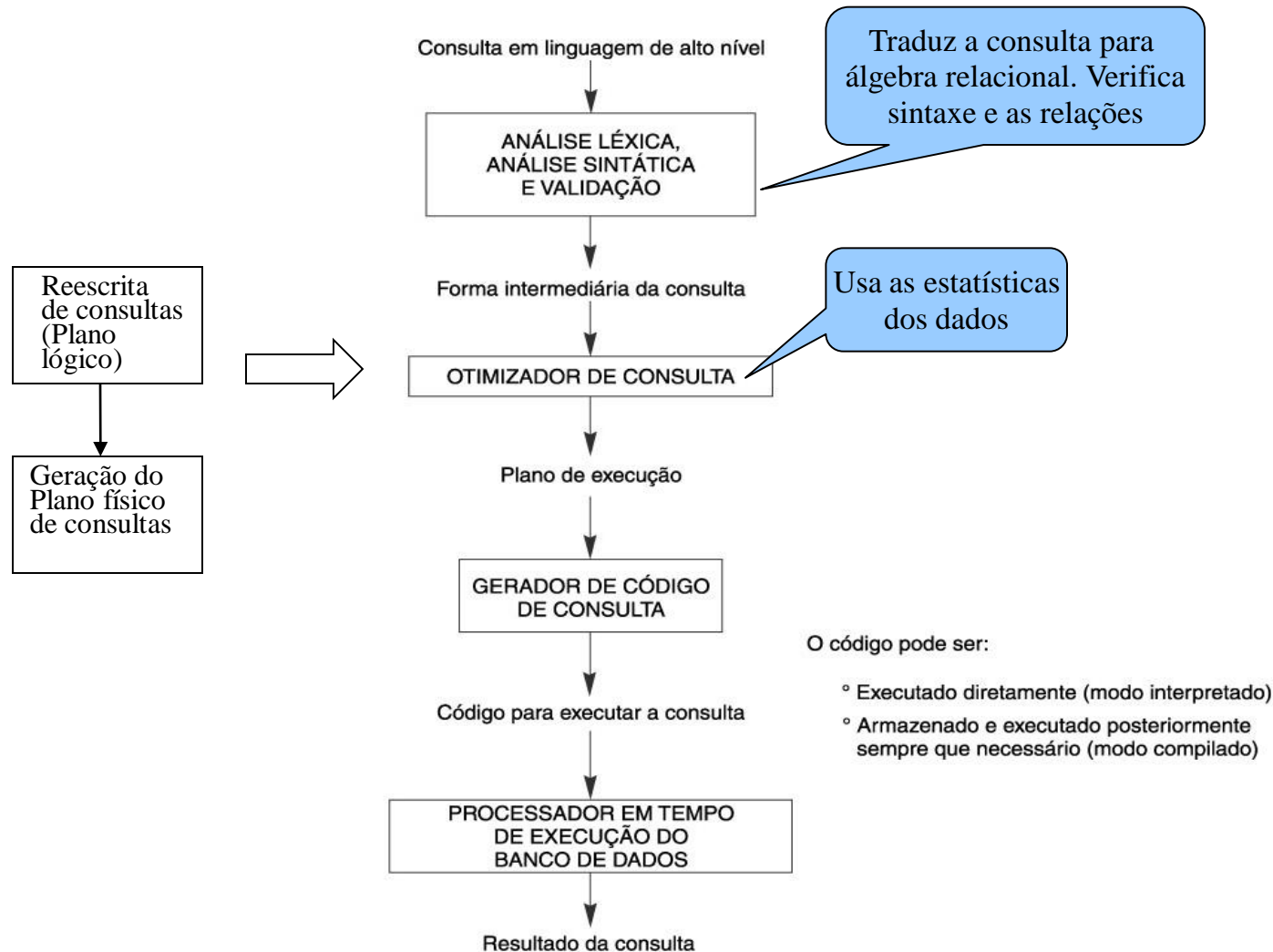


Algoritmos para Processamento e Otimização de Consultas

Laboratório de Bancos de Dados

Passos típicos durante a execução de uma consulta de alto nível



Passos Básicos no Processamento de Consultas : Otimização

- Uma expressão em álgebra relacional pode ter várias expressões equivalentes
 - Ex., $\sigma_{saldo < 2500}(\Pi_{saldo}(conta))$ é equivalente a $\Pi_{saldo}(\sigma_{saldo < 2500}(conta))$
- Cada operação em álgebra relacional pode ser avaliada usando um de vários algoritmos diferentes
 - Portanto, uma expressão em álgebra relacional pode ser avaliada de várias formas.
- Expressão anotada (especificam como executar) que especifica uma estratégia de avaliação detalhada é chamada um **plano de avaliação ou execução (plano físico)**.
 - Ex., pode usar um índice sobre *saldo* para achar contas com saldo < 2500,
 - ou podem desenvolver um rastreamento completo na relação e descartar contas com saldo ≥ 2500

Passos Básicos no Processamento de Consultas : Otimização

- Diferentes planos de execução → diferentes custos
- Responsabilidade do sistema construir um plano de execução que minimize o custo da avaliação da consulta → otimização
- Para otimizar a consulta, um otimizador de consulta precisa saber o custo de cada operação. Difícil de calcular, mas possível de estimar.
- Vamos estudar inicialmente os custos das operações (da álgebra), os processos de canalização (“pipeline”) e execução de expressões, para voltar mais tarde no processo de otimização.

