PUCRS
Pontifícia Universidade Católica
do Rio Grande do Sul

ESCOLA
POLITÉCNICA

Otimização Baseada em Custos

98H00-04 - Infraestrutura para Gestão de Dados

Prof. Msc. Eduardo Arruda
eduardo.arruda@pucrs.br

Otimização baseada em custos

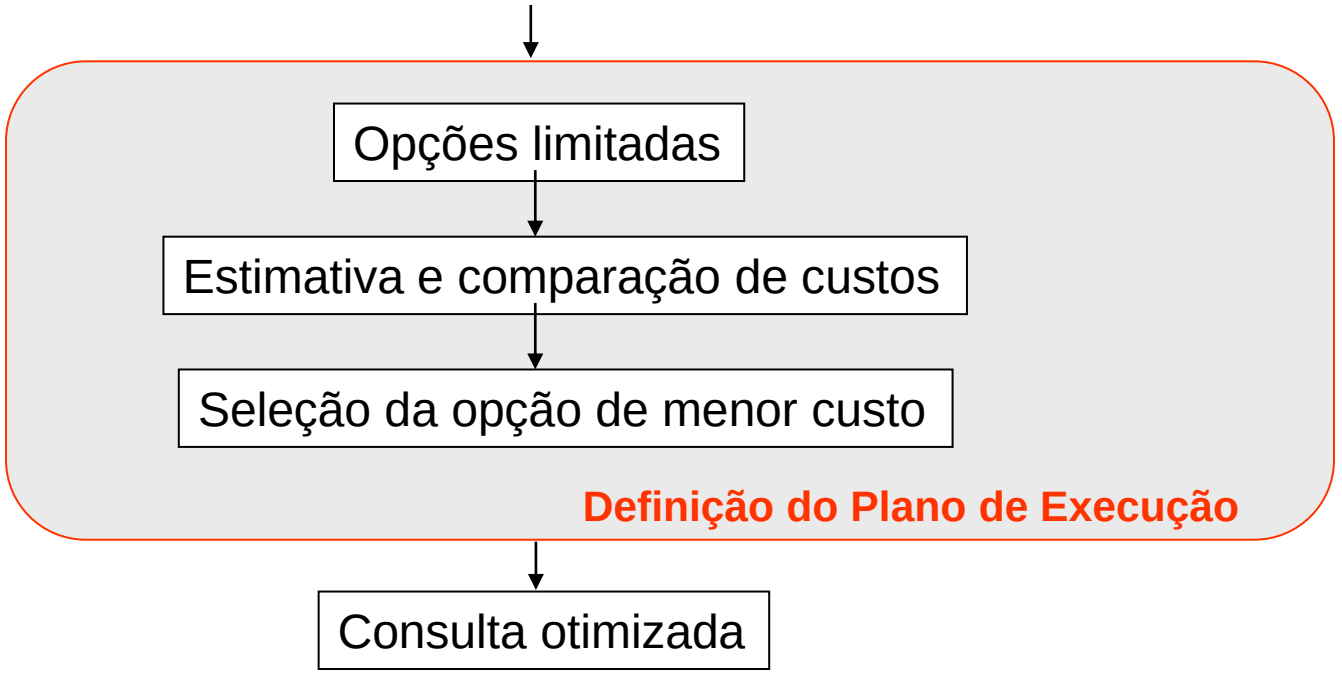
- Estimar e comparar os custos de executar uma consulta usando diferentes estratégias
  - Escolher a estratégia com o menor custo estimado
- Formas alternativas de avaliar uma dada consulta
  - Expressões equivalentes
  - Diferentes algoritmos para cada operação
- Diferença de custo pode ser enorme
  - Exemplo: realizar um r X s seguido de uma seleção r.A = s.B é muito mais lento do que fazer uma junção na mesma condição

Otimização baseada em custos

- Necessidade de estimar o custo das operações
  - Depende de informações estatísticas sobre as relações que o banco mantém
    - Ex. Núm. tuplas, núm. de valores distintos para atributos de junção
- Questões a considerar
  - Custo da função
  - Número de estratégias a ser considerado
- Componentes de custo
  - Acesso à disco
  - CPU
  - Comunicação em rede, etc.
- Nota: diferentes SGBDs podem focar em diferentes componentes de custo

Otimização usando custos estimados

- As funções de custo utilizadas na otimização de consultas são estimativas e não medidas exatas de custo
- Assim, a otimização pode selecionar uma estratégia de execução que não seja a melhor



Medidas de custo

- Relembre que
  - Tipicamente acessos a disco são o custo predominante e é relativamente fácil de estimar
  - O número de transferências de datablock do disco é usado como medida para o custo de avaliação
  - É assumido que todas as transferências de datablock têm o mesmo custo
    - Otimizadores reais distinguem as transferências e avaliam vários componentes
- Não são considerados aqui os custos de escrita no disco, pois, em tese, são iguais para todos os métodos, pois o resultado final é o mesmo

Estimativas sobre os dados

n_R	número de tuplas na tabela R
t_R	tamanho (em bytes) de uma tupla de R
$t_R(a_i)$	tamanho (em bytes) do atributo a_i de R
f_R	fator de bloco de R (quantas tuplas de R cabem em um datablock*) * datablock: unidade de R / W em disco (medida básica de avaliação) $f_R = \lfloor t_{bloco} / t_R \rfloor$
$V_R(a_i)$	número de valores distintos do atributo a_i de R
$C_R(a_i)$	cardinalidade (estimada) do atributo a_i de R (tuplas de R que satisfazem um predicado de igualdade sobre a_i) (estimando distribuição uniforme: $C_R(a_i) = n_R / V_R(a_i)$
$GS_R(a_i)$	grau de seletividade do do atributo a_i de R (estimando distribuição uniforme : $GS_R(a_i) = 1 / C_R(a_i) = V_R(a_i)/n_{R^*}$
b_R	número de datablocks necessários para manter tuplas de R $b_R = \lceil n_R / f_R \rceil$

