



CEFET/RJ

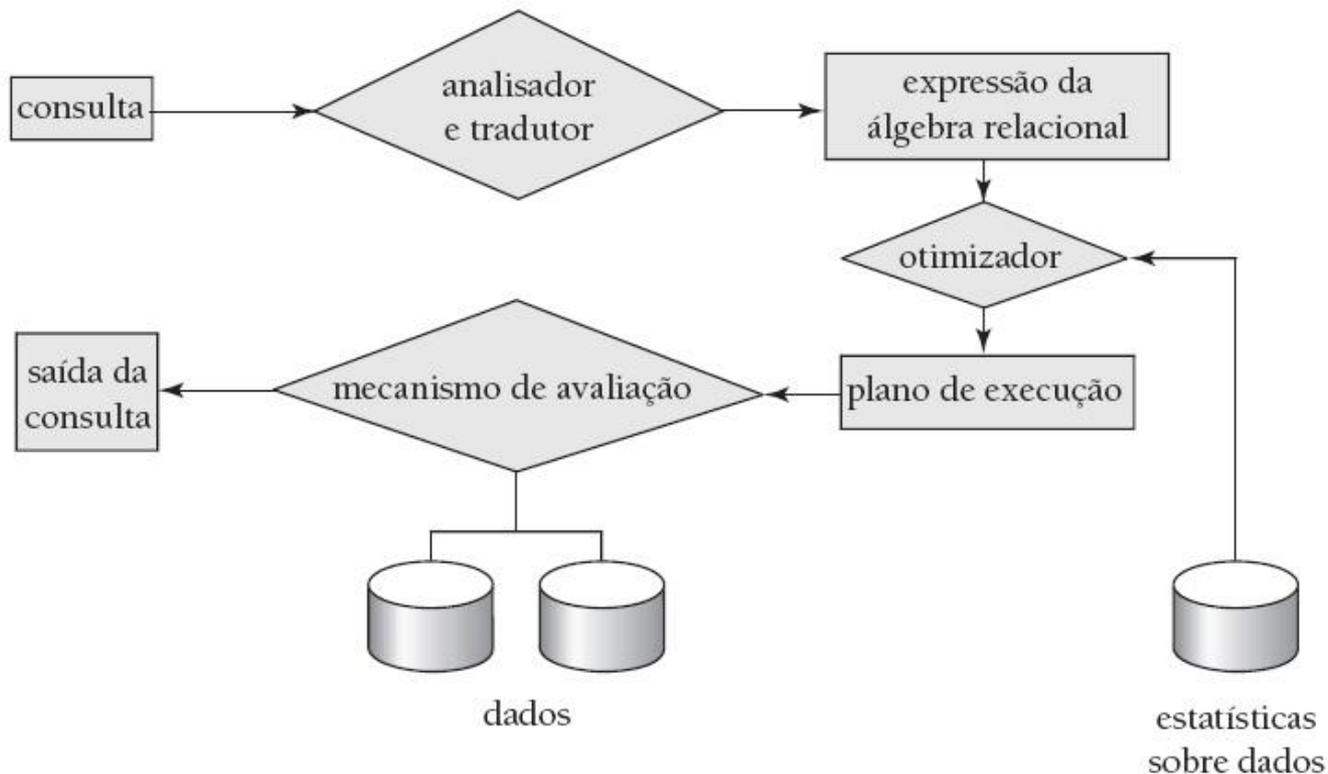


PROCESSAMENTO DE CONSULTA

Eduardo Ogasawara
eogasawara@ieee.org
<https://eic.cefet-rj.br/~eogasawara>

Etapas básicas no processamento da consulta

- Análise e tradução
- Otimização
- Avaliação



Etapas básicas no processamento da consulta: Análise e tradução & Avaliação

- Análise e tradução
 - Traduz a consulta para o seu formato interno
 - Este é então traduzido para a álgebra relacional
 - O analisador verifica a sintaxe e as relações
- Avaliação
 - O mecanismo de execução de consulta produz um plano de avaliação de consulta
 - Executa esse plano
 - Retorna as respostas à consulta