Medidas do Custo de uma Consulta

- Custo é geralmente medido como o tempo total gasto para responder uma consulta
 - vários fatores contribuem para o esse custo
 - *Acessos a disco, CPU, ou mesmo comunicação na rede*
- Tipicamente o acesso a disco é o custo predominante, e é também relativamente fácil de estimar. Medido tomando em conta
 - Número de buscas (seek) * média do custo de busca
 - Número of blocos lidos * média do custo de leitura de um bloco
 - Número de blocos escritos * média do custo de escrita de um bloco
 - ⇒ Custo para escrever um bloco é maior que o custo para lêr um bloco
 - Dados são lidos novamente depois de ser escritos para estar certos que foram gravados com sucesso

Medidas do Custo de uma Consulta (Cont.)

- Por simplicidade nos usamos só o *número de transferências de blocos do disco* como a medida do custo
 - Nos ignoramos a diferença em custo entre E/S seqüencial e aleatória por simplicidade
 - Nos também ignoramos os custos de CPU por simplicidade
- Os custos dependem do tamanho do buffer em memória principal
 - Tendo mais memória reduz a necessidade de acesso a disco
 - Quantidade de memória real disponível para buffer depende dos outros processos de OS concorrentes, e é difícil determinar previamente à execução real
 - Nos usamos freqüentemente o estimativo do pior caso, assumindo unicamente a quantidade mínima de memória necessária para que a operação seja possível, é disponível.
- Sistemas reais tomam em conta o custo de CPU, diferenciam entre E/S seqüencial e aleatório, e tem em conta o tamanho do buffer
- Não incluímos o custo de escrever a saída no disco na fórmula de custo

Operação de Seleção

- **Explorador (scan) de arquivo** –algoritmos de busca que localizam e recuperam registros que cumprem uma condição de seleção.
- **Algoritmo A1** (*busca linear*). Explora cada bloco do arquivo e testa todos os registros para verificar se eles satisfazem a condição de seleção.
 - Estimativo de custo (número de blocos de disco explorados) = b_r
 - $\Rightarrow b_r$ define o número de blocos contendo registros da relação r
 - Se a seleção é um atributo chave, custo = $(b_r/2)$
 - \Rightarrow Para ao achar um registro
 - Busca linear pode ser aplicado sem importar a
 - \Rightarrow condição de seleção ou
 - \Rightarrow Ordenação de registros no arquivo, ou
 - \Rightarrow disponibilidade de índices

Operação de Seleção (cont.)

- **A2** (*busca binária*). Aplicável se seleção é uma comparação de igualdade sobre o atributo no qual o arquivo está ordenado.
 - Assuma que os blocos de uma relação são armazenados de forma consecutiva
 - Estimação do custo (número of blocos de disco a serem explorados):
 - $\Rightarrow \lceil \log_2(b_r) \rceil$ — custo de localização da primeira tupla por uma busca binária sobre os blocos
 - \Rightarrow *Mais* número de blocos contendo registros que satisfazem a condição de seleção
 - Nos veremos como estimar este custo depois.

Seleção Usando Índices

- **Exploração de Índice** – algoritmos de busca usam um índice
 - condição de seleção deve ser sobre uma chave de busca do índice.
- **A3** (*índice primário, igualdade baseada na chave*). Recupera o único registro que satisfaz a condição de igualdade
 - $Custo = HT_i + 1$
- **A4** (*índice de agrupamento, igualdade baseada num atributo não chave*) Recupera múltiplos registros.
 - Registros armazenados em blocos consecutivos
 - $Custo = HT_i + \text{número de blocos contendo registros recuperados}$
- **A5** (*índice secundário, igualdade*).
 - Recupera um único registro se a chave de busca é uma chave candidata
 - $Custo = HT_i + 1$
 - Recupera múltiplos registros se a chave de busca não é uma chave candidata
 - $Custo = HT_i + \text{numero de registros recuperados}$
 - Pode ser muito custosa!
 - Cada registro pode estar num bloco diferente
 - um acesso a bloco para cada registro recuperado

Seleções com Comparações

- Podem ser implementadas seleções da forma $\sigma_{A \leq V}(r)$ ou $\sigma_{A \geq V}(r)$ usando
 - uma busca linear ou busca binária,
 - ou usando índices das seguintes maneiras:
 - **A6** (*índice primário ou de agrupamento, comparação*). (Relação é ordenada pelo A)
 - Para $\sigma_{A \geq V}(r)$ use o índice para achar a primeira tupla $\geq v$ e explore a relação seqüencialmente desde aí
 - Para $\sigma_{A \leq V}(r)$ só explore a relação seqüencialmente até a primeira tupla $> v$; Não usa índice
 - **A7** (*índice secundário, comparação*).
 - Para $\sigma_{A \geq V}(r)$ use o índice para achar o primeira entrada no índice $\geq v$ e explore o índice seqüencialmente desde lá, para achar “pointers” para registros.
 - Para $\sigma_{A \leq V}(r)$ só explore páginas folha do índice para achar “pointers” para registros, até a primeira entrada $> v$
 - Em qualquer caso, recupere os registros apontados
 - » Pode requerer uma E/S para cada registro
 - » A busca linear pode ser mais barata se vários registros devem ser recuperados!

Implementação de Seleções Complexas

Seletividade é útil neste caso.
A razão entre o número de registros que satisfazem a condição e o número total de registros

- **Conjunção:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A8** (*seleção conjuntiva usando um índice*).
 - Seleciona uma combinação de θ_i e algoritmos A1 até A7 que resultam no mínimo custo para $\sigma_{\theta_i}(r)$.
 - verifica as outras condições da tupla depois de trazer ela para a memória (buffer).
- **A9** (*seleção conjuntiva usando um índice composto*).
 - Use o índice composto (multi-chave) apropriado se estiver disponível.
- **A10** (*seleção conjuntiva através da interseção de identificadores*).
 - Requer índices com apontadores a registros.
 - Usa o índice correspondente para cada condição, e faz a interseção de todos os conjuntos de registros de apontadores obtidos.
 - Então recupera os registros do arquivo
 - Se alguma condição não tem os índices apropriados, aplica teste em memória.

