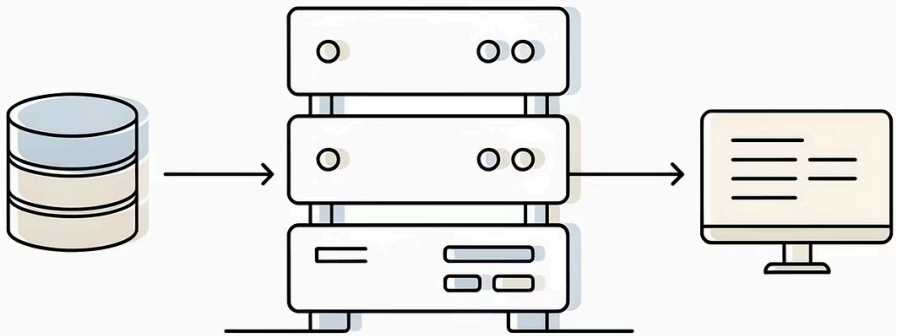


CEFET/RJ

Processamento de Consulta em Bancos de Dados

O processamento de consultas é um dos componentes fundamentais de um Sistema Gerenciador de Banco de Dados (SGBD). Este módulo explora como as consultas são analisadas, otimizadas e executadas para retornar resultados eficientes aos usuários.

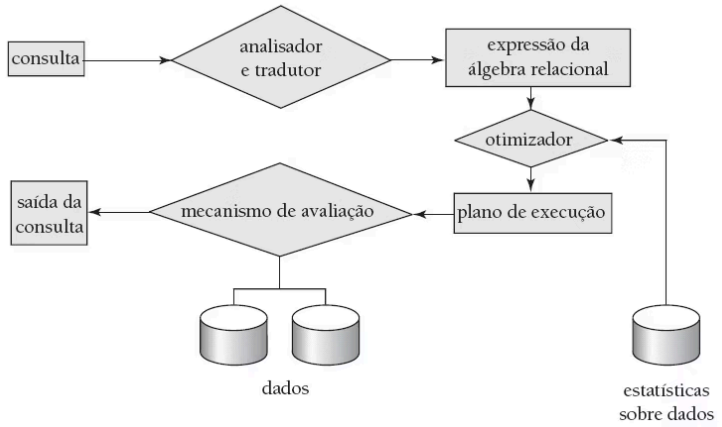
Eduardo Ogasawara

eduardo.ogasawara@cefet-rj.br
https://eic.cefet-rj.br/~eogasawara

FUNDAMENTOS

Etapas Básicas no Processamento da Consulta

O processamento de uma consulta em um banco de dados relacional passa por três etapas fundamentais que transformam uma requisição SQL em resultados concretos. Cada etapa desempenha um papel crucial na eficiência e correção do processo.



01

Análise e Tradução

Converte a consulta SQL em uma representação interna e verifica sua validade sintática e semântica

02

Otimização

Identifica o plano de execução mais eficiente entre múltiplas alternativas equivalentes

03

Avaliação

Executa o plano escolhido e retorna os resultados ao usuário

DETALHAMENTO

Análise, Tradução e Avaliação

Análise e Tradução

Nesta primeira etapa, o sistema realiza uma série de transformações e verificações essenciais:

- A consulta SQL é traduzida para um formato interno do SGBD
- Essa representação interna é convertida em expressões de álgebra relacional
- O analisador sintático verifica se a consulta está gramaticalmente correta
- O analisador semântico valida se as relações e atributos referenciados existem no esquema

Avaliação

A fase final executa o plano selecionado e produz os resultados:

- O mecanismo de execução de consulta recebe o plano de avaliação otimizado
- Executa cada operação do plano na ordem especificada
- Acessa os dados necessários em disco ou na memória
- Retorna as tuplas resultantes à aplicação ou usuário

OTIMIZAÇÃO

Etapas Básicas no Processamento da Consulta: Otimização

A otimização de consultas é o coração do processamento eficiente em SGBDs. Esta etapa crítica determina como uma consulta será executada, impactando diretamente no tempo de resposta e no uso de recursos do sistema.

O otimizador analisa múltiplas estratégias de execução, cada uma com diferentes ordenações de operações e métodos de acesso aos dados. Utilizando estatísticas armazenadas no catálogo do banco de dados, ele estima o custo de cada alternativa e seleciona aquela que promete o melhor desempenho.

Árvore de Expressão como Plano de Avaliação

A representação interna de uma consulta como árvore de expressão é fundamental para o processamento eficiente. Esta estrutura hierárquica permite visualizar e manipular as operações da álgebra relacional de forma sistemática.

Estrutura da Árvore

Cada nó interno representa uma operação da álgebra relacional (seleção, projeção, junção), enquanto as folhas representam as relações base do banco de dados

Planos Equivalentes

Múltiplas árvores diferentes podem representar a mesma consulta lógica, produzindo resultados idênticos mas com custos de execução distintos

Escolha Otimizada

O otimizador avalia as alternativas e seleciona a árvore com menor custo estimado, considerando estatísticas e heurísticas de otimização

ESTRATÉGIA

Processo de Otimização de Consultas

Análise de Alternativas

Entre todos os planos de avaliação equivalentes, o otimizador identifica e compara múltiplas estratégias de execução

Estimativa de Custo

Utiliza informações estatísticas do catálogo (número de tuplas, tamanho de registros, distribuição de valores) para calcular o custo esperado

Seleção do Melhor

Escolhe o plano com menor custo estimado para execução real

Objetivos do Processamento de Consultas

- Desenvolver métricas precisas para medir custos de diferentes estratégias de execução
- Implementar algoritmos eficientes para avaliar operações da álgebra relacional
- Combinar operações individuais em planos de execução completos e otimizados

No módulo de otimização de consultas, o foco está em encontrar o plano de avaliação com menor custo estimado, aplicando técnicas como transformações algébricas, ordenação de operações e seleção de métodos de acesso.