Etapas básicas no processamento da consulta: Otimização

- Uma expressão da álgebra relacional pode ter muitas expressões equivalentes
 - $\sigma_{saldo < 2500}(\pi_{saldo}(conta))$ é equivalente a $\pi_{saldo}(\sigma_{saldo < 2500}(conta))$
- Cada operação da álgebra relacional pode ser avaliada usando um dos vários algoritmos possíveis
 - Da mesma forma, uma expressão da álgebra relacional pode ser avaliada de muitas maneiras
- Expressão anotada especificando estratégia de avaliação detalhada é chamada de plano de avaliação
 - pode usar um índice sobre saldo para encontrar contas com $saldo < 2500$
 - ou pode realizar varredura completa da relação e descartar contas que não tenham $saldo < 2500$

Processo de Otimização

- Otimização da consulta
 - Entre todos os planos de avaliação equivalentes, escolha aquele com o menor custo
 - O custo é estimado usando informações estatísticas do catálogo de banco de dados
 - Exemplo: número de tuplas em cada relação, tamanho das tuplas etc.
- Objetivo de “Processamento de consultas”
 - Medir custos da consulta
 - Avaliar as operações da álgebra relacional
 - Combinar algoritmos para operações individuais a fim de avaliar uma expressão completa
- No módulo de “Otimização de consultas”
 - Estuda-se como otimizar consultas, ou seja, como encontrar um plano de avaliação com o menor custo estimado

Medidas de custo da consulta

- O custo geralmente é medido como tempo total gasto para responder a consulta
 - Muitos fatores contribuem para o custo de tempo
 - acessos ao disco, CPU, ou mesmo comunicação da rede
- Normalmente, o acesso ao disco é o custo predominante e relativamente fácil de estimar.
 - Leva-se em conta:
 - Número-de-buscas * custo-médio-de-busca
 - Número-de-blocos-lidos * custo-médio-de-leitura-de-bloco
 - Número-de-blocos-escritos * custo-médio-de-escrita-de-bloco
 - Custo para escrever um bloco é maior do que para ler um bloco
- Para simplificar, usamos apenas número de transferências de bloco do disco como medida de custo
 - Pode-se ignorar (para simplificar) a diferença no custo entre E/S sequencial e aleatória
 - Pode-se ignorar (para simplificar) os custos de CPU

Medidas de custo da consulta – influência da memória

- Os custos dependem do tamanho do buffer na memória principal
 - Ter mais memória reduz a necessidade de acesso ao disco
 - A quantidade de memória real disponível para o buffer depende de outros processos concorrente do SO, e é difícil de determinar antes da execução real
 - Normalmente usamos estimativas do pior caso, supondo que apenas a quantidade mínima de memória necessária para a operação esteja disponível

Dominando as operações da álgebra

- Seleção
 - Varredura de arquivo
 - Uso de índice primário
 - Uso de índice secundário
 - Uso combinado de índice
- Ordenação
 - Quick Sort
 - Merge Sort
- Junção
 - Loops Aninhados
 - Hash Join
 - Merge Join

Seleção por varredura de arquivo

- Usado para qualquer predicado de consulta
- Algoritmo A1 (busca linear)
 - Varra cada bloco de arquivo e teste todos os registros para ver se satisfazem a condição de seleção
 - Estimativa de custo (número de blocos de disco varridos) = b_R
 - b_R indica número de blocos contendo registros da relação R
 - A busca linear pode ser aplicada independentemente de:
 - condição de seleção
 - ordenação dos registros no arquivo
 - disponibilidade de índices

Seleção via índices

- Varredura de índice – algoritmos de busca que usam um índice
 - condição de seleção precisa ser sobre chave de busca do índice
- Tipos
 - Seleção com predicado simples de igualdade (=)
 - $A = v$
 - Seleção com predicado simples de comparação (<, ≤, > e ≥)
 - $A < v$
 - Seleção com predicado composto
 - $A = v_1 \wedge B \geq v_2$