Algoritmos para Seleções Complexas

- **Disjunção:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.
- **A11** (*seleção disjuntiva através da união de identificadores*).
 - Aplicável se *todas as* condições tem índices disponíveis.
 - Senão usar busca linear.
 - Use o índice correspondente para cada condição, e faça a união de todos os registros de apontadores obtidos.
 - Então traga os registros do arquivo
- **Negação:** $\sigma_{\neg\theta}(r)$
 - Use busca linear sobre o arquivo
 - Se muito poucos registros satisfazem $\neg\theta$, e um índice é aplicável a θ
 - Achar os registros que satisfazem usando o índice e traga eles do arquivo

Ordenação

- Desempenha um papel importante nos SGBDs:
 - Saída ordenada
 - Melhora o desempenho de outras operações (junções)
- Nós podemos construir um índice da relação, e então usar o índice para ler a relação em ordem. Pode conduzir a um acesso a um bloco de disco para cada tupla.
- Para relações que cabem na memória, técnicas como quicksort podem ser usada. Para relações que não cabem na memória, o algoritmo de ordenação **externa merge-sort** é uma boa escolha.

Ordenação externa merge-sort

Seja M o tamanho da memória (em páginas ou blocos).

1. **Criar runs (seqüências ordenadas).** Seja $i = 0$ inicialmente. Repetidamente fazer o seguinte até o fim da relação:
 - (a) Leia M blocos da relação em memória
 - (b) Ordene os blocos em memória
 - (c) Escreva os dados ordenados no run R_i ; incremente i .Seja o valor final de i , N

Fase de
ordenação

2. **Intercalar os runs (intercalação N-way).** Nós assumimos (por enquanto) que $N < M$.

1. Usar N blocos de memória para carregar (“buffer”) runs de entrada, e 1 bloco para carregar a saída. Leia o primeiro bloco de cada run no sua página de buffer

2. **repita**

1. Selecione o primeiro registro (ordenado) entre todas as páginas do buffer
2. Escreve o registro no buffer de saída. Se o buffer de saída está cheio grave-o no disco.
3. Remove o registro da página (buffer) de entrada.
Se a página buffer vira vazia então
leia o próximo bloco (se houver) do run no buffer.

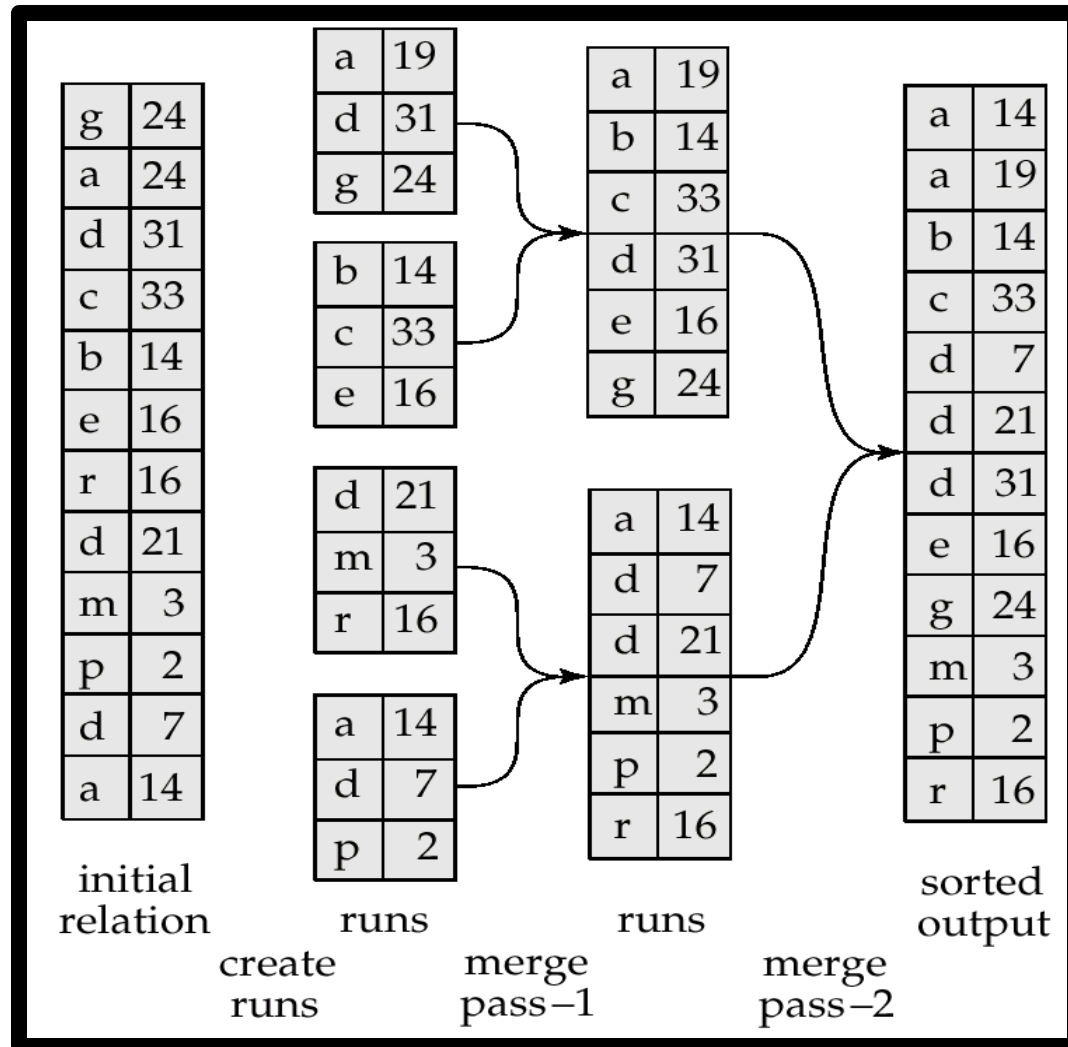
3. **até** que todas as páginas do buffer de entrada estejam vazias:

Fase de
Fusão
ou
Intercala-
ção

Ordenação externa merge-sort (cont.)

- Se $i \geq M$, vários *passos* de intercalação são necessários.
 - Em cada passo, grupos contíguos de $M - 1$ runs são intercalados.
 - Um passo reduz o número de runs por um fator de $M - 1$, e cria runs maiores pelo mesmo fator.
 - E.g. 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
 - Passos repetidos são desenvolvidos até que todos os runs tenham sido intercalados num run.

Exemplo: Ordenação Externa Usando Sort-Merge



Merge Sort Externo

- Análise de Custo:
 - Número total de passos de intercalação requeridos:
 $\lceil \log_{M-1}(b_r/M) \rceil$.
 - Acessos a disco para a criação do run inicial assim como em cada passo é $2b_r$
 - Para o passo final, nos não contamos o custo de escrita
 - nos ignoramos o custo da gravação final para todas as operações desde que a saída de uma operação possa ser enviada à operação pai sem ser escrito no disco

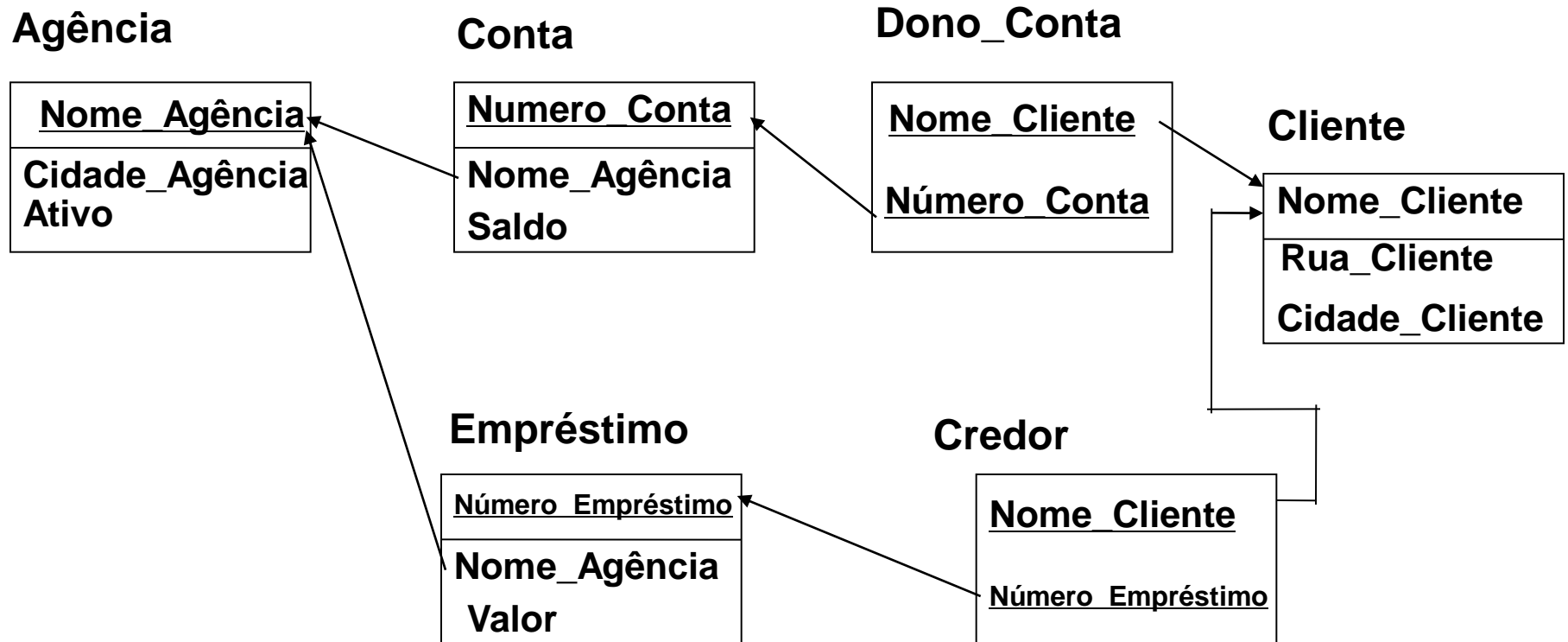
Sendo assim, o número total de acessos para a ordenação externa:

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

Operação de Junção

- Vários algoritmos diferentes para implementar junções
 - Junção de laços aninhados (Nested-loop)
 - Junção de laços aninhados por blocos
 - Junção de laços aninhados indexados
 - Junção sort-merge
 - Junção-hash
- A escolha é baseada na estimativa de custo
- Exemplos usam a seguinte informação
 - Número de registros de *cliente*: 10,000; *dono-conta*: 5000
 - Número de blocos de *cliente*: 400; *dono-conta*: 100

Novo Esquema Exemplo



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 $t_r \cdot t_s$ ao
 resultado.
 end
end
- r é chamada a **relação externa** e s a **relação interna** da junção.
- 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.