Faixas de Valores

Estimativa	Atributo Chave	Atributo não chave	Atributo Booleano	Atributo "Data Aniversário"
$V_R(a_i)$	n_R	1 até n_R-1	2	366
$C_R(a_i) = n_R/V_R(a_i)$	1	1 até n_R	$n_R/2$	$n_R/366$
$GS_R(a_i) = V_R(a_i)/n_R$	1	<1	$2/n_R$	$366/n_R$=0,0027

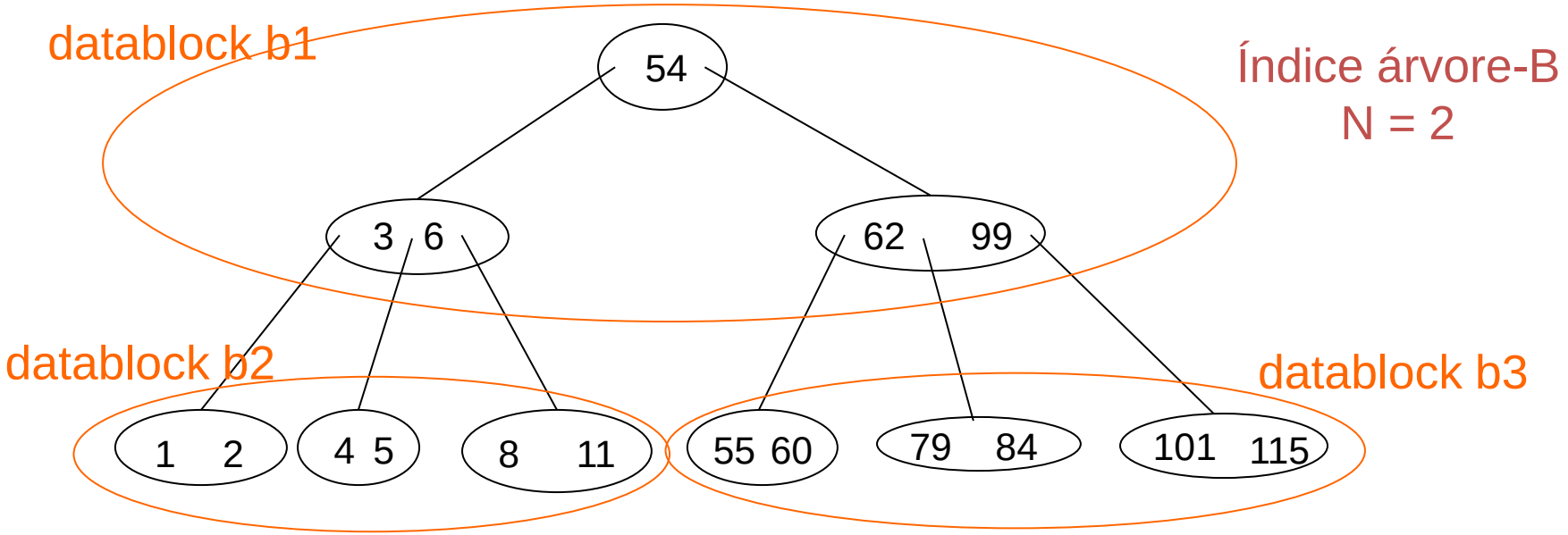
Exemplo de Estimativas de Tabela

- Existem 10000 Contas correntes cadastradas na tabela Contas; cada tupla possui 200 bytes em média e 1 datablock lê/grava 4 kb (=4096 bytes); contas estão distribuídas em 50 agências.
- Estimativas:
- nContas = 10000 tuplas na relação contas
- tContas = 200 bytes (tamanho de tupla)
- fMContas = $\lfloor 4096 / 200 \rfloor$ = 20 tuplas/datablock
- bContas = $\lceil 10000 / 20 \rceil$ = 500 datablocks (necessários para manter as tuplas de contas)
- Vconta(agencia) = 50 (50 agências diferentes)
- Cr(agencia)= nContas /Vconta(agencia) = 10000/50 = 200
- GSr(agencia)= Vconta(agencia)/nContas = 50/10000 = 0,005

Estimativas sobre os índices B+Tree

f_i	fator de bloco do índice i (fan-out do índice), ou seja, quantos nodos de uma árvore-B cabem em um *datablock*
h_i	número de níveis (de datablocks) do índice para valores de um atributo a_i (“altura” do índice) $h_i = \lceil \log_{fi} \lceil V_R(a_i) / N \rceil \rceil$ (para índices árvore-B) (N é o número de valores que cabem em um nodo) - h_i é a altura do índice (em datablocks), ou seja, quantos datablocks de índice árvore-B devem ser percorridos para se alcançar os datablocks de dados indexados
bf_i	número de datablocks de índice no nível mais baixo do índice (número datablocks “folha”)

Exemplo de Estimativas de Índice



- Estimativas
  - fíndice = 3 nodos cabem em um datablock
  - híndice = ⌈logfi ⌈VR (ai) / N⌉ ⌉ = log3 ⌈17 / 2⌉ = 2 datablocks devem ser percorridos até chegar nos datablocks de dados
  - bfíndice = 2 datablocks no nível folha (6 nodos folha / 3)

Processamento de Seleções (σ)