CUSTO

Medidas de Custo da Consulta

O custo de execução de uma consulta é tipicamente medido pelo tempo total necessário para produzir os resultados. Diversos fatores contribuem para este custo, mas o acesso ao disco geralmente domina a métrica.

Componentes do Custo de E/S em Disco

1 Operações de Busca

Número de buscas × custo médio de busca — O tempo para posicionar o cabeçote de leitura na posição correta

2 Leitura de Blocos

Número de blocos lidos × custo médio de leitura — A transferência de dados do disco para a memória

3 Escrita de Blocos

Número de blocos escritos × custo médio de escrita — Operações de escrita são mais custosas que leituras devido a verificações adicionais

Simplificações na Modelagem

Para facilitar a análise, frequentemente utilizamos o número de transferências de bloco como medida única de custo.

Podemos ignorar:

- Diferenças entre E/S sequencial e aleatória
- Custos de processamento da CPU
- Custos de comunicação em rede

Influência da Memória nas Medidas de Custo

A quantidade de memória disponível para buffers é um fator crítico que afeta dramaticamente o custo real de execução das consultas. Um buffer maior reduz significativamente a necessidade de acessos ao disco, melhorando o desempenho.

Desafio da Previsão

A quantidade real de memória disponível é difícil de determinar antecipadamente, pois depende de processos concorrentes do sistema operacional e de outras operações do SGBD em execução simultânea.

Estratégia Conservadora

Tipicamente, as estimativas de custo assumem o **pior caso possível**, supondo que apenas a quantidade mínima de memória necessária para executar a operação estará disponível.

Impacto no Desempenho

Ter mais memória disponível pode transformar operações que exigiriam múltiplas passadas pelo disco em operações realizadas completamente em memória, reduzindo drasticamente o tempo de execução.

OPERAÇÕES FUNDAMENTAIS

Dominando as Operações da Álgebra Relacional

Para processar consultas eficientemente, é essencial compreender os algoritmos disponíveis para cada operação fundamental da álgebra relacional. Cada operação pode ser implementada de múltiplas formas, cada uma adequada a diferentes cenários.

Seleção

Filtra tuplas que satisfazem um predicado. Implementações incluem varredura completa de arquivo, uso de índice primário, secundário ou combinação de índices.

Ordenação

Organiza tuplas segundo um ou mais atributos. Algoritmos principais incluem Quick Sort para dados em memória e Merge Sort externo para volumes grandes.

Junção

Combina tuplas de duas relações. Técnicas incluem Loops Aninhados (simples mas custoso), Hash Join (eficiente para grandes volumes) e Merge Join (ótimo para dados ordenados).

Avaliação da Operação de Projeção

A operação de projeção elimina atributos de uma relação, mantendo apenas as colunas especificadas. Embora conceitualmente simples, sua implementação eficiente apresenta desafios importantes.

O principal desafio é a **eliminação de duplicatas**. Quando projetamos sobre atributos que não formam uma chave, tuplas idênticas podem surgir e precisam ser removidas para manter a semântica de conjuntos da álgebra relacional.

Projeção Baseada em Ordenação

Ordena as tuplas projetadas e então remove duplicatas consecutivas em uma única passada. Eficiente quando a ordenação pode ser feita em memória ou com poucas passadas.

Projeção Baseada em Hashing

Utiliza uma função hash para particionar as tuplas e eliminar duplicatas dentro de cada partição. Adequado quando há memória suficiente para as estruturas de hash.

O custo da operação depende criticamente do tamanho da relação, da necessidade real de eliminar duplicatas (que pode ser evitada se projetarmos sobre uma chave) e da memória disponível para buffers.

VARREDURA COMPLETA

Seleção por Varredura de Arquivo

Varredura Linear Completa

O método mais simples: examina cada bloco do arquivo sequencialmente, testando o predicado de seleção em cada tupla

Custo de E/S

Requer br transferências de bloco, onde br é o número de blocos que a relação ocupa no disco

Quando Usar

Adequado para predicados de baixa seletividade ou quando não há índices disponíveis sobre os atributos do predicado

A varredura de arquivo é o método de acesso mais básico e funciona para qualquer predicado de seleção. Apesar de sua simplicidade, pode ser bastante eficiente quando:

- O predicado tem **baixa seletividade** (seleciona grande porcentagem das tuplas)
- Os dados estão armazenados de forma **sequencial** no disco
- O sistema operacional pode realizar **leitura antecipada** (read-ahead)

Vantagem: Não requer estruturas auxiliares ou índices

Desvantagem: Sempre examina toda a relação, mesmo para predicados altamente seletivos

ACESSO INDEXADO

Seleção via Índices

Índices são estruturas de dados auxiliares que permitem acesso rápido a tuplas específicas sem examinar toda a relação. Existem diversos tipos de índices, cada um otimizado para diferentes padrões de acesso.

