice de s**.

- Custo da junção: **br + nr + c**
    - br = blocos em r [externa]
    - nr = registros em r [externa]
    - c = custo de percorrer o índice trazendo todas as tuplas de **s** que casam com uma tupla de **r**
        - c pode ser estimado como o custo de uma única seleção sobre **s** usando a condição de junção.

- Se existirem índices sobre os atributos de junção de **r** e **s**, use a relação com menos tuplas como relação externa.

Ex:

Relembrando dados iniciais:

- Número de registros de **cliente: 10,000** | **dono-conta: 5000**
- Número de blocos de **cliente: 400** | **dono-conta: 100**

Compute *dono-conta JOIN cliente*, com **dono-conta como a relação externa.**

Seja cliente com um índice primário B+ sobre o attr de junção *nome-cliente* que tem **20 entradas em cada nó**.

A relação ***cliente* tem 10.000 tuplas**, a **altura da árvore é 4** e um ou mais acessos são necessários pra achar o dado real.

Dono-conta tem **5000 tuplas**.

Custo da junção:

Se for aninhada por blocos: **br * bs + br** = 100 * 400 + 100 = **40100 acessos a disco** assumindo o pior caso

Junção de laços aninhados indexada:

br + nr + c

100 * 5000 + 5 = 25100 acessos a disco. Custo de CPU menor!

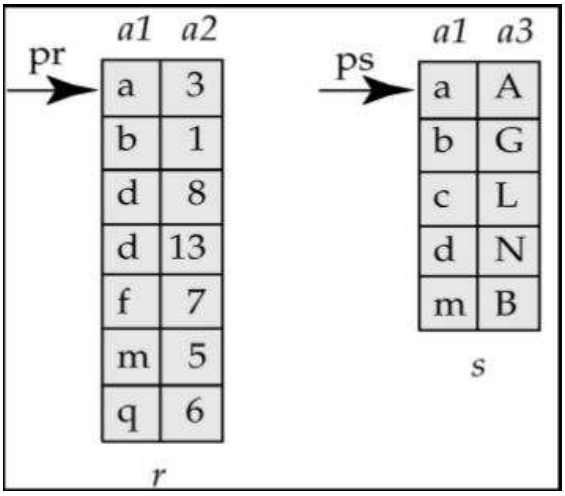
Junção por fusão

1) Ordena as duas relações pelo seu atributo de junção (se não estiverem ordenados)

2) Fusiona as relações ordenadas para fazer a junção:

a) Passo de junção similar ao passo de merge do merge-sort

b) A diferença principal é a manipulação de valores duplicados nos atributos de junção (todo par com o mesmo valor no atributo de junção devem ser casados)



- Cada bloco precisa ser lido **somente uma vez** (assumindo que todas as tuplas para algum dado valor do atributo de junção cabem na memória)
- **<u>Número de acessos a blocos:</u>**
  - **br + bs + custo de ordenar (se necessário)**

- **Junção por fusão híbrida:** Se uma relação estiver <u>ordenada</u> e a outra tem <u>índice secundário B+ sobre o atributo de junção</u>:
  1) Fusione a relação ordenada com as entradas das folhas da relação índice (B+)
  2) Ordene o resultado de acordo com os endereços das tuplas da relaçao desordenada
  3) Busca a relação desordenada em ordem de endereço físico e fusiona com os resultados anteriores para mudar endereços pelas tuplas reais

  - **Busca sequencial mais eficiente que busca aleatória**

Junção Hash

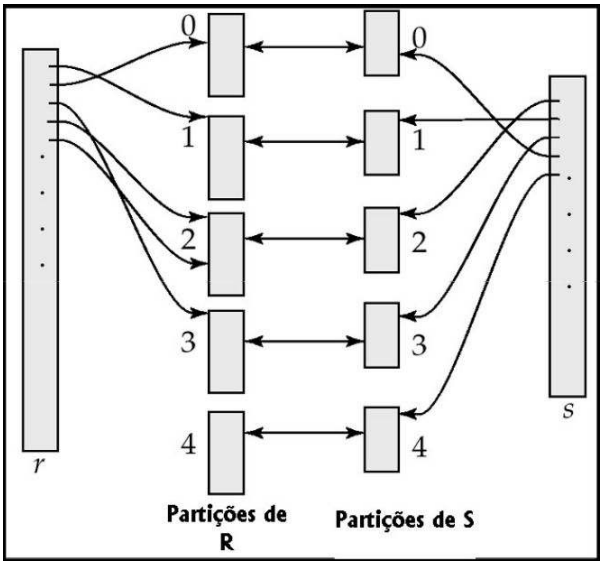
- Aplicável para equi-joins e joins naturais
- Uma função hash *h* é usada para dividir tuplas das duas relações
- *h* aplica os JoinAttrs a valores {0, 1, ... , n}, onde JoinAttrs representam os attrs comuns de **r** e **s** usados na junção natural.
- Tuplas de **r** em **ri** unicamente precisam ser comparadas com tuplas de **s** em **si**
- Não precisam ser comparadas com tuplas de s em qualquer outra partição, já que:
    - uma tupla de r e uma tupla de s que satisfazem a condição de junção terá o mesmo valor para os atributos de junção.
    - Se esse valor é associado (hashed) a algum valor i, a tupla de r tem que estar em ri e a tupla de s em si.
- Ou seja, vc compara as tuplas *hasheadas*.