Seleção com predicado simples de igualdade: uso de índice sobre chave primária ou candidata

- Algoritmo A2:
 - Apanhe um único registro que satisfaz a condição de igualdade correspondente
 - $Custo = HT_i + 1$
 - Custo de percorrer o índice + um bloco do arquivo propriamente dito
 - O índice pode ser primário ou secundário neste caso

Seleção com predicado simples de igualdade: uso de índice primário

- Algoritmo A3: Apanha vários registros
 - Os registros serão sobre blocos consecutivos
 - Custo = HT_i + número de blocos contendo registros apanhados

Seleção com predicado simples de igualdade: uso de índice secundário para atributos não-chave

- Algoritmo A4
 - Apanhe vários registros se a chave de busca não for uma chave candidata
 - Custo = HT_i + número de registros apanhados
 - Pode ser muito dispendioso!
 - Cada registro pode estar em um bloco diferente
 - Um acesso ao bloco para cada registro apanhado

Seleção com predicado simples de comparação

- Considere as seleções na forma $\sigma_{A \leq V}(R)$ ou $\sigma_{A \geq V}(R)$
 - uma varredura de arquivo linear
 - ou usando índices das seguintes maneiras:
- A5 (índice primário sobre A, comparação)
 - Para $\sigma_{A \geq V}(R)$, use índice para encontrar primeira tupla $\geq V$ e varra a relação sequencialmente a partir de lá
 - Para $\sigma_{A \leq V}(R)$, basta varrer a relação sequencialmente até a primeira tupla $> V$; não use índice
- A6 (índice secundário sobre A, comparação)
 - Para $\sigma_{A \geq V}(R)$, use índice para encontrar primeira entrada de índice $\geq v$ e varra o índice sequencialmente, a partir de lá, para encontrar ponteiros para registros
 - Para $\sigma_{A \leq V}(R)$, basta varrer páginas de folha de índice encontrando ponteiros para registros, até primeira entrada $> v$
 - Em ambos casos, consultar registros via varredura de arquivo linear pode ser menos custosa se muitos registros tiverem que ser apanhados!

Seleção com predicado composto (conjunção)

- Conjunção: $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(R)$
 - Composição de "e"s de predicados simples
- A7 (seleção conjuntiva usando um índice)
 - Selecione do θ_i com um algoritmo (A1 a A6) que resulte no menor custo para $\sigma_{\theta_i}(R)$
 - Teste outras condições na tupla, depois de buscá-la, no buffer da memória
- A8 (seleção conjuntiva usando índice composto)
 - Use o índice composto apropriado (chave múltipla) se estiver disponível
- A9 (seleção conjuntiva por interseção de identificadores)
 - Exige índices com ponteiros de registro
 - Use índice correspondente para cada condição e apanhe a inserção de todos os conjuntos obtidos de ponteiros de registro
 - Apanhe registros do arquivo
 - Se algumas condições não tiverem índices apropriados, aplique o teste na memória

Seleção com predicado composto (disjunção)