Índice Primário

Construído sobre a chave de ordenação do arquivo. Permite busca binária eficiente e acesso sequencial rápido a registros relacionados.

Índice Secundário

Construído sobre atributos não-chave. Pode apontar para múltiplas tuplas e geralmente é mais custoso que índices primários para acessos múltiplos.

Índice Hash

Utiliza função hash para localizar rapidamente tuplas. Excelente para buscas por igualdade, mas não suporta consultas de intervalo.

Escolha entre Varredura e Uso de Índices

A decisão de usar um índice ou varredura sequencial não é trivial. O otimizador deve considerar múltiplos fatores para fazer a escolha correta.

Seletividade do Predicado

Quantas tuplas satisfazem a condição? Predicados altamente seletivos favorecem índices.

Tipo de Índice Disponível

Índice primário? Secundário? Hash? Cada tipo tem custos diferentes.

Custo Estimado de E/S

Comparação entre o custo de usar o índice versus varrer o arquivo inteiro.

Percepção Importante

Índices **nem sempre são a melhor opção**. Para predicados com baixa seletividade (que retornam muitas tuplas), uma varredura sequencial pode ser mais eficiente do que múltiplos acessos aleatórios via índice.

CHAVE ÚNICA

Seleção com Igualdade: Índice sobre Chave

Quando o predicado de seleção especifica igualdade sobre uma chave primária ou candidata, sabemos que no máximo uma tupla satisfará a condição. Essa garantia de unicidade permite otimizações significativas.

1

Busca no Índice

O otimizador utiliza o índice (tipicamente uma árvore B+ ou estrutura hash) para localizar rapidamente a entrada correspondente ao valor buscado

2

Acesso Direto

Uma vez localizada a entrada no índice, um único acesso ao arquivo de dados recupera a tupla desejada

3

Retorno do Resultado

A tupla é retornada ao usuário. Não são necessárias verificações adicionais devido à restrição de unicidade

Custo de E/S

Para uma árvore B+ de altura h:

- **h + 1** transferências de bloco no total
- h acessos para navegar no índice
- 1 acesso para recuperar a tupla

Eficiência

Este é um dos cenários mais eficientes de busca em bancos de dados. Com índices bem dimensionados, h é tipicamente 3-4 mesmo para tabelas com milhões de registros.

Seleção com Igualdade: Índice Primário

Quando temos um predicado de igualdade sobre atributos que não formam uma chave, múltiplas tuplas podem satisfazer a condição. Um índice primário (construído sobre a chave de ordenação do arquivo) oferece vantagens especiais neste cenário.

Localização Inicial

O índice primário é usado para encontrar o primeiro registro que satisfaz o predicado de igualdade

Leitura Sequencial

Como o arquivo está ordenado pela chave de indexação, todos os registros que satisfazem a condição estão fisicamente adjacentes

Acesso Eficiente

Registros adicionais são lidos sequencialmente até que a condição não seja mais satisfeita

Custo: h + b acessos a blocos, onde:

- h = altura da árvore do índice
- b = número de blocos contendo registros que satisfazem o predicado

A leitura sequencial é muito mais eficiente que acessos aleatórios, tornando índices primários ideais para predicados com múltiplos resultados.

Vantagem chave: Localidade física dos dados relacionados minimiza movimentos do cabeçote do disco

ATRIBUTO NÃO-CHAVE

Seleção com Igualdade: Índice Secundário

Índices secundários são construídos sobre atributos que não são a chave de ordenação do arquivo. Diferentemente de índices primários, os registros correspondentes não estão fisicamente agrupados.

Busca no Índice

Navegação na estrutura do índice secundário (h acessos)

Lista de Ponteiros

Recuperação da lista com n ponteiros para registros

Acessos Aleatórios

Até n acessos ao disco para recuperar cada registro individualmente

---

Custo: Pior Caso

h + n transferências de bloco

Onde n é o número de registros que satisfazem o predicado. Cada registro pode estar em um bloco diferente.

Custo: Caso Otimista

Se alguns registros compartilham blocos, o custo pode ser menor. Porém, essa otimização é difícil de prever.

Atenção: Para predicados com baixa seletividade, um índice secundário pode ser **menos eficiente** que uma varredura completa devido aos acessos aleatórios.

Seleção com Predicado de Comparação

Predicados de comparação (maior que, menor que, entre outros) requerem estratégias diferentes dos predicados de igualdade. A eficiência do acesso depende fortemente da disponibilidade e tipo de índice.

Com Índice Ordenado

Árvores B+ permitem navegação eficiente. Para $A \geq v$, localiza-se v e percorre-se para frente; para $A \leq v$, percorre-se desde o início até v

Sem Índice Adequado

Varredura completa do arquivo é necessária, examinando cada tupla para verificar se satisfaz a condição de comparação

Custos por Tipo de Acesso

Índice Primário ($\geq$ ou $\leq$)

Custo: h + b blocos, onde b é o número de blocos contendo resultados. Leitura sequencial após localização inicial.

Índice Secundário ($\geq$ ou $\leq$)

Custo: h + n blocos (pior caso), onde n é o número de tuplas. Acessos potencialmente aleatórios a cada registro.

Varredura de Arquivo

Custo: br blocos, onde br é o tamanho total da relação. Exame de todas as tuplas.

CONJUNÇÃO

Seleção com Predicado Composto: Conjunção

Quando temos múltiplas condições conectadas por AND ($\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n$), todas devem ser simultaneamente verdadeiras. O otimizador pode explorar diferentes estratégias dependendo dos índices disponíveis.