Junção de laços aninhados (cont.)

- No pior caso, se a memória somente pode conter um bloco de cada relação, o custo estimado é
$$n_r * b_s + b_r$$
acessos a disco.
- Se a menor relação cabe totalmente na memória, use ela como a relação interna. Reduz o custo para $b_r + b_s$ acessos a disco.
- 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 a relação externa, e
 - $1000 * 100 + 400 = 1,000,400$ 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.
- O algoritmo de laços aninhados por blocos (próxima transparência) é preferível.

Junção de laços aninhados por blocos

- 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  $t_r \bullet t_s$  ao  
        resultado  
      end  
    end  
  end  
end
```

Junção de laços aninhados por blocos (cont.)

- Estimativo do pior caso: $b_r * b_s + b_r$ acessos a blocos.
 - Cada bloco na relação interna s é lido uma vez para cada *bloco* na relação externa (no lugar de uma vez para cada tupla na relação externa)
- Melhor caso: $b_r + b_s$ *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 (próximo slide)

Junção de laços aninhados indexada

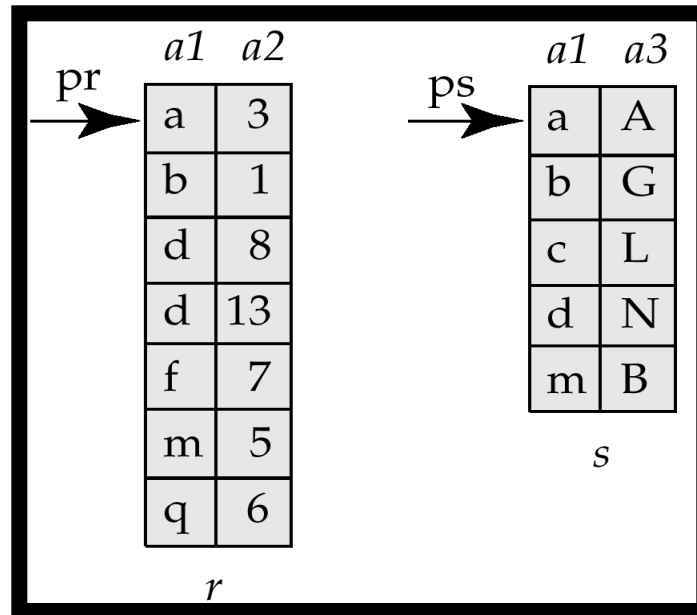
- Busca em índices pode substituir a exploração no arquivo se
 - A junção é uma equi-join ou junção natural e
 - um índice é disponível sobre o atributo de junção da relação interna
 - Pode ser construído um índice para calcular a junção.
- Para cada tupla t_r na relação externa r , use o índice para procurar tuplas em s que satisfaçam a condição de junção com a tupla t_r .
- Pior caso: buffer tem espaço para unicamente uma página de r , e, para cada tupla de r , nos fazemos uma busca no índice de s .
- Custo da junção: $b_r + n_r * c$
 - Onde c é o custo de percorrer o índice trazendo todas as tuplas de s que casam com a 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.

Exemplo custos da junção de laços aninhados

- Compute *dono-conta* ⋈ *cliente*, com *dono-conta* como a relação externa.
- Seja *cliente* com um índice primário B⁺-tree sobre o atributo de junção *nome-cliente*, que tem 20 entradas em cada nó do índice.
- A relação *cliente* tem 10,000 tuplas, a altura da árvore é 4, e um ou mais acessos são necessários para achar o dado real
- *Dono-conta* tem 5000 tuplas
- Custo da junção de laços aninhados por blocos
 - $400 \cdot 100 + 100 = 40,100$ acessos a disco assumindo o pior caso de memória (pode ser significativamente menor com mais memória)
- Custo da junção de laços aninhados indexada
 - $100 + 5000 \cdot 5 = 25,100$ acessos a disco.
 - Custo de CPU provavelmente menor que para a junção de laços aninhados por blocos.

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. O passo de junção é similar à passo de fusão merge do algoritmo de 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
 - c. Algoritmo detalhado no livro



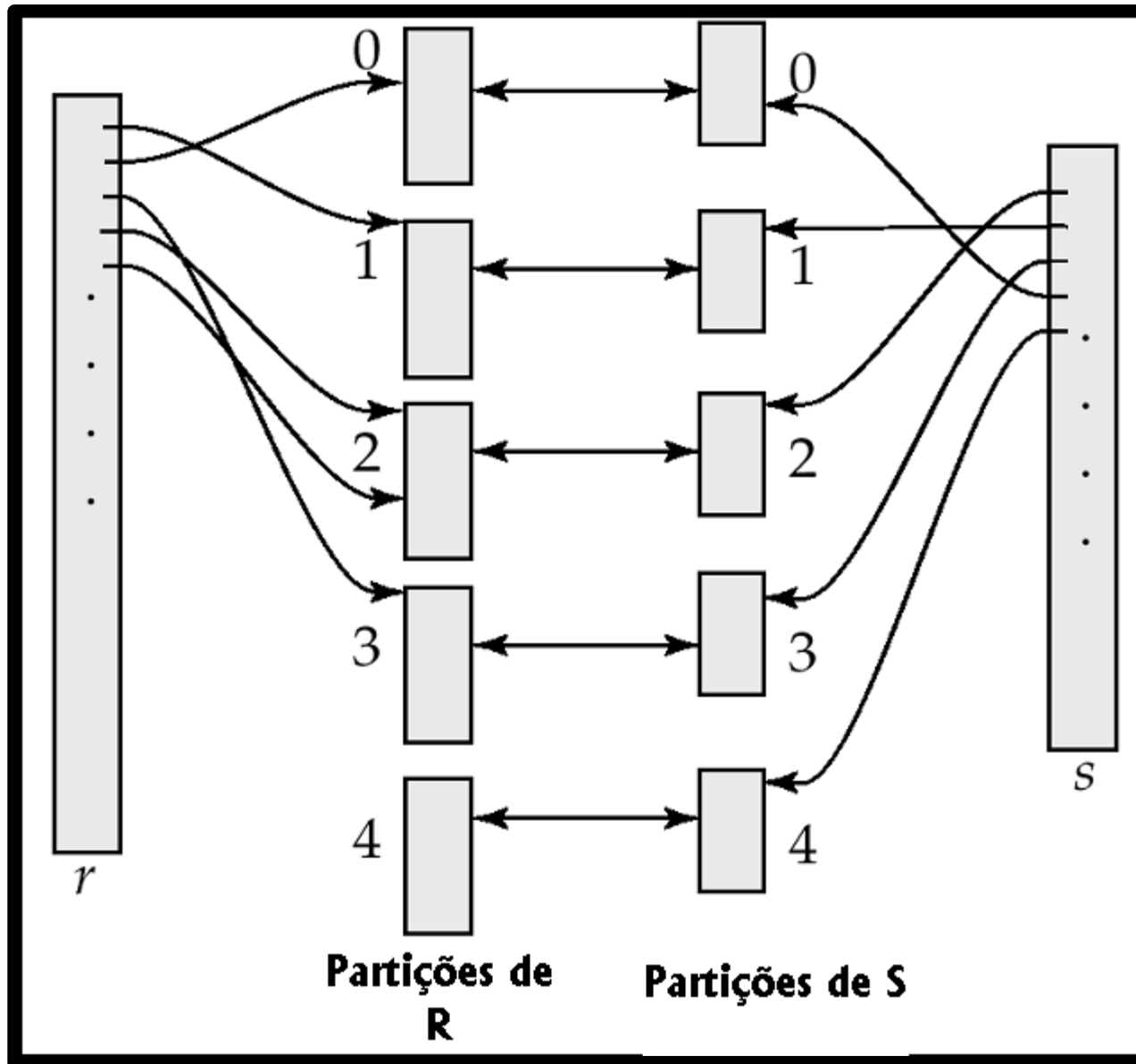
Junção por fusão(Cont.)

- Pode ser usado somente por junções por igual e junções naturais
- Cada bloco precisa ser lido somente uma vez (assumindo que todas as tuplas para algum dado valor do atributo de junção cabem na memória).
- Assim o número de acessos a blocos para a junção por fusão é $b_r + b_s$ + o custo de ordenar se for necessário.
- **Junção por fusão híbrida:** Se uma relação estiver ordenada, e a outra tem um índice secundário B⁺-tree sobre o atributo de junção
 - Fusione a relação ordenada com as entradas das folhas da relação índice (B⁺-tree) .
 - Ordena o resultado de acordo com os endereços das tuplas da relação desordenada
 - 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 a busca aleatória

Junção Hash

- Aplicável para equi-junções e junções naturais.
- Uma função hash h é usada para dividir tuplas das duas relações