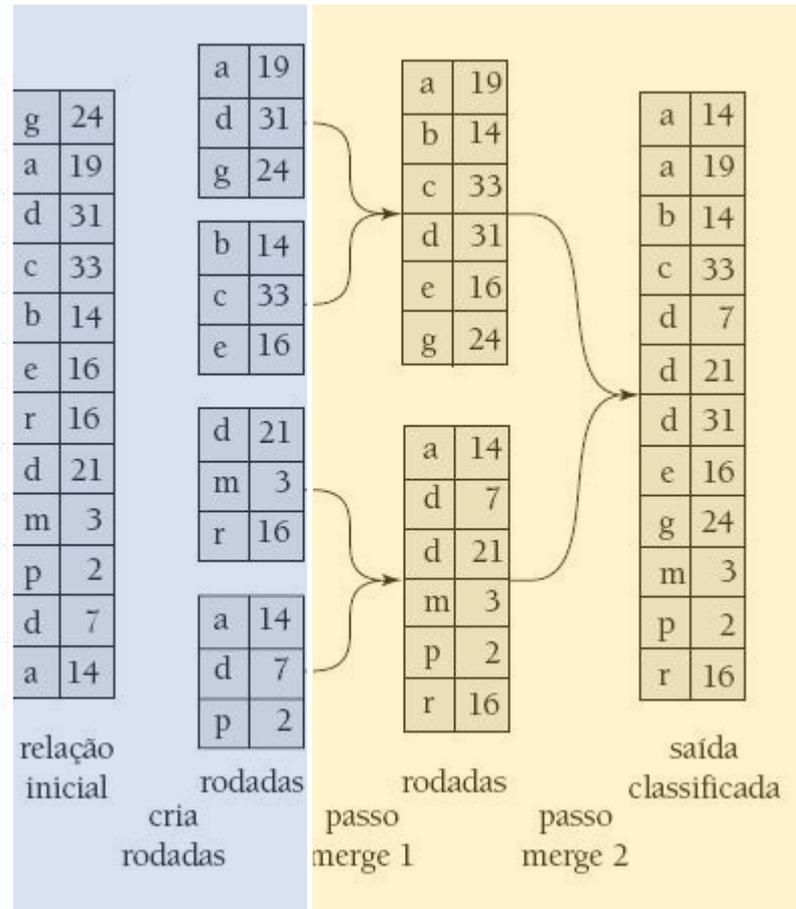
- Disjunção: $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(R)$
 - Composição de "ou"s de predicados simples
 - Só vale a pena se o número de tuplas estimado for baixo
- A10 (seleção disjuntiva pela união de identificadores)
 - Aplicável se todas as condições tiverem índices disponíveis
 - Caso contrário, use varredura linear
 - Use índice correspondente para cada condição, e apanhe a união de todos os conjuntos obtidos de ponteiros de registro
 - Depois apanhe os registros do arquivo

Ordenação

- Para relações que cabem na memória
 - técnicas como quicksort podem ser usadas
- Para relações que não cabem na memória
 - O merge-sort é uma boa escolha

Merge Sort – Princípio Geral

1. Fase de ordenação



2. Fase de mesclagem ordenada

Sort-merge

- Crie rodadas de M blocos ordenados
- Faça o seguinte repetidamente até o final da relação
 - Leia M blocos da relação para a memória
 - Classifique os M blocos
 - Escreva M blocos classificados
- Mescle as rodadas (mesclagem de $M - 1$ vias)
 - Em cada passada, grupos contíguos de $M - 1$ rodadas são mesclados
 - Uma passada reduz o número de grupos por um fator de $M - 1$ e cria grupos maiores pelo mesmo fator
 - Se $M = 11$ e existem 90 grupos, uma passada reduz o número de grupos para 9, cada grupo 10 vezes o tamanho do grupo iniciais
 - Passadas repetidas são realizadas até que todos os grupos tenham sido mescladas em um único grupo
 - Note que no slide anterior, $M = 3$
 - 3 blocos por vez na ordenação
 - 2 blocos como vias de merge para produção de um bloco ordenado

Complexidade do Merge-Sort

- Cada passagem pela relação envolve a leitura e a escrita da relação toda: $2b_R$
- O número passos de mesclagem exigidas: $\left\lceil \log_{M-1} \left(\frac{b_R}{M} \right) \right\rceil$
- O número total de acessos ao disco para a classificação externa:
 $2b_R \left[1 + \log_{M-1} \left(\frac{b_R}{M} \right) \right]$
 - 1 refere-se a fase de ordenação
 - $\log_{M-1} \left(\frac{b_R}{M} \right)$ refere-se ao número de fases de mesclagem