Estratégia 1: Índice Único

Selecione um índice sobre um dos atributos que prometa maior seletividade. Use-o para recuperar tuplas candidatas e então avalie as demais condições em memória.

Estratégia 2: Índice Composto

Se existe um índice composto sobre múltiplos atributos do predicado, use-o para filtrar simultaneamente por várias condições.

Estratégia 3: Interseção de Índices

Use múltiplos índices separados, recupere os conjuntos de identificadores de registro de cada um, compute a interseção e então acesse apenas os registros na interseção.

Escolha da Estratégia

A estratégia 3 (interseção) é vantajosa quando cada condição individual tem baixa seletividade, mas a conjunção é altamente seletiva. A estratégia 1 é preferível quando uma condição já é altamente seletiva.

DISJUNÇÃO

Seleção com Predicado Composto: Disjunção

Predicados com disjunção ($\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n$) são mais desafiadores: basta que uma condição seja verdadeira. Isso limita as oportunidades de otimização em comparação com conjunções.

Com Índices em Todas as Condições

Se há índices disponíveis para todos os atributos mencionados na disjunção:

- Use cada índice para recuperar conjuntos de identificadores
- Compute a **união** dos conjuntos de identificadores
- Acesse os registros correspondentes aos identificadores na união
- Elimine duplicatas se necessário

Sem Índices Completos

Se falta índice para qualquer uma das condições na disjunção:

- Necessário realizar **varredura completa** do arquivo
- Avaliar o predicado completo para cada tupla
- Não é possível otimizar parcialmente

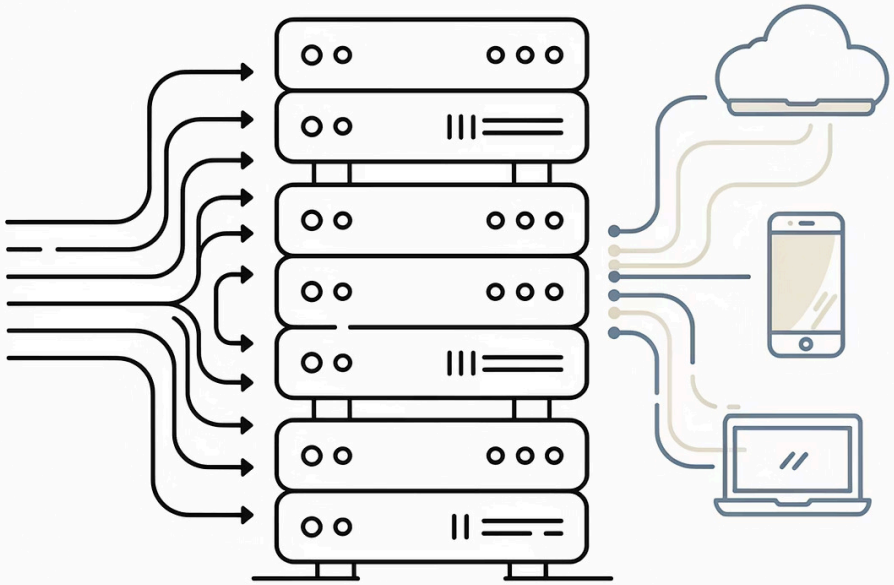
Razão: Mesmo que alguns atributos tenham índice, não podemos garantir que capturamos todas as tuplas sem examinar o arquivo inteiro.

Custo com Índices Completos

Soma dos custos de uso de cada índice individual, mais o custo de unir os resultados e eliminar duplicatas.

Custo sem Índices Completos

br transferências de bloco para varrer toda a relação, independente de quantos índices parciais existam.



Ordenação e Avaliação de Consultas em Bancos de Dados

Uma exploração detalhada das técnicas de ordenação e operações de junção para processamento eficiente de consultas em sistemas de gerenciamento de bancos de dados relacionais.

FUNDAMENTOS

Estratégias de Ordenação para Relações

Relações que cabem na memória

Quando o conjunto completo de dados pode ser carregado inteiramente na memória RAM, algoritmos de ordenação in-memory são altamente eficientes. O **quicksort** é uma escolha excelente nestes casos, oferecendo complexidade média de O(n log n) e aproveitando a velocidade de acesso à memória principal.

- Acesso rápido aos dados
- Sem overhead de I/O
- Ideal para datasets pequenos e médios

Relações que não cabem na memória

Para grandes volumes de dados que excedem a capacidade da memória, técnicas de ordenação externa são necessárias. O **merge-sort externo** é particularmente adequado, pois minimiza operações de I/O de disco, que são o principal gargalo de performance nestas situações.