- h aplica os *JoinAttrs* a valores $\{0, 1, \dots, n\}$, onde *JoinAttrs* representam os atributos comuns de r e s usados na junção natural.
 - r_0, r_1, \dots, r_n representam as partições de tuplas de r
 - Cada tupla $t_r \in r$ é colocada na partição r_i onde $i = h(t_r[\text{JoinAttrs}])$.
 - s_0, s_1, \dots, s_n representam as partições de tuplas de s
 - Cada tupla $t_s \in s$ é colocada na partição s_i , onde $i = h(t_s[\text{JoinAttrs}])$.

Junção Hash (Cont.)

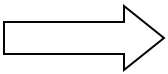


Junção Hash (Cont.)

- *Tuplas de r em r_i unicamente precisam ser comparadas com tuplas de s em s_i . 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 r_i e a tupla de s em s_i .

Algoritmo da Junção Hash

A junção hash de r e s é calculada como segue:

1. Divida a relação s usando a função de hashing h . Quando dividimos uma relação, um bloco de memória é reservado como buffer de saída para cada partição.
2. Divida r de forma similar.  CONSTRUÇÃO
3. Para cada i :
 - (a) Carregue s_i na memória e construa um índice de hash em memória de s usando o atributo de junção. Este índice de associação usa uma função de hash diferente de h .
 - (b) Leia as tuplas em r_i do disco uma por uma. Para cada tupla t_r ache cada tupla t_s em s_i que casa usando o índice de hash em memória. Gere a concatenação de seus atributos.

Relação s é chamada a **entrada de construção** e r é chamada a **entrada de teste**.

Algoritmo da Junção Hash (Cont.)

- O valor n e a função hash h é escolhida de tal forma que cada s_i deveriam caber na memória.
 - Tipicamente n é escolhido como $\lceil b_s/M \rceil * f$ onde f é a “fudge factor”, tipicamente cerca de 1.2
 - As partições da relação de teste r_i não precisam caber na memória

Manipulação de Overflows

- **Overflow na tabela Hash** ocorre na partição s_i se s_i não cabe na memória. Razões poderiam ser
 - Várias tuplas em s com o mesmo valor dos atributos de junção
 - Má função de hash
- A divisão está **sesgada** se algumas divisões têm mais tuplas que outras
- A solução dos overflows pode ser feito na fase de construção
 - divida s_i de novo usando uma função de hash diferente.
 - A divisão r_i deve ser dividida de forma similar.
- Para evitar overflows na fase de construção as divisões devem ser feitas cuidadosamente
 - E.g. divida a relação de construção em muitas divisões, então combina elas até caber na memória
- As duas abordagens falham com um grande número de duplicatas
 - Opção: use junção de laços aninhados por blocos sobre as divisões com overflow.

Custo de uma Junção de Hash

- O custo da junção de hash é
$$3(b_r + b_s) + 2 * n_h$$
- O número de divisões da relação de proba r é o mesmo que para a relação de construção s ; o número de passos para dividir r é também o mesmo que para s .
- Além disso, é melhor escolher a menor relação como a relação de construção.
- Quando a entrada de construção pode ser armazenada totalmente na memória, n_h passa ser 0 e o algoritmo não divide as relações em arquivos temporários. O custo estimado vai para $b_r + b_s$.

Exemplo de Custo da Junção de Hash

cliente ⋈ *dono-conta*

- Assuma que o tamanho da memória é de 20 blocos
- $B_{\text{dono-conta}} = 100$ e $b_{\text{cliente}} = 400$.
- *Dono-conta* é usado como entrada de construção. Divide esta em cinco divisões, cada uma de 20 blocos. Esta divisão pode ser feita num só passo.
- Da mesma forma, a relação *cliente* é dividida em cinco divisões, de tamanho 80. Isto também é feito num só passo.
- O custo total: $3(100 + 400) = 1500$ transferências de blocos
 - ignora custo de escrever os blocos parcialmente cheios

Junções Complexas

- Junção com uma condição conjuntiva:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

- Ou usa laços aninhados/laços aninhados por blocos, ou
- Calcule o resultado de cada uma das junções simples