Custo:

3(br + bs) + 2n

- O número de divisões da relação de prova r é o mesmo que para a relação de construção s; o número de passos para dividir r é o mesmo que para s
- É melhor escolher a MENOR relação para a de CONSTRUÇÃO
- Quando a entrada de construção pode ser armazenada TOTALMENTE na memória, o **n** de *h* vai para 0 e o algoritmo não divide as relações em arquivos temporários.
    - O custo passa a ser **br + bs** no melhor caso.

Ex:



Exemplo de Custo da Junção de Hash

Cliente* |*X*| *dono-conta

Assuma que o tamanho da memória é de 20 blocos

Relembrando dados iniciais:

- Número de blocos de **cliente: 400** | **dono-conta: 100**

- *Dono-conta* é usado como entrada de construção.
- Divide esta em 5 divisões (**100/20**), cada uma de 20 blocos.
- Da mesma forma, relação cliente é dividida em 5 divisões, e vai ter tamanho 80.

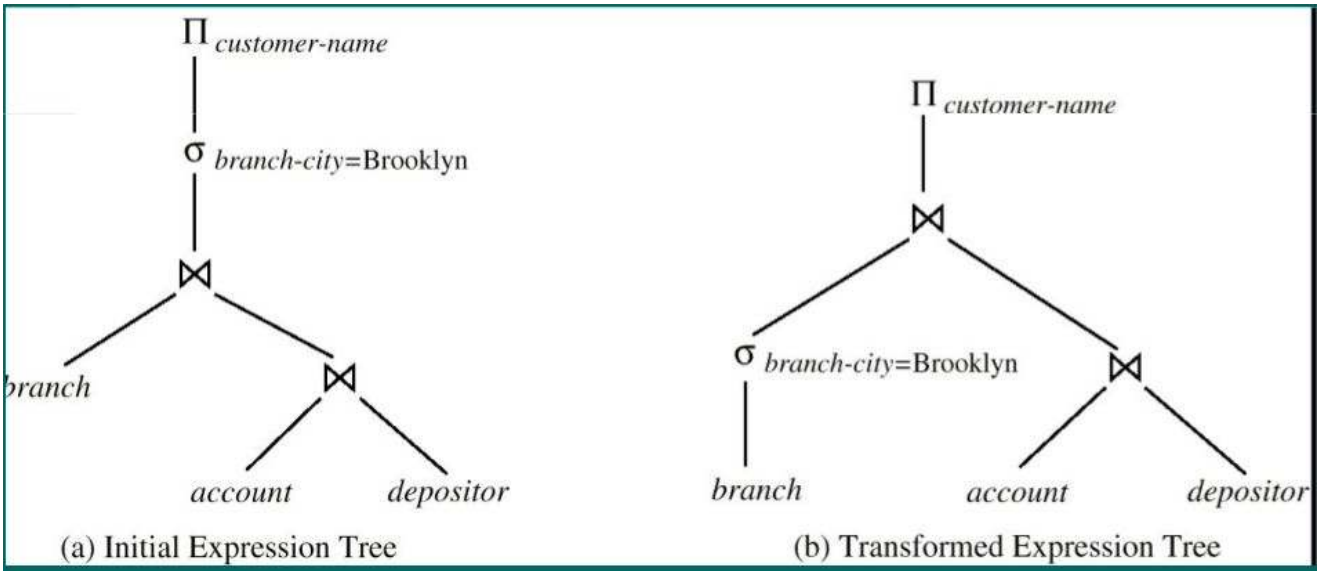
Custo total: 3(br + bs) + 2n

3(100 + 400) = 1500 transferências de blocos

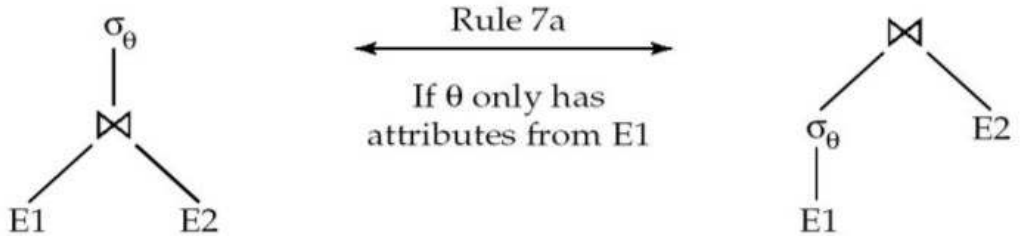
- ignora o custo de escrever os blocos parcialmente cheios

Otimização de Consultas

- Diferença de custo entre uma boa e uma má maneira de avaliar uma consulta pode ser grande
    - Exemplo: execução de r X s seguido por uma seleção r.A = s.B é bem mais lento que executar uma junção sobre a mesma condição



Dica: joga o WHERE mais perto da “folha”:



Ex1:

- Consulta: Achar os nomes de todos os clientes que tem uma conta em alguma agência localizada em Brooklyn.