- Operações sequenciais em disco
- Divisão em partições gerenciáveis
- Processamento em múltiplas fases

ALGORITMO

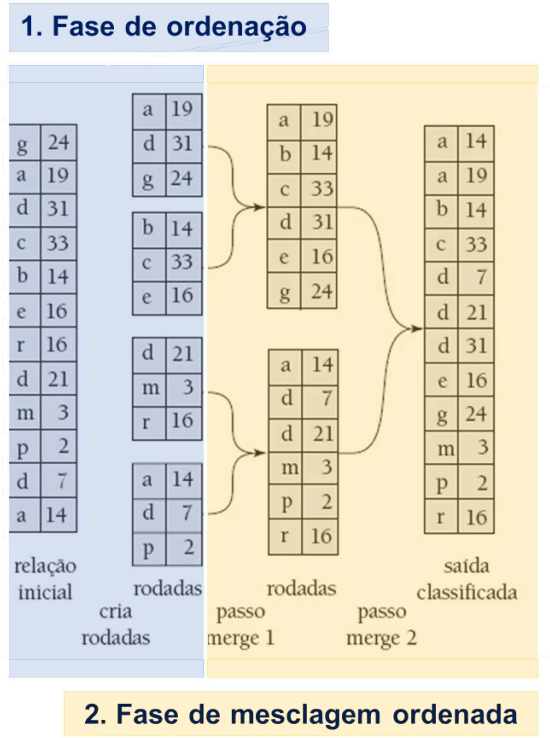
Sort-Merge: Princípio Geral

Fase 1: Ordenação Inicial

A primeira fase do merge-sort externo envolve a criação de runs ordenados. O algoritmo lê blocos de dados que cabem na memória, ordena-os internamente usando quicksort ou outro algoritmo eficiente, e grava os resultados como runs temporários no disco.

Fase 2: Mesclagem Ordenada

Na segunda fase, os runs ordenados são mesclados iterativamente. Em cada passada, múltiplos runs são combinados em runs maiores, mantendo a ordem global. Este processo continua até que reste apenas um único run ordenado contendo todos os dados.



O diagrama ilustra o fluxo de dados através das duas fases principais: particionamento e ordenação local seguidos por mesclagem progressiva até alcançar a ordenação completa.

Sort-Merge: Abordagem Integrada

O algoritmo **sort-merge** combina as fases de ordenação e mesclagem em um único processo unificado, otimizado para operações em grandes volumes de dados armazenados em disco.

01

Particionamento Inicial

Divisão dos dados em partições que cabem na memória disponível

02

Ordenação Local

Aplicação de algoritmo de ordenação rápida em cada partição

03

Mesclagem Multi-Way

Combinação inteligente de múltiplas partições ordenadas simultaneamente

04

Geração do Resultado

Produção do conjunto de dados completamente ordenado

Esta técnica é fundamental para operações de junção e ordenação em sistemas de bancos de dados relacionais modernos, especialmente quando lidamos com datasets que ultrapassam a capacidade da memória RAM.

ANÁLISE

Complexidade do Merge-Sort Externo

A análise de complexidade do merge-sort externo considera principalmente o número de acessos a disco, que é o fator dominante no tempo de execução para grandes volumes de dados.

1

Fase de Ordenação

Custo: Cada bloco é lido uma vez e gravado uma vez

Operações I/O: 2 × (número de blocos)

Esta fase tem custo linear em relação ao tamanho dos dados de entrada.

2

Fase de Mesclagem

Número de passadas: $\lceil \log_{M-1}(br/M) \rceil$

Custo por passada: 2 × br

Onde M é o número de buffers disponíveis e br é o número de blocos da relação.

3

Custo Total

Fórmula: $br \times (2 \times \lceil \log_{M-1}(br/M) \rceil + 1)$

Esta complexidade logarítmica torna o merge-sort externo escalável para grandes volumes de dados.

Avaliação de Operações de Conjunto

Operações Fundamentais

As operações de conjunto são essenciais em consultas relacionais:

- **União (∪):** Combina tuplas de duas relações
- **Interseção (∩):** Retorna tuplas comuns
- **Diferença (−):** Remove tuplas de uma relação

Estratégias de Implementação

Duas abordagens principais são utilizadas:

Baseada em Ordenação

Ordena ambas as relações e processa sequencialmente

Baseada em Hashing

Usa funções hash para particionar e comparar eficientemente

Ambas as abordagens requerem pré-processamento: as relações devem estar ordenadas pelo atributo de comparação ou particionadas por uma função de hash consistente.

OPERADOR CRÍTICO

Operação de Junção

A **junção** (join) é uma das operações mais importantes e custosas em bancos de dados relacionais. Ela combina tuplas de duas relações baseando-se em uma condição de junção, tipicamente uma igualdade entre atributos.

Nested Loop Join

Abordagem iterativa simples que compara cada tupla de uma relação com todas as tuplas da outra. Eficiente para relações pequenas ou quando índices estão disponíveis.

Merge Join

Requer que ambas as relações estejam ordenadas pelo atributo de junção. Realiza uma varredura linear sincronizada, sendo muito eficiente para grandes conjuntos ordenados.

Hash Join

Usa funções hash para particionar ambas as relações em buckets correspondentes. Excelente desempenho para grandes volumes quando há memória suficiente.

A escolha do algoritmo de junção adequado pode impactar dramaticamente a desempenho de uma consulta, variando de segundos a horas de execução.

Junção de Loop Aninhado em Bloco

O algoritmo de **junção de loop aninhado em bloco** (block nested-loop join) é uma otimização que processa dados em blocos ao invés de tuplas individuais, reduzindo significativamente o número de operações de I/O.

01

Seleção da Relação Externa

Escolhe a menor relação como externa para minimizar iterações

02

Leitura de Blocos

Carrega múltiplos blocos da relação externa na memória

03

Varredura da Relação Interna

Para cada conjunto de blocos externos, varre completamente a relação interna

04

Geração do ResultSet (RS)

Produz tuplas que satisfazem a condição de junção

A eficiência deste algoritmo depende criticamente da quantidade de memória disponível para buffers. Quanto mais blocos da relação externa puderem ser mantidos em memória simultaneamente, menor será o número de varreduras necessárias da relação interna.