- Alternativas e suas estimativas de custo
  - A1: pesquisa linear (força bruta)
  - A2: pesquisa binária
  - A3: índice primário para atributo chave
  - A4: índice clustering para atributo não-chave
  - A5: índice secundário para atributo chave
  - A6: índice secundário para atributo não-chave
  - A7: desigualdade (>, >=, <, <= com índice primário
  - A8: desigualdade com índice secundário
  - A9: seleção conjuntiva com índices individuais
  - A10: seleção conjuntiva com índice composto
  - A11: seleção conjuntiva com intersecção de índices
  - A12: seleção disjuntiva

Pesquisa linear (A1)

- Varre todo o arquivo para buscar os dados desejados
  - acessa todos os datablocks do arquivo
- Em alguns casos, é a única alternativa possível
- Custo para uma tabela R
  - custo = b_R
    - b_R = número de datablocks necessários para manter tuplas de R
  - Se a seleção for em um atributo chave, supomos que a metade dos datablocks é varrida antes de o registro ser encontrado
    - Custo = b_R / 2

Pesquisa binária (A2)

- Aplicado sobre uma tabela R quando
  - dados estão ordenados pelo atributo de seleção a_i
  - há uma condição de igualdade sobre a_i
- Custo
  - custo para acessar o datablock da 1ª tupla: $\lceil \log_2 b_R \rceil$
  - custo para acessar os datablocks das demais tuplas: $\lceil (C_R(a_i) / f_R) \rceil - 1$
  - custo = $\lceil \log_2 b_R \rceil + \lceil (C_R(a_i) / f_R) \rceil - 1$
  - se a_i é chave: custo = $\lceil \log_2 b_R \rceil$

Pesquisa binária (A2) - Exemplo

- f_{conta} = 20 (20 tuplas de conta cabem em um datablock)
- V_{conta}(nome_agencia) = 50 (50 agências diferentes)
- n_{conta}=10 000 (a relação conta possui 10.000 tuplas)
- Seja $\sigma_{nome\_agencia=\text{“PUCRS”}}$(conta)
  - b_{conta}=500 datablocks necessários para manter a relação (10.000/20)
    - Se usar varredura linear: 500 acessos
  - Suponha que conta esteja ordenado por nome_agencia
    - Como V(nome_agencia, conta) = 50
      - 10.000/50 = 200 tuplas da relação pertencem à agência PUC
      - Estas tuplas caberiam em 200/20 = 10 datablocks
      - Busca binária para o primeiro registro: $\log_2 500 = 9$ acessos
      - Custo total da busca binária: $\lceil \log_2 500 \rceil + \lceil (10.000/50) / 20) \rceil - 1 = 9+10-1=18$

Seleções utilizando índices

- Atributo a_i com índice primário ou *clustered* ou INTERNO
  - leitura do índice corresponde à leitura na ordem física do arquivo
    - arquivo fisicamente ordenado por valores de a_i
  - se a_i é chave (A3)
    - custo = h_i
  - se a_i é não-chave (A4)
    - custo = $h_i + \lceil (C_R(a_i) / f_R) \rceil - 1$

Pesquisa índice *clustered* (A4) - Exemplo

- f_{conta} = 20 (20 tuplas de conta cabem em um datablock)
- V_{conta}(nome_agencia) = 50 (50 agências diferentes)
- n_{conta} = 10 000 (a relação conta possui 10.000 tuplas)
- Seja $\sigma_{nome\_agencia=\text{“PUCRS”}}$(conta)
  - b_{conta} = 500 datablocks necessários para manter a relação (10.000/20)
    - Se usar varredura linear: 500 acessos
  - Suponha que conta esteja ordenado por nome_agencia
    - Como V(nome_agencia, conta) = 50
      - 10.000/50 = 200 tuplas da relação pertencem à agência PUCRS
      - Estas tuplas caberiam em 200/20 = 10 datablocks
      - Custo de pesquisar pelo índice *clustering* = custo = $h_i + \lceil (C_R(a_i) / f_R) \rceil$
      - Índice com 50 entradas caberia em 1 nível (datablock)
      - Custo = $1 + \lceil (10.000/50) / 20) \rceil = 1 + 10 = 11$

Seleções utilizando índices secundários

- Atributo a_i com índice secundário ou *non-clustered* ou EXTERNO
  - arquivo não está fisicamente ordenado por valores de a_i
  - se a_i é chave ou é único (A5)
    - custo = $h_i + 1$
  - se a_i é não-chave (A6)
    - supor que o datablock folha do índice aponta para uma lista de apontadores para as tuplas desejadas
      - Estimar que esta lista cabe em um datablock
    - custo = $h_i + 1 + C_R(a_i)$ (pode ser menor se distribuição não for homogênea e for possível de alguma forma ordenar os ponteiros por datablock)

Pesquisa índice secundário (A6) - Exemplo

- f_{conta} = 20 (20 tuplas de conta cabem em um datablock)
- V_{conta}(nome_agencia) = 50 (50 agências diferentes)
- n_{conta} = 10.000 (a relação conta possui 10.000 tuplas)
- Seja $\sigma_{nome\_agencia}$="PUC"(conta)
  - b_{conta} = 500 datablocks necessários para manter a relação (10.000/20)
    - Se usar varredura linear: 500 acessos
  - Suponha que conta NÃO esteja ordenado por nome_agencia, mas que existe um índice secundário por este campo
    - Como V(nome_agencia, conta) = 50
      - 10.000/50=200 tuplas da relação pertencem à agência PUC
      - Estas tuplas podem estar em qualquer datablock
      - Custo da busca pelo índice secundário: $h_i + 1 + C_R(a_i)$
      - Índice com 50 entradas caberia em 1 nível (*datablock*)
      - $C_R(a_i)$ = 10.000/50 = 200
      - Custo = 1 + 1 + 200 = 202 acessos a *datablocks*

Exercício 1

- Relação = Pac (codp, nome, idade, cidade, doença)
- Estimativas: n_{Pac} = 1000 tuplas; t_{Pac} = 100 bytes; V_{Pac}(codp) = 1000; V_{Pac}(doença) = 80; V_{Pac}(idade) = 50; um índice primário B+Tree para codp (I1) com H_i = 5; fI1 = 10; um índice secundário B+Tree para doença (I2) com H_i = 3; fI2 = 5; um índice secundário B+Tree para Idade (I3) com H_i = 1; fI3 = 10; datablock = 2 kb
- Supondo as seguinte expressões algébricas:
  1. $\sigma_{\text{doença = 'câncer'}}$ (Pac)
  2. $\sigma_{\text{codP = 52}}$ (Pac)

- Quais os custos para processar 1, usando busca linear e indexada?
- Quais os custos para processar 2, usando busca linear, pesquisa binária e indexada?