  $\Pi_{\textit{nome-cliente}} (\sigma_{\textit{cidade-agência} = \text{“Brooklyn”}} (\textit{agência} \bowtie (\textit{conta} \bowtie \textit{dono-conta})))$

- Transformação usando regra 7a.

  $\Pi_{\textit{nome-cliente}} ((\sigma_{\textit{cidade-agência} = \text{“Brooklyn”}} (\textit{agência})) \bowtie (\textit{conta} \bowtie \textit{dono-conta}))$

- Executando a seleção o mais cedo possível reduz o tamanho da relação a ser reunida (joined).

Ex2:

- Consulta: Achar os nomes de todos os clientes com uma conta numa agência de Brooklyn cujo saldo é maior de $1000.

  $\Pi_{\textit{nome-cliente}} (\sigma_{\textit{cidade-agência} = \text{“Brooklyn”} \wedge \textit{saldo} > 1000} (\textit{agência} \bowtie (\textit{conta} \bowtie \textit{dono-conta})))$

- Transformação usando junção associativamente (Regra 6a):

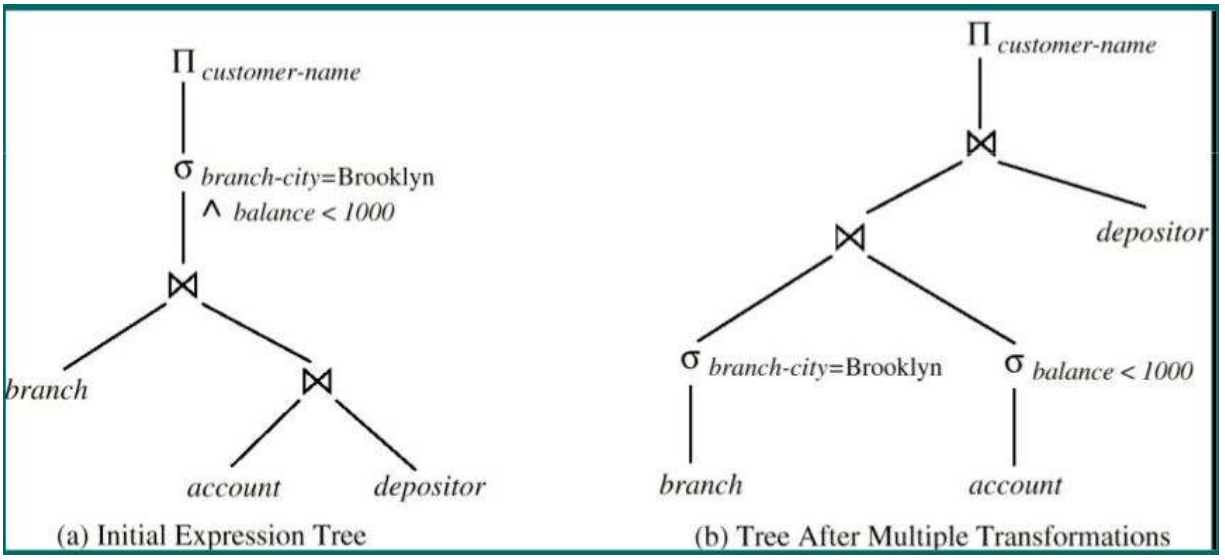
  $\Pi_{\textit{nome-cliente}} ((\sigma_{\textit{cidade-agência} = \text{“Brooklyn”} \wedge \textit{saldo} > 1000} (\textit{agência} \bowtie (\textit{conta})) \bowtie \textit{dono-conta})$

- Segunda forma fornece uma oportunidade para aplicar a regra “desenvolver as seleções cedo”, resultando na subexpressão

  $\sigma_{\textit{cidade-agência} = \text{“Brooklyn”}} (\textit{agência}) \bowtie \sigma_{\textit{saldo} > 1000} (\textit{conta})$

- Assim uma seqüência de transformações pode ser útil

Ex3:



Estimação do custo:

nr: número de tuplas de r
br: número de blocos com tuplas de r
sr: tamanho de uma tupla de r
fr: fator de bloqueio de r => número de tuplas de r que cabem num bloco
V(A, r): número de valores diferentes que aparecem em r para o attr A

- **Se as tuplas de r são armazenadas fisicamente juntas num arquivo, então:**

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$