⚠ PIOR CASO

Junção de Loop Aninhado em Bloco: Pior Caso

O pior cenário ocorre quando a memória disponível é extremamente limitada, permitindo apenas um bloco de cada relação mais um bloco de saída.

3

Blocos Mínimos

Apenas 3 blocos de memória disponíveis

$b_1 \times b_2$

Acessos Totais

Cada bloco externo requer varredura completa da interna

Fórmula de Custo no Pior Caso:

$$Custo = b_r + (b_r \times b_s)$$

Onde **br** é o número de blocos da relação externa e **bs** é o número de blocos da relação interna.

Neste cenário, para cada bloco da relação externa, toda a relação interna precisa ser lida do disco. Isso resulta em um número quadrático de operações de I/O, tornando o algoritmo impraticável para grandes relações.

Este caso ilustra a importância crítica da alocação adequada de memória para operações de junção em SGBDs.

☆ MELHOR CASO

Junção de Loop Aninhado em Bloco: Melhor Caso

O melhor cenário ocorre quando há memória suficiente para carregar completamente a menor relação, eliminando a necessidade de múltiplas varreduras.

Condição Ideal

Memória suficiente para toda a relação externa: M ≥ br + 2

Varredura Única

Cada relação é lida exatamente uma vez do disco

Performance Máxima

Custo linear minimizado em relação ao tamanho dos dados

Fórmula de Custo no Melhor Caso:

$$Custo = b_r + b_s$$

Este é o cenário ideal onde:

- Toda a relação externa cabe na memória
- Apenas uma varredura da relação interna é necessária
- O número de operações de I/O é minimizado

Implicações práticas:

Otimizadores de consulta sempre tentam alocar a menor relação como externa e maximizar o uso de memória disponível para aproximar-se deste caso ideal.

Junção de Loop Aninhado em Bloco: Caso Geral

No caso geral, a memória disponível permite carregar múltiplos blocos da relação externa, mas não toda a relação. Este é o cenário mais comum na prática.

Fórmula de Custo no Caso Geral:

$$Custo = b_r + \lceil \frac{b_r}{M-2} \rceil \times b_s$$

Variáveis da Fórmula

- **br:** Número de blocos da relação externa
- **bs:** Número de blocos da relação interna
- **M:** Número de blocos de memória disponíveis
- **M-2:** Blocos utilizáveis para a relação externa (2 reservados para entrada/saída)

Interpretação

O termo ⌈br/(M-2)⌉ representa o número de vezes que precisamos varrer a relação interna. Cada iteração processa (M-2) blocos da relação externa contra toda a relação interna.

A eficiência aumenta significativamente com mais memória disponível. Dobrar a memória pode reduzir o custo pela metade, demonstrando o retorno não-linear do investimento em recursos de memória.

EXEMPLO PRÁTICO

Exemplo de Custos de Junção de Loop Aninhado

Vamos analisar concretamente a operação $\text{depositante} \bowtie \text{cliente}$, com depositante como relação externa.

Dados de Entrada

- Tuplas em cliente: 10.000
- Tuplas em depositante: 5.000
- Blocos de cliente: 400
- Blocos de depositante: 100

Configuração

Relação Externa: depositante (menor)

Relação Interna: cliente

40100

Pior Caso

400 × 100 + 100 blocos

500

Melhor Caso

400 + 100 blocos

Caso Geral com M blocos de memória:

$$Custo = 100 + \lceil \frac{100}{M-2} \rceil \times 400$$

Análise dos Resultados

Com M=5 blocos: $\lceil 100/3 \rceil$ × 400 + 100 = 13.500 blocos

Com M=12 blocos: $\lceil 100/10 \rceil$ × 400 + 100 = 4.100 blocos

Impacto da Memória

Aumentar a memória de 5 para 12 blocos reduz o custo em 70%, demonstrando o impacto dramático da alocação de recursos.

Junção de Merge (Merge Join)

A **junção de merge** é um algoritmo eficiente que aproveita dados ordenados para realizar junções com complexidade linear. É especialmente vantajoso quando as relações já estão ordenadas ou quando índices ordenados estão disponíveis.

01

Pré-requisito: Ordenação

Ambas as relações devem estar ordenadas pelos atributos de junção. Se não estiverem, aplica-se merge-sort externo primeiro.

02

Mesclagem Sincronizada

Varre ambas as relações simultaneamente, comparando valores dos atributos de junção em cada passo.

03

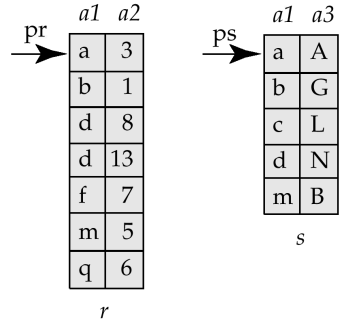
Tratamento de Duplicatas

Quando valores duplicados aparecem no atributo de junção, todas as combinações possíveis devem ser geradas (produto cartesiano local).

04

Geração do Resultado

Produz tuplas que satisfazem a condição de junção mantendo a ordem de saída.



A principal vantagem do merge join é sua previsibilidade: uma vez que as relações estejam ordenadas, o custo da junção é sempre linear, independente da distribuição dos dados.