Operação de junção

- Vários algoritmos diferentes para implementar junções
 - Junção de loop aninhado
 - Junção de merge
 - Junção de hash
- Escolha baseada em estimativa de custo
- Notação:
 - b_R = número de blocos de R
 - n_R = número de tuplas em R
- Os exemplos usam a seguinte informação:
 - Número de registros de cliente: 10.000 depositante: 5.000
 - Número de blocos de cliente: 400 depositante: 100

Junção de loop aninhado em bloco

- A variante da junção de loop aninhado em que cada bloco da relação interna é emparelhado com cada bloco da relação externa
 - $RS \leftarrow \emptyset$
 - for each bloco x of R
 - for each bloco y of S
 - for each tupla t in x
 - for each tupla u in y
 - if $match(t, u)$
 - $RS \leftarrow RS \cup \{tu\}$
- O match cuida do predicado de ligação das tabelas
 - (cláusula de junção)

Junção de loop aninhado em bloco: pior caso

- Cenário de pior caso
 - Memória para ler três blocos por vez
 - Um para cada relação R e S
 - Um para o RS
- Estimativa no pior caso: $b_R \times b_S + b_R$ acessos ao bloco
 - Cada bloco na relação interna S é lido uma vez para cada bloco na relação externa R

Junção de loop aninhado em bloco: melhor caso

- Cenário de melhor caso
 - Memória para ler M blocos por vez
 - $b_S < M$
 - Um para o RS
- Melhor caso: $b_R + b_S$ acessos ao disco
 - Note que para dar certo, a relação interna é que tem que caber na memória

Junção de loop aninhado em bloco: caso geral

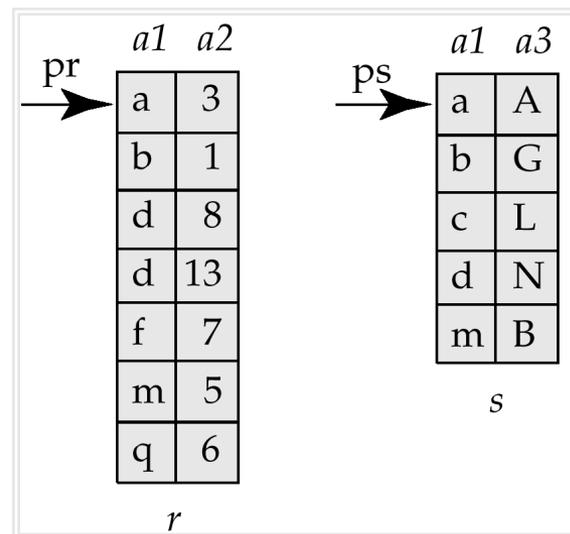
- Cenário geral
 - Memória para ler M blocos por vez
 - Ler $M - 2$ blocos de S (relação interna) por vez
 - Um para R (relação externa) e outro para o RS
 - $Custo = b_R \times \left\lceil \frac{b_S}{M-2} \right\rceil + b_R$
 - A relação interna deve ser a maior neste caso

Exemplo de custos de junção de loop aninhado

- Calcule depositante \bowtie cliente, com depositante como relação externa
- Número de registros de cliente: 10.000 depositante: 5000
- Número de blocos de cliente: 400 depositante: 100
- Calcule depositante \bowtie cliente no pior caso?
- Calcule depositante \bowtie cliente no melhor caso?
- Calcule depositante \bowtie cliente no caso geral?
- Indique, para cada um dos casos, a relação interna e externa

Junção de merge

- Ordene as duas relações pelos seus atributos de junção
 - Caso já não estejam ordenadas por estes atributos
- Mescle as relações ordenadas para juntá-las
 - A etapa de junção é semelhante ao estágio de mesclagem do algoritmo de merge-sort
 - A principal diferença é o tratamento de valores duplicados no atributo de junção - cada par com o mesmo valor no atributo de junção precisa ser combinado