$$r \bowtie_{\theta_i} s$$

- O resultado final compreende essas tuplas no resultado intermediário que satisfazem o resto das condições

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

- Junção com uma condição disjuntiva

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

- Ou usa laços aninhados/laços aninhados por blocos, ou
- Calcule a união dos registros das junções $r \bowtie_{\theta_i} s$ individuais:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

Outras Operações

- **Eliminação de duplicatas** pode ser implementada via ordenação.
 - Na ordenação, tuplas duplicatas aparecerão consecutivas, e todas menos uma das duplicatas serão removidas.
 - *Otimização*: duplicatas podem ser removidas durante a geração da sequência (run) assim como nos passos intermediários de merge.
 - Hashing é similar – duplicatas aparecerão no processo de partição (mesmo bucket).
- **Projeção** é implementada desenvolvendo a projeção de cada tupla seguida por uma eliminação de duplicatas.

Outras Operações : Agregação

- **Agregação** pode ser implementada de forma similar à eliminação de duplicatas.
 - Ordenação ou hashing podem ser usadas para colocar as tuplas no mesmo grupo juntas, e então as funções agregadas podem ser aplicadas a cada grupo.
 - *Otimização*: combinar tuplas no mesmo grupo durante a geração do run e merges intermediários, calculando valores parciais agregados
 - Para count, min, max, sum: mantêm valores agregados das tuplas achadas até o momento no grupo.
 - Para avg, mantem sum e count, e divide sum por count no final.

Outras Operações : Operações de conjuntos

- **Operações de conjuntos** (\cup , \cap e $-$): podem ou usar variações da junção-merge depois de ordenar elas, ou uma variação da junção hash.
- E.x., Operações de conjuntos usando hashing:
 1. Divida as duas relações usando a mesma função de hash, criando, r_1, \dots, r_n e s_1, s_2, \dots, s_n
 2. Processa cada partição i como segue. Usando uma função de hashing diferente, construa um índice de hash em memória para r_i depois este é levado na memória.
 3. — $r \cup s$: Adicione tuplas em s_i ao índice de hash se eles não estão já neste. No fim de s_i adicione as tuplas no índice hash ao resultado.
 - $r \cap s$: Adicione tuplas de s_i ao resultado se elas já estão no índice de hash.
 - $r - s$: para cada tupla em s_i , se ela está no índice de hash, remova este do índice. No fim de s_i adicione as tuplas remanescentes no índice de hash ao resultado.

Outras Operações : Junção Externa

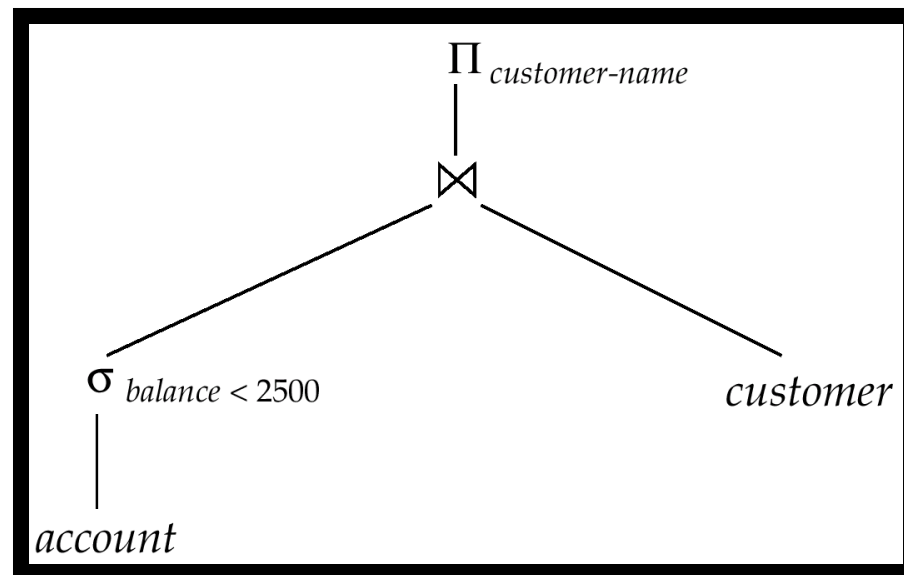
- **A junção externa** pode ser calculada ou como
 - Uma junção seguida pela adição de tuplas que não casam preenchidas com valores nulos.
 - modificação dos algoritmos de junção.
- Modificando o merge join to calcular $r \sqsupset \bowtie s$
 - Em $r \sqsupset \bowtie s$, as tuplas que não participam são essas em $r - \Pi_R(r \bowtie s)$
 - Modifique merge-join para calcular $r \sqsupset \bowtie s$: Durante a fusão, para toda tupla t_r de r que não case com qualquer tupla de s , gere como saída t_r preenchida com nulos.
 - Right outer-join e full outer-join podem ser calculados similarmente.
- Modificando a junção hash para calcular $r \sqsupset \bowtie s$
 - Se r é a relação de teste, gere tuplas de r que não casam preenchidas com nulos
 - Se r é a relação de construção, quando se testa (busca) quais tuplas de r casam com as tuplas de s . No fim de s_i gere tuplas de r que não casam preenchidas com com nulos

Avaliação de Expressões

- Até aqui: nos temos visto algoritmos para operações individuais
- Alternativas para avaliar uma árvore de expressão completa
 - **Materialização**: gera resultados de uma expressão cujas entradas são relações ou são já calculadas, os **materializa** (armazena) no disco.