CONTINUAÇÃO

Junção Merge: Análise Detalhada

Custo Total

O custo da junção de merge considera duas componentes principais:

$$Custo_{total} = Custo_{mesclagem} + Custo_{ordenação}$$

Custo de Ordenação

Se as relações não estiverem ordenadas:

- Relação r: $b_r \times (2\lceil \log_{M-1}(b_r/M) \rceil + 1)$
- Relação s: $b_s \times (2\lceil \log_{M-1}(b_s/M) \rceil + 1)$

Custo de Mesclagem

Caso típico: $b_r + b_s$

Cada bloco de ambas as relações é lido exatamente uma vez durante a fase de mesclagem.

Caso com Duplicatas

Se houver muitos valores duplicados no atributo de junção, pode ser necessário fazer backup e reler alguns blocos, aumentando o custo para até $b_r \times b_s$ no pior caso extremo.

Vantagens

- Performance previsível e linear
- Excelente para grandes volumes ordenados
- Aproveita índices existentes

Desvantagens

- Requer ordenação prévia (custo adicional)
- Pode ser custoso com muitas duplicatas
- Não ideal para dados não ordenados

HASH JOIN

Junção de Hash (Hash Join)

A **junção de hash** é um algoritmo poderoso que usa funções hash para particionar e juntar relações de forma eficiente, sendo especialmente efetivo quando há memória suficiente disponível.

Fase 1: Particionamento (Build Phase)

Aplica uma função hash **h** nos atributos de junção de ambas as relações, dividindo-as em partições correspondentes. Tuplas com o mesmo valor hash vão para a mesma partição.

Fase 2: Sondagem (Probe Phase)

Para cada par de partições correspondentes, constrói uma tabela hash na memória para a partição menor e sonda com a partição maior, gerando tuplas de resultado que satisfazem a junção.

Características Principais

- Não requer dados ordenados
- Custo aproximadamente linear
- Sensível à distribuição de hash
- Requer memória adequada

Função Hash Ideal

A eficiência depende de uma função hash que distribua uniformemente as tuplas entre as partições, evitando desbalanceamento que poderia causar partições muito grandes.

Princípio da Junção de Hash

O diagrama ilustra o fluxo completo do algoritmo de hash join, mostrando como as relações são particionadas usando a função hash e posteriormente processadas em pares.

Particionamento Inicial

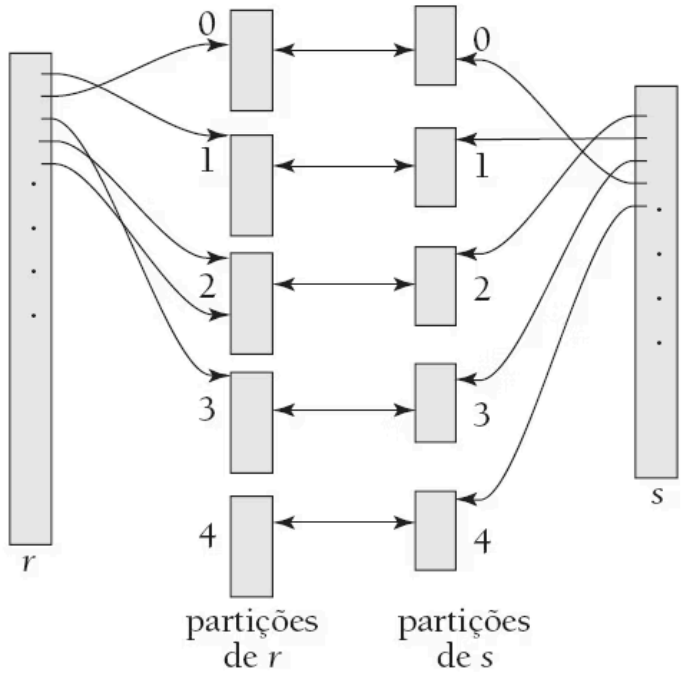
Relação r dividida em n partições: $r_0, r_1, \ldots, r_{n-1}$

Particionamento Correspondente

Relação s dividida nas mesmas n partições: $s_0, s_1, \ldots, s_{n-1}$

Processamento por Pares

Junção de r_i com s_i para todo i, combinando apenas partições correspondentes



Garantia de Correção: Tuplas que satisfazem a condição de junção sempre terão o mesmo valor hash, portanto estarão na mesma partição. Isso garante que nenhuma tupla de resultado seja perdida.

O número de partições (n) é escolhido de forma que cada partição de construção caiba confortavelmente na memória disponível, tipicamente $n = \lceil b_r / M \rceil$ onde M é o tamanho da memória.

COMPARAÇÃO

Comparações via Junção de Hash

O número de comparações realizadas durante uma junção de hash é significativamente influenciado pela qualidade da função hash e pela distribuição dos dados.

Caso Ideal

Distribuição uniforme perfeita: Cada partição tem aproximadamente o mesmo tamanho. Número de comparações ≈ (nr × ns)/n, onde n é o número de partições.

Caso Típico

Distribuição razoável: Pequenas variações no tamanho das partições. Número de comparações próximo ao ideal com overhead mínimo para tratar desbalanceamento.

Caso Problemático

Distribuição enviesada: Algumas partições muito grandes (skew). Pode degenerar para comparações próximas a nr × ns se uma partição contém a maioria das tuplas.