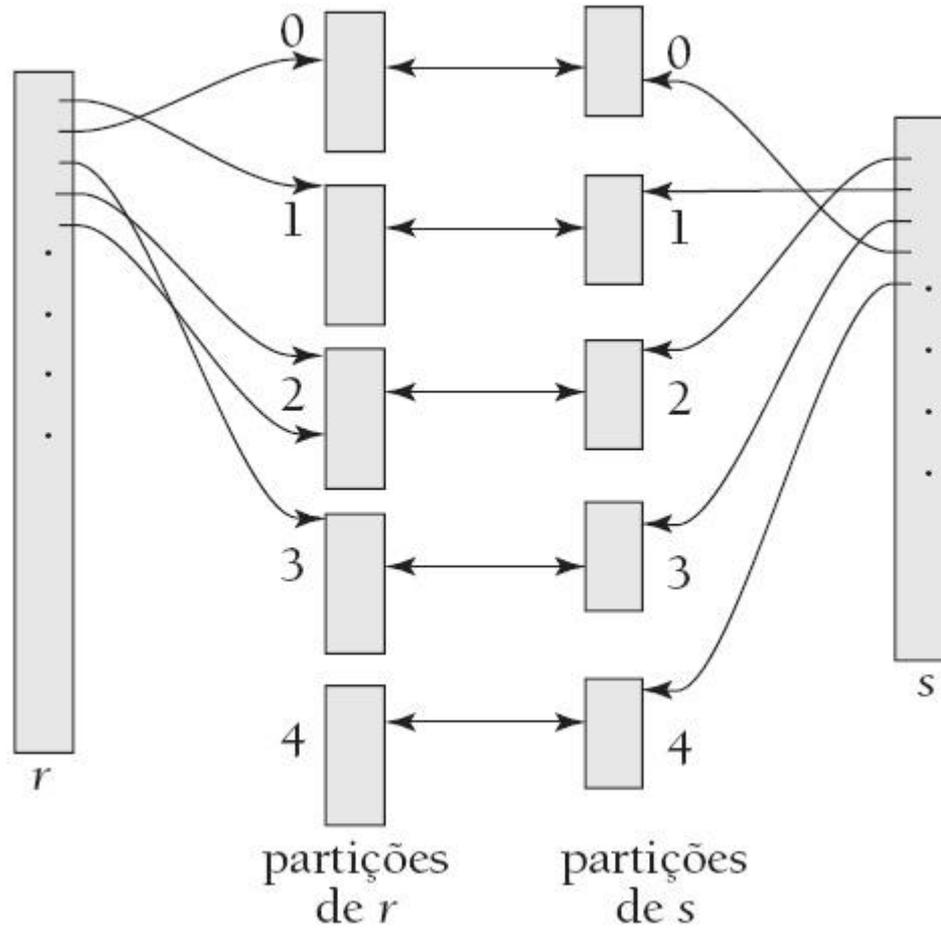
Junção merge (cont.)

- Só pode ser usada para junções de igualdade e naturais
- Cada bloco precisa ser lido uma vez
 - Supondo que todas as tuplas para determinado valor dos atributos de junção caibam na memória
- O número de acessos a bloco para a junção merge é $b_R + b_S$ + custo de ordenar R e S (se não estiverem)

Junção de hash

- Aplicável para equi-junções e junções naturais
- Uma função de hash h é usada para particionar tuplas das duas relações
- h mapeia valores de $JoinAttrs$ para $\{0, 1, \dots, n\}$, onde $JoinAttrs$ indica os atributos comuns de R e S usados na junção
 - hr_0, hr_1, \dots, hr_n indica partições de r tuplas
 - Cada tupla $tr \in R$ é colocada na partição hr_i onde $i = h(tr, JoinAttrs)$
 - hs_0, hs_1, \dots, hs_n indica partições de s tuplas
 - Cada tupla $ts \in S$ é colocada na partição hs_i , onde $i = h(ts, JoinAttrs)$
- No caso geral, o valor de n depende do número M de blocos que podem ficar na memória
 - Quando não o número de blocos não couber na memória, faz-se um particionamento recursivo