Intervalos abertos

- Recuperando registros com condições de intervalo aberto: $\sigma_{A \leq V}(r)$ ou $\sigma_{A \geq V}(r)$
- Busca linear ou binária: idem p/ valor exato
- Algoritmo A7 (índice primário):
  - O arquivo está ordenado pelo atributo A
    - Para $\sigma_{A \geq V}(r)$: usa-se índice para encontrar a primeira tupla que satisfaz a condição e depois percorre-se o arquivo sequencialmente até o fim
    - Para $\sigma_{A \leq V}(r)$: apenas percorre-se o arquivo até encontrar o primeiro valor que não satisfaz a condição. NÃO se usa índice

Intervalos abertos

- Algoritmo A8 (índice secundário):
  - Para $\sigma_{A \geq V}(r)$: Usa-se índice para encontrar a primeira entrada >= v do índice e percorre-se o índice sequencialmente a partir daí, para encontrar os ponteiros para os registros que contém os dados
    - Custo = $h_i + bf_i/2 + b_R/2$
  - Para $\sigma_{A \leq V}(r)$: apenas percorre-se os nós-folhas do índice, encontrando os ponteiros para os registros, até a primeira entrada > v
    - Em ambos os casos, a recuperação dos registros apontados exige uma leitura de *datablock* para cada registro.
    - Busca linear poderá ser mais barata, se há muitos registros a serem recuperados (utiliza-se $GS_R(a_i)$)
    - Custo = $bf_i/2 + b_R/2$

Seleções complexas

- Conjunções: $\sigma_{c1 \wedge c2 \wedge \ldots cn}(r)$
  - A9 (cada seleção possui um índice individual):
    - Escolhe-se o ci (com um dos algoritmos de A1 a A7) que resulta no menor custo para $\sigma_{ci}(r)$
    - Para cada tupla recuperada, testa-se cada uma das outras condições em memória, antes de adicionar a tupla ao resultado final
    - Custo é o mesmo do índice secundário (A6)
  - A10 (existem índices compostos):
    - Usa-se o índice composto disponível
    - Custo é o mesmo do índice secundário (A6)
  - A11 (interseção de identificadores - *buckets*):
    - Exige índices com ponteiros para registros
    - Usa-se o índice correspondente de cada condição e, a partir da interseção dos mesmos, busca-se no disco. Aplica-se teste em memória para condições que não possuem índices apropriados.
    - Custo = $bf_1 + bf_2 + C_R(a_{1,2})$
    - Pode ser melhor fazer pesquisa linear

Seleções Complexas

- Disjunções: $\sigma_{c1 \vee c2 \vee \ldots cn}(r)$
  - A12 (se ambas as operações possuírem índices nos campos envolvidos na operação, pode-se dividir em duas consultas e depois combinar o resultado)
    - Custo = $h_i + 1 + C_R(a_i) + h_i + 1 + C_R(a_i)$ + custo de combinar (devem ser eliminadas as duplicatas)
    - Pode ser melhor a pesquisa linear