 - **Pipelining (Canalização)**: passa os resultados (tuples) às operações pai mesmo quando uma operação está sendo executada
- Nos estudaremos as alternativas acima em detalhe

Materialização

- **Avaliação Materializada:** avalia uma operação ao mesmo tempo, iniciando no nível mais baixo. Usa resultados intermediários materializados em relações temporárias para avaliar as operações do seguinte nível.
- E.x., na figura abaixo, calcule e armazene $\sigma_{saldo < 2500}(conta)$ então calcule e armazene sua junção com *cliente*, e finalmente calcule a projeção sobre *nome-cliente*.



Materialização (Cont.)

- Avaliação materializada é sempre aplicável
- Custo de escrever os resultados no disco e ler de novo pode ser bastante alto
- Nossas fórmulas de custo para operações ignoram o custo de gravar os resultados no disco, assim
 - $\text{Custo Total} = \text{Soma de custos das operações individuais} + \text{custo de gravar os resultados intermediários no disco}$
- Double buffering: usa dois buffers de saída para cada operação, quando um está cheio o grava no disco enquanto o outro está sendo preenchido
 - Permite overlap de gravações no disco com processamento e reduz o tempo de execução

Pipelining (Canalização)

- **Avaliação Pipelined:** avalia várias operações simultaneamente, passando os resultados de uma operação para a próxima.
- E.x., na árvore de expressões acima, não é armazenado o resultado de

$$\sigma_{saldo < 2500}(conta)$$

no lugar, passa as tuplas diretamente para a junção. De forma similar, não armazena o resultado da junção, passa as tuplas diretamente para a projeção.

- Mais barata que a materialização: não precisa armazenar uma relação temporária no disco.
- Pipelining pode não ser sempre possível – e.x., ordenação, junção hash.
- Para que pipelining seja efetivo, use algoritmos de avaliação que geram tuplas de saída mesmo que tuplas sejam recebidas como entradas à operação.
- Pipelines pode ser executado de duas formas: **conduzido pela demanda e conduzido pelo produtor**

Pipelining (Cont.)

- Na forma **conduzida pela demanda** ou **avaliação preguiçosa**
 - O sistema repetidamente pede a próxima tupla da operação desde o topo
 - Cada operação pede a próxima tupla das operações filhas, para gerar sua próxima tupla
 - Entre as chamadas, a operação tem que manter o “**estado**” para saber o que retornar a próxima vez
 - Cada operação é implementada como um **iterador** que implementa as seguintes operações
 - abrir()
 - E.x. file scan: inicia uma exploração do arquivo, armazena como estado o ponteiro ao início do arquivo
 - E.x junção merge: ordena as relações e o estado inicial são os apontadores ao início das relações ordenadas
 - próximo()
 - E.g. para file scan: gera próxima tupla, e avança e armazena o ponteiro
 - E.g. para junção merge: continue com o merge a partir do último estado até que a próxima tupla de saída é achada. Salva apontadores como estado do iterador.
 - fecha()

Pipelining (Cont.)

- Na forma conduzida pelo produtor ou pipelining impaciente
 - Os Operadores produzem tuplas impacientemente e as passam até seus pais
 - Buffer mantido entre operadores, filho coloca tuplas no buffer, pai remove tuplas do buffer
 - Se o buffer estiver cheio, o filho espera até que exista um espaço no buffer, e então gera mais tuplas
 - O sistema programa as operações que têm espaço no buffer de saída e podem processar mais tuplas de entrada

Algoritmos de Avaliação para Pipelining

- Alguns algoritmos não conseguem gerar resultados na medida de que eles obtêm tuplas de entrada
 - E.g. merge join, ou junção hash
 - Estes geram resultados intermediários a serem escritos em disco e então lidos de novo sempre
- Variações dos algoritmos são possíveis para gerar resultados na medida que as tuplas de entrada são lidas