Princípio da junção de hash



Comparações via junção de hash

- As tuplas da relação R em hr_i só precisam ser comparadas com as tuplas da relação S em hs_i
- Elas não precisam ser comparadas com as tuplas s em qualquer outra partição, pois:
 - uma tupla r e uma tupla s que satisfazem a condição de junção tem o mesmo valor para os atributos de junção
 - Se esse valor for transformado por hash em algum valor, a tupla r precisa estar em hr_i e a tupla s em hs_i

Custo da junção de hash

- Se o particionamento recursivo não for exigido:
 - custo da junção é aproximadamente $3(b_R + b_S)$
 - 2 varreduras para o particionamento e 1 para a junção
- Se o particionamento recursivo for exigido:
 - Custo da junção é $2(b_R + b_S) [\log_{M-1}(b_S) - 1] + b_R + b_S$
 - 2 varreduras por nível de particionamento
 - Número de nível de particionamento é definido por $\log_{M-1}(b_S) - 1$

Exemplo de custo de merge join e hash join

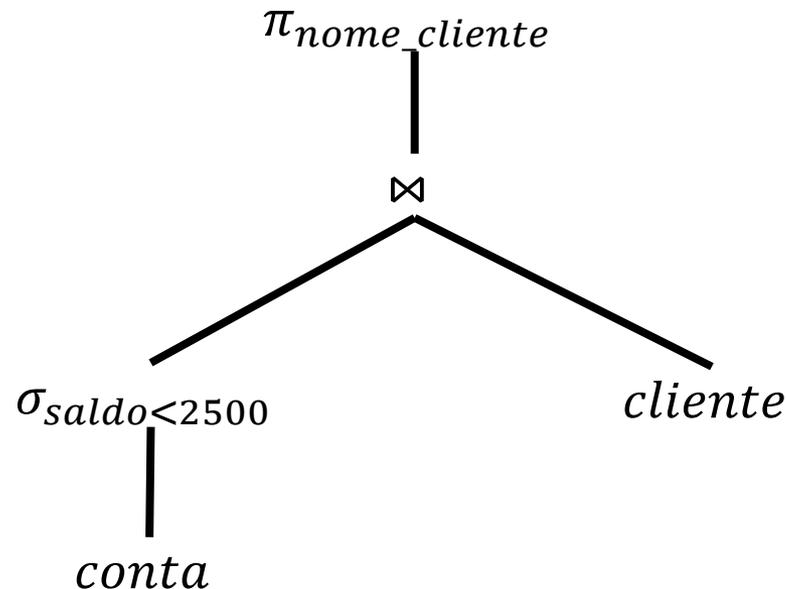
- Calcule a junção depositante \bowtie cliente, calcule os custos do merge join e hash-join, considerando o caso mais geral
 - Considere que o tamanho da memória seja de 20 blocos, i.e., $M = 20$
 - $b_{depositante} = 100$ e $b_{cliente} = 400$
- Merge Join
 - $Sort_{depositante} = 2b_{depositante} \left[1 + \log_{20-1} \left(\frac{b_{depositante}}{20} \right) \right]$
 - $Sort_{cliente} = 2b_{cliente} \left[1 + \log_{20-1} \left(\frac{b_{cliente}}{20} \right) \right]$
 - Total: $b_{depositante} + b_{cliente} + Sort_{depositante} + Sort_{cliente}$
 - $100 + 400 + 200[1 + \log_{19}(5)] + 800[1 + \log_{19}(20)]$
 - $\cong 3300$
 - Se cliente já estiver ordenado, $\cong 900$
- Hash Join
 - $2(b_R + b_S) [\log_{M-1}(b_S) - 1] + b_R + b_S$
 - $2(100 + 400) [\log_{19}(100) - 1] + 100 + 400$
 - $\cong 1500$

Avaliação de expressões

- Uma consulta é um conjunto de expressões
 - Expressões são composições de operações individuais
- As alternativas para avaliação de uma árvore de expressão inteira
 - *Materialização*: gera resultados de uma expressão cujas entradas são relações, originais ou já são calculadas, e materialize (armazene) isso em disco.
 - *Pipelining*: passe adiante as tuplas para operações ancestrais mesmo quando uma operação está sendo executada