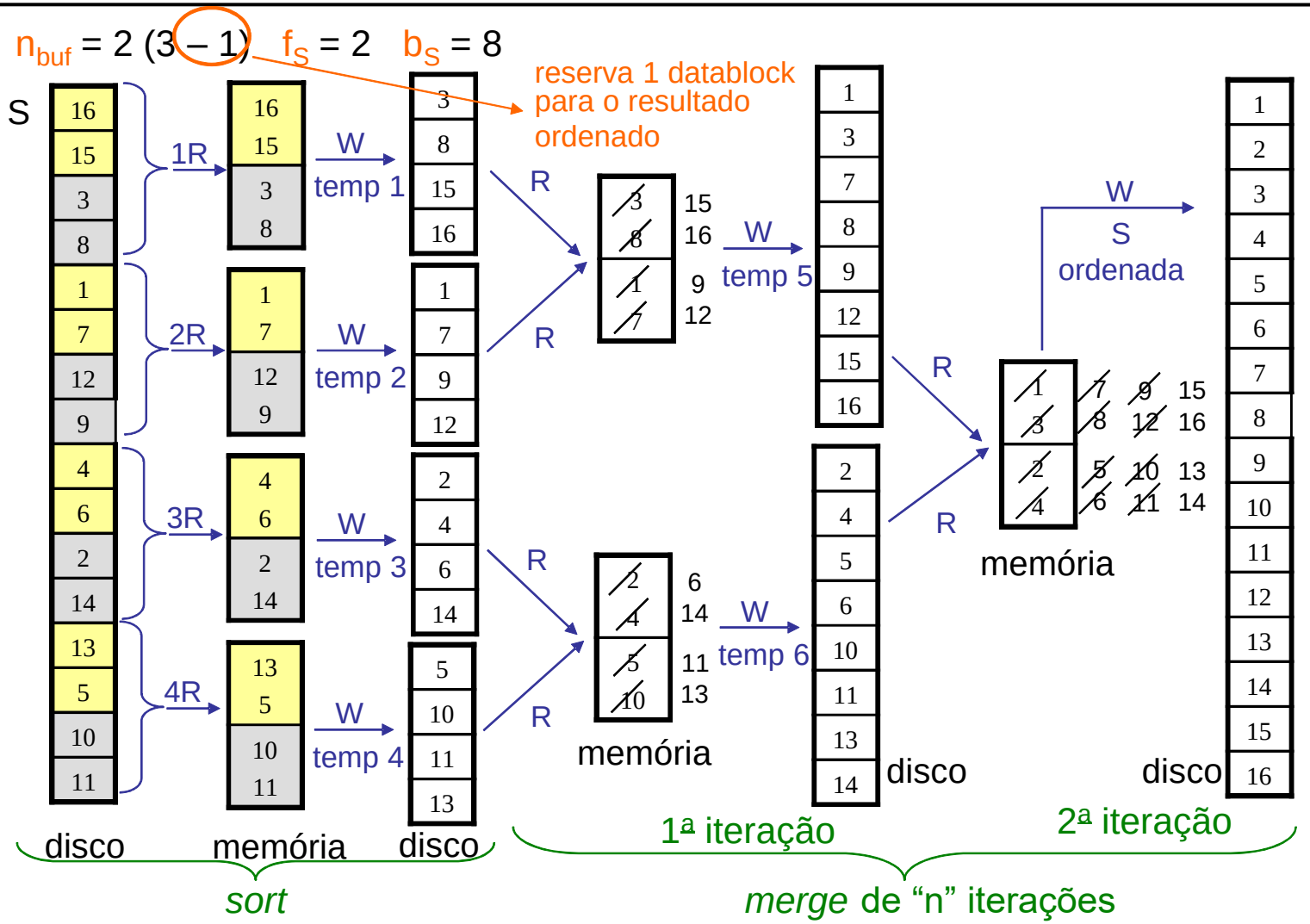
Classificação

- Para obter a relação classificada diretamente do disco, teríamos que ter um índice para o atributo e usar o índice para ler a relação de forma ordenada (percorrendo as folhas, se for o caso)
  - Isso pode levar a uma leitura de um *datablock* para cada tupla → Muito caro!
- Se a relação resultante cabe na memória → usar técnicas de ordenação (quicksort, p.ex.) depois de recuperar pelos métodos normais
- Se a relação resultante não cabe na memória, a técnica SORT-MERGE EXTERNO é uma boa opção
  - Essa técnica consiste em recuperar partes da relação, ordenar, gravar de novo, fazer "merge" das partes, e assim sucessivamente até obter a relação toda ordenada

Sort merge externo

- Executa em 2 etapas
  - Etapa 1 – Sort
    - ordena partições da relação em memória
      - tamanho da partição depende da disponibilidade de *buffers* em memória (n_{buf} = número de *buffers* disponíveis)
      - gera um *resultset* temporário ordenado para cada partição
  - Etapa 2 – Merge de “n” iterações
    - ordena um conjunto de *resultsets* temporários a cada iteração
      - gera um novo temporário resultante da ordenação
      - ordenação termina quando existir somente um temporário que contém a relação inteira ordenada

Sort merge externo - Exemplo



Sort merge externo - Exemplo

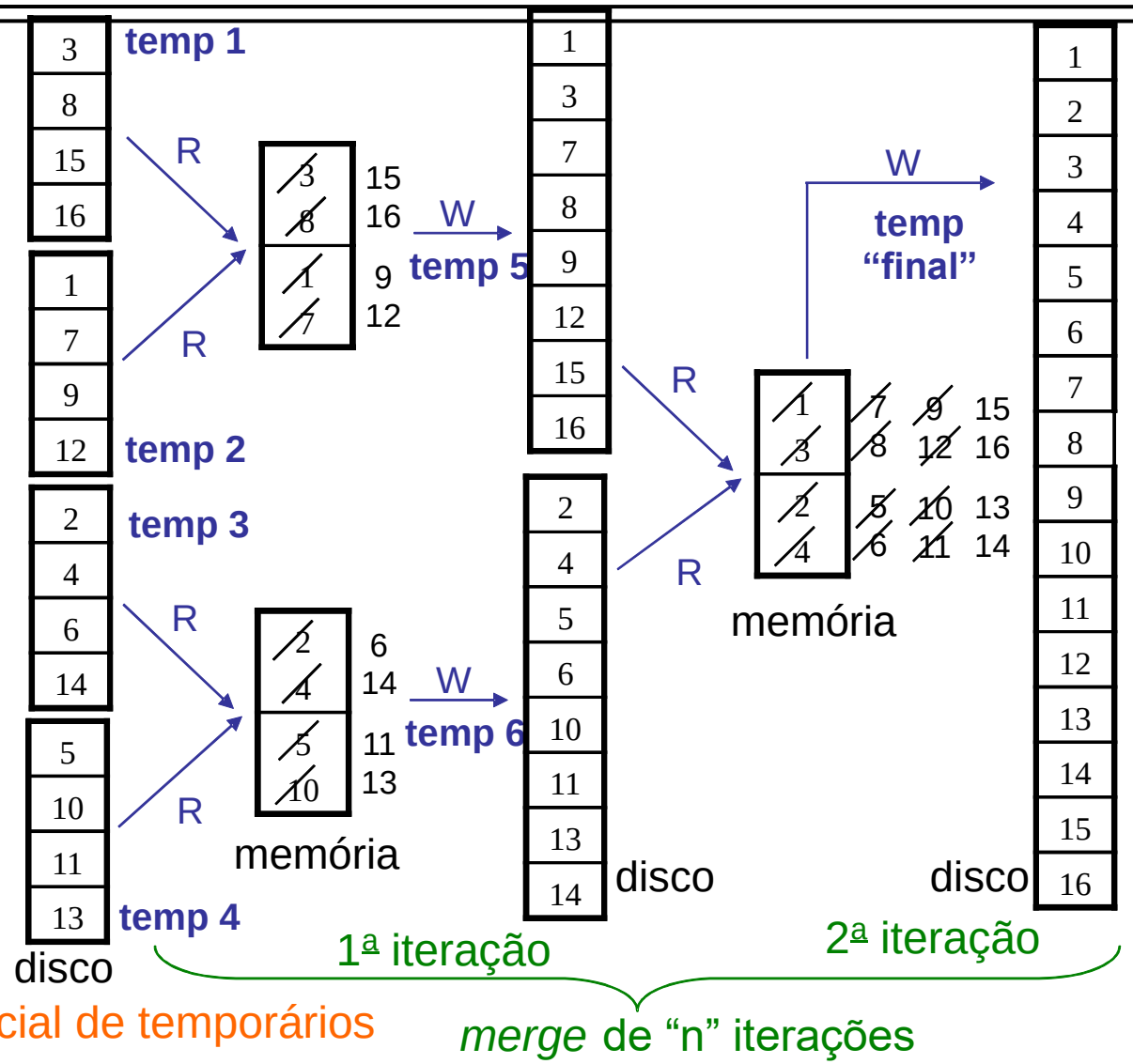
$n_{buf} = 2 \quad f_S = 2 \quad b_S = 8$

Nº de iterações é dependente do nº de temporários a ordenar

A cada iteração, o nº de temporários se reduz a um fator de n_{buf}

- reserva-se 1 buffer para cada temporário. Logo ordena-se n_{buf} temporários a cada iteração
- nº iterações: $\log_{n_{buf}} (b_S / n_{buf})$
- 1 R + 1 W a cada iteração: $2 * b_S$

custo = **$2 * b_S * \log_{n_{buf}} (b_S / n_{buf})$**



Sort merge externo

- Custo = $2 * b_S * \log_{n_{buf}} (b_S / n_{buf})$
  - Fortemente dependente do número de *buffers* disponível para a ordenação
  - Mesmo com n_{buf} pequeno, o custo ainda é linear $O(b_s * \log(b_s))$

Junção

- Diversos algoritmos diferentes para implementar junções:
  - *Nested loops join* por linha
  - *Nested loops join* por bloco
  - *Nested loops join* com índice
  - *Sort merge join*
  - *Hash join*
- A escolha, como sempre, é baseada na estimativa do custo

Nested loops join por bloco

- Para computar a junção teta: $R \bowtie_{(C)} S$
- Vamos chamar **R** de relação externa e **S** de relação interna da junção
- Cada bloco da relação externa será emparelhado com cada bloco da relação interna:

for each block B_R of R do begin
  for each block B_S of S do begin
    for each tuple t_R in B_R do begin
      for each tuple t_S in B_S do begin
        Verifique se (t_R ,t_S) satisfazem a condição
        Se satisfazem, adicione $t_R \bullet t_S$ ao resultado

Nested loops join por bloco

- No pior dos casos, teremos os seguintes custos:

  acessar cada bloco de R:

  custo = b_R

  para cada bloco de R, acessar cada bloco de S e comparar as tuplas:

  custo = $b_R * b_S$
- Custo de leitura = $b_R + b_R * b_S$
- Com isso, é melhor que a tabela externa seja a menor

Nested loops join com índice

- Se tivermos um índice para a relação interna:
  - Para cada tupla t_R da relação externa R, usa-se o índice para encontrar tuplas de S que satisfazem a condição da junção com a tupla t_R
- Pior caso
  - o *buffer* possui espaço para apenas um bloco de R e, para cada tupla de R, realiza-se uma procura no índice de S
- Custo da junção: $b_R + n_R * c$
  - onde c é o custo para percorrer o índice e recuperar todas as tuplas de S que casam com uma tupla de R
  - c pode ser estimado como sendo o custo de uma seleção única em S usando a condição da junção
- Também nesse caso, deve-se usar a relação com menos tuplas para ser a relação externa, caso ambas as relações possuam índices

Exemplo

- Considere:
  - tabela Cliente: 10.000 tuplas , $b_{Cliente} = 400$,
  - tabela Aplicação: 5.000 tuplas , $b_{Aplicação} = 100$
- Vamos computar Cliente $\bowtie_{(C)}$ Aplicação, com Aplicação sendo a relação externa
- Considere que Cliente possui um índice primário de B+Tree com 10 entradas por nó, no atributo que faz a junção: cod_cliente
- Com 10.000 tuplas, a altura da árvore será 4 e portanto o nro de acessos será 5 para cada entrada
- Custo do *nested loops join* por bloco:
  - 100 + 100 * 400 = 40.100 acessos a disco
- Custo do *nested loops join* com índice:
  - 100 + 5000 * 5 = 25.100 acessos a disco

Sort merge join

- Se ambas as relações (R e S) estão ordenadas
  - custo = $b_R + b_S$
- Se uma delas (R) não está ordenada
  - custo = $2 * b_R (\log n_{buf} (b_R / n_{buf}) + 1) + b_R + b_S$
- Se ambas as relações não estão ordenadas
  - custo = $2 * b_R (\log n_{buf} (b_R / n_{buf}) + 1) +$
    $2 * b_S (\log n_{buf} (b_S / n_{buf}) + 1) + b_R + b_S$
- Pouco eficiente a não ser que o resultado deva ser ordenado pela chave de join