Materialização

- Avaliação materializada: avalie uma operação de cada vez, começando no nível mais baixo
 - Use resultados intermediários materializados em relações temporárias para avaliar as operações do nível ancestral
 - Por exemplo, na figura a seguir, calcule e armazene $\sigma_{\text{saldo} < 2500}(\text{conta})$
 - Depois calcule o armazenamento de sua junção com cliente
 - Finalmente, calcule as projeções sobre nome-cliente



Aspectos da Materialização

- A avaliação materializada sempre é aplicável
- O custo da escrita de resultados em disco e sua leitura de volta pode ser muito alto
 - Custo geral = Soma dos custos de operações individuais + custo da escrita de resultados intermediários em disco
- Buffer duplo: utiliza dois buffers de saída para cada operação, quando uma é a escrita inteira para o disco enquanto a outra está sendo preenchida
 - Permite a sobreposição de escritas de disco com cálculo e reduz o tempo de execução

Pipelining

- Avaliação em pipeline: avalia várias operações simultaneamente, passando os resultados de uma operação para a seguinte
- Por exemplo, na árvore de expressão anterior, não armazene o resultado da junção entre o saldo filtrado e a conta
 - em vez disso passe tuplas diretamente para a projeção
- O custo computacional é menor quando comparado a materialização: não precisa armazenar uma relação temporária em disco
- O *pipelining* nem sempre pode ser possível, por exemplo, ordenação, junção de hash
- Para que o *pipelining* seja eficiente, use algoritmos de avaliação que gerem tuplas de saída enquanto as tuplas são recebidas para entradas da operação
- As pipelines podem ser executadas de duas maneiras: controladas por demanda e controladas por produtor

Pipelining por demanda

- O sistema repetidamente solicita a próxima tupla a operação do fim do pipeline
 - Cada operação solicita a próxima tupla de operações filhas, conforme a necessidade, a fim de gerar sua próxima tupla
 - Entre as chamadas, a operação precisa manter o “estado”, para que saiba o que retornar em seguida
 - Cada operação é implementada como um iterador implementando as seguintes operações
 - `open()`
 - inicializar varredura de arquivo e ponteiro para início do arquivo como estado
 - Por exemplo: junção de merge: classificar relações e armazenar ponteiros para o início das relações armazenadas como estado
 - `next()`
 - gerar próxima tupla, e avançar e armazenar ponteiro de arquivo
 - Por exemplo: para junção merge: continuar com merge a partir do estado anterior até a próxima tupla de saída ser encontrada. Salvar ponteiros como estado do iterador
 - `close()`

Pipelining por produção

- Os operadores produzem tuplas rapidamente e as passam para 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 haja espaço no buffer e depois gera mais tuplas
- O sistema escalona operações que têm espaço no buffer de saída e podem processar mais tuplas de entrada

Algoritmos de avaliação para pipelining

- Alguns algoritmos não são capazes de gerar resultados mesmo quando recebem tuplas de entrada
 - Por exemplo: junção merge, ou junção de hash
 - Estes resultam em resultados intermediários sendo gravados em disco e depois lidos de volta sempre
- Variantes de algoritmo são possíveis para gerar (pelo menos alguns) resultados no ato, enquanto tuplas de entrada são lidas
 - Por exemplo: a junção de hash híbrida gera tuplas mesmo quando tuplas de relação de sonda da partição na memória (partição 0) são lidas
 - Técnica de junção em pipeline: Junção de hash híbrida, modificada para colocar em buffer tanto as tuplas da partição 0 de relações na memória, lendo-as quando estiverem disponíveis, quanto resultados de saída de quaisquer combinações entre as tuplas da partição 0
 - Quando uma nova tupla r_0 é encontrada, combine-a com tuplas s_0 existentes, gere combinações de saída e salve-a em r_0
 - Simetricamente para tuplas s_0

Referências