Hash Join

- Melhor caso: hash da tabela R cabe todo em memória
- Fase de Particionamento
  - Lê toda tabela R (b_R)
- Fase de Junção
  - Lê toda tabela S
- Custo Total
  - custo = $b_R + b_S$

Hash Join

- Caso normal: executa o hash somente da chave da tabela R – tabela com a *foreign key* – normalmente cabe todo em memória)
- Fase de Particionamento
  - Lê toda tabela R (b_R)
- Fase de Junção
  - Lê toda tabela S (b_S) e para cada registro que atende o join lê o registro de R
- Custo Total
  - custo = $b_R + (b_S + n_{Resultado} * b_R)$
  - $n_{Resultado}$ depende do número de linhas que o join retorna, normalmente o número de linhas da tabela que é filho (R)
  - Quanto maior for a tabela filho, menos eficiente o hash join

Execução encadeada de operações

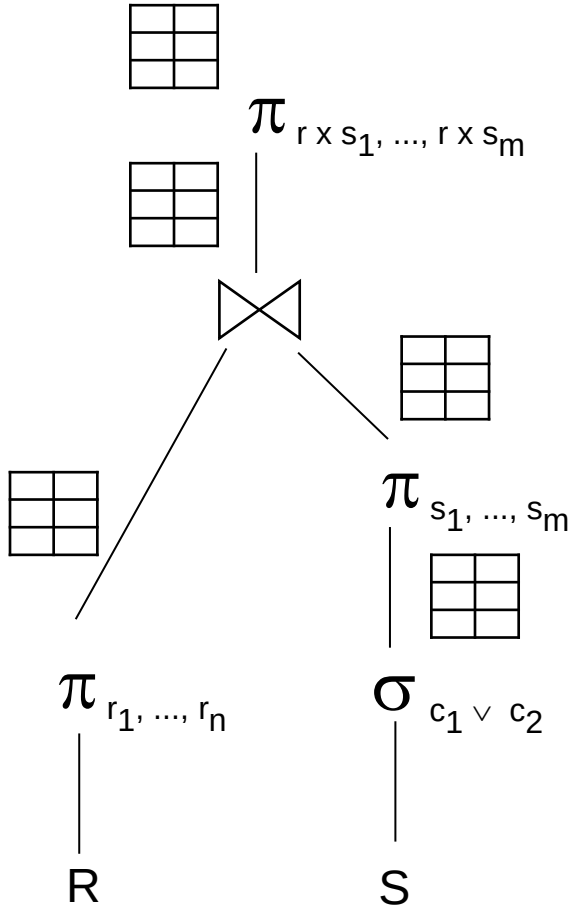
- A execução de uma consulta é um encadeamento da execução das partes da árvore de expressão algébrica
- Para avaliar completamente uma árvore de expressão, há duas alternativas:
  - MATERIALIZAÇÃO: gera resultados para cada operação, cujas entradas são relações pré-existentes ou já computadas, e materializa (armazena em disco) esse resultado. Repete esse processo até atingir o resultado final.
  - PIPELINING: executa todas as operações (em cada tupla, quando possível) do começo ao fim, mantendo no *buffer* os resultados intermediários.
    - Resultado de uma operação é passado para a operação seguinte, sem armazenamento de resultados intermediários.

Materialização X Pipeline

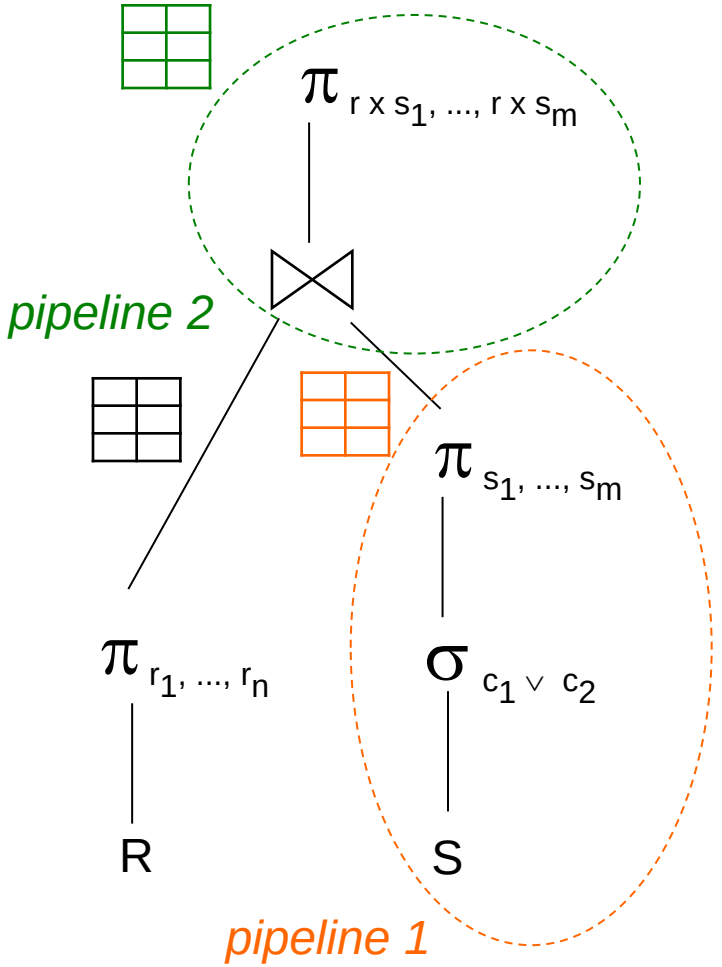
- Materialização
  - cada operação da álgebra é materializada em uma relação temporária (se necessário) e utilizada como entrada para a próxima operação
  - situação padrão no processamento de consultas
- Pipeline
  - uma sequência de operações algébricas é executada em um único passo
    - cada tupla gerada por uma operação é passada para a operação seguinte
      - cada tupla passa por um canal (*pipe*) de operações
    - somente o resultado ao final do pipeline é materializado (se necessário)

Materialização X Pipeline

Materialização



Definição de *Pipelines*

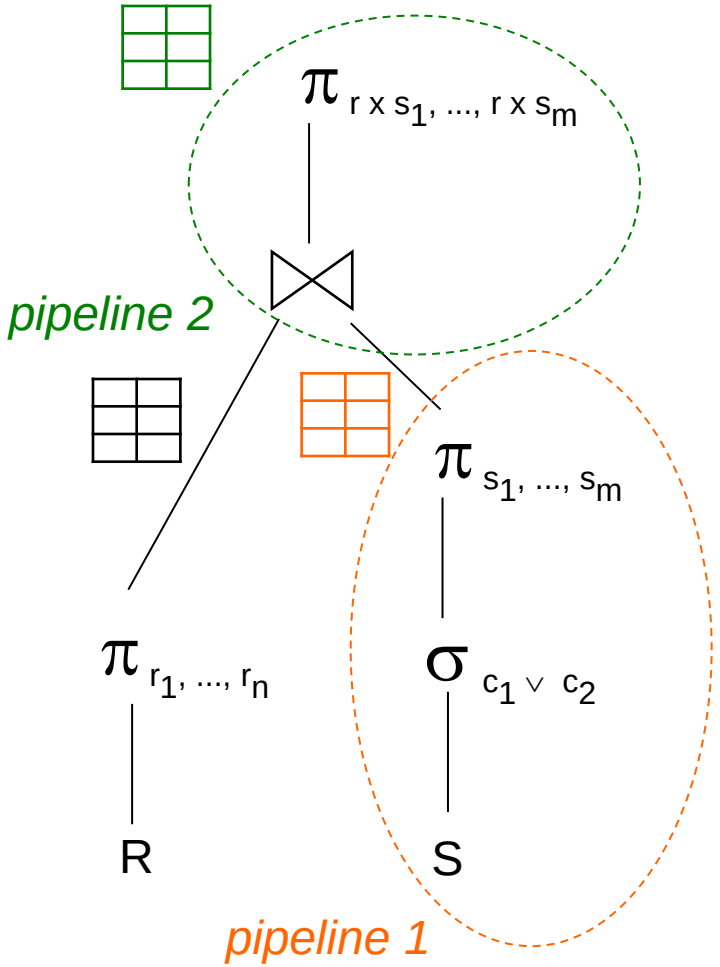


Pipeline de Operações

- + : evita a materialização de todos os resultados intermediários no processamento de uma consulta
- - : resultado não é passado de forma completa para uma próxima operação dentro do pipeline
  - algoritmos de processamento das operações algébricas deve ser modificados para invocar outras operações para cada tupla gerada
    - algoritmos “dinâmicos”
  - algumas alternativas não podem ser estimadas
    - exemplos: sort merge join; operações de conjunto
      - exigem um resultado completo e ordenado para processar

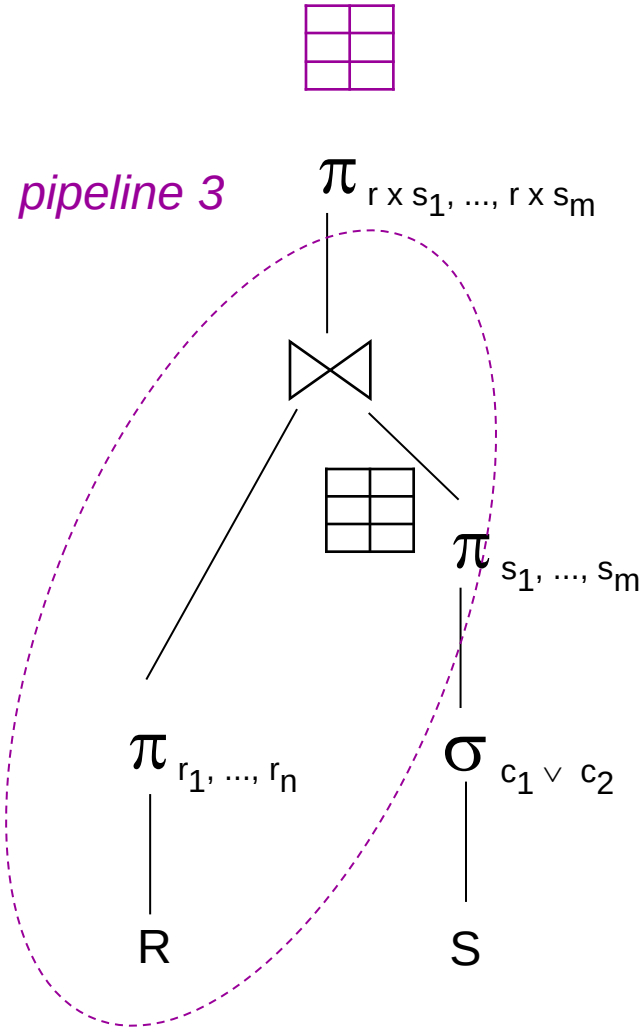
Uso mais comum de pipelines

- Em uma sequência de operações que
  - inicia em um nodo folha ou uma operação binária
  - termina ou no resultado da consulta ou em uma operação binária ob_x, sem incluir ob_x



Uso mais comum de pipelines

- Em uma sequência composta apenas por operações π e operações produtórias, a partir de um nodo folha ou uma operação binária ob_x, incluindo ob_x
  - considera que o tamanho dos resultados intermediários das operações π é muito grande para ser materializado
    - mesmo assim, avaliar se o custo das operações produtórias não aumenta com o pipeline...



Dúvidas?