Otimizações Práticas

- Uso de funções hash múltiplas
- Re-particionamento de partições grandes
- Amostragem para detectar skew
- Híbridos com nested loop para partições pequenas

Métricas de Qualidade

SGBDs modernos monitoram estatísticas de hash join:

- Fator de desbalanceamento de partições
- Taxa de colisões de hash
- Número de re-particionamentos necessários

ANÁLISE FINAL

Custo da Junção de Hash e Comparação com Merge Join

Vamos consolidar a análise de custos dos principais algoritmos de junção, considerando cenários práticos e trade-offs de desempenho.

Fases do Hash Join

Particionamento, construção e sondagem

Varreduras por Relação

Cada relação é lida duas vezes

Número de Partições

Tipicamente escolhido como n ≥ br/M

Fórmula de Custo do Hash Join:

$$Custo_{hash} = 3(b_r + b_s) + 4n_h$$

Onde nh é o número de blocos usados para armazenar estruturas auxiliares de hash.

Hash Join

Melhor para: Grandes volumes de dados não ordenados, junções equi-join, quando há memória suficiente

Custo típico: 3(br + bs)

Merge Join

Melhor para: Dados já ordenados, índices disponíveis, junções que requerem saída ordenada

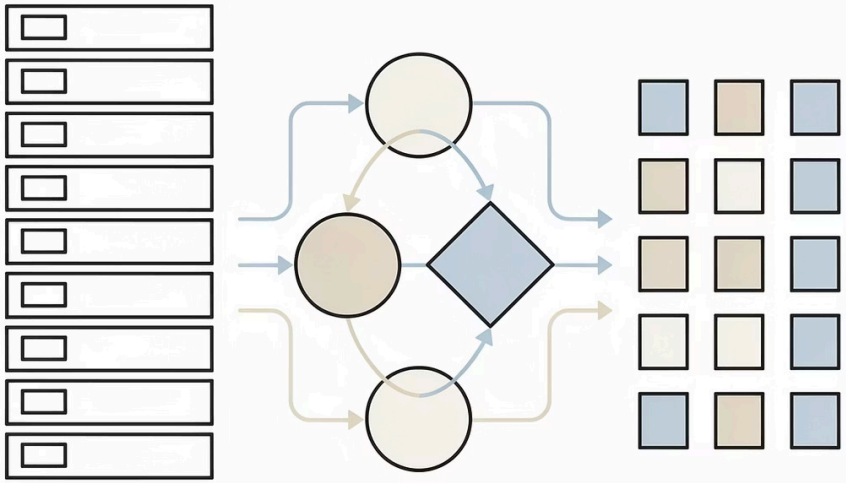
Custo típico: br + bs (se já ordenado)

Nested Loop

Melhor para: Relações pequenas, quando índices na relação interna estão disponíveis

Custo típico: br + (br × bs)/(M-2)

A escolha do algoritmo de junção ideal depende de múltiplos fatores: tamanho das relações, disponibilidade de índices, quantidade de memória, existência de ordenação prévia e seletividade da junção. Otimizadores modernos de consulta usam estatísticas detalhadas e modelos de custo para fazer esta escolha automaticamente.



Avaliação de Expressões em Bancos de Dados

Compreender como o banco de dados processa e executa consultas SQL é fundamental para otimizar o desempenho de sistemas de dados. Uma consulta não é executada de forma instantânea - ela passa por um processo estruturado de avaliação que determina como os dados serão recuperados e processados.

FUNDAMENTOS

Expressões e Suas Composições

Uma consulta SQL é, essencialmente, um **conjunto organizado de expressões**. Cada expressão representa uma composição de operações individuais que trabalham em conjunto para recuperar e transformar dados.

Quando o banco de dados recebe uma consulta, ele a converte em uma árvore de expressões - uma estrutura hierárquica que representa a ordem lógica das operações. Esta árvore precisa ser avaliada de forma eficiente, e existem estratégias distintas para isso.

Estratégias de Avaliação

O sistema de gerenciamento de banco de dados pode escolher entre duas abordagens principais para processar a árvore de expressões:

Materialização

Gera e **armazena resultados intermediários** em disco. Cada operação é executada completamente antes da próxima, e seus resultados ficam persistidos em relações temporárias.

Pipelining

Passa tuplas adiante para operações ancestrais **durante a execução**, sem esperar a conclusão completa de cada etapa. Funciona como uma linha de produção contínua.

Materialização: Avaliação Passo a Passo

A **avaliação materializada** segue uma abordagem metódica e sequencial. O sistema processa uma operação por vez, sempre começando pelo nível mais baixo da árvore de expressões e subindo gradualmente até completar toda a consulta.

01

Identificar operação base

Localiza a operação mais básica na árvore de expressões - aquela que não depende de nenhum resultado intermediário

02

Executar e materializar

Executa completamente a operação e armazena o resultado em uma relação temporária no disco

03

Subir na hierarquia

Utiliza os resultados materializados como entrada para as operações do nível superior

04

Repetir até o topo

Continua o processo até que todas as operações sejam executadas e o resultado final seja obtido

Exemplo Prático

Considere uma consulta que precisa:

1. Filtrar contas com `saldo < 2500`
2. Fazer junção com a tabela `cliente`
3. Projetar apenas `nome_cliente`

Na materialização, o sistema:

1. **Calcula e armazena** $\sigma_{saldo<2500}(conta)$
2. **Lê do disco** e faz junção com cliente
3. **Projeta** o resultado final

ANÁLISE CRÍTICA

Aspectos Críticos da Materialização

✓ Aplicabilidade Universal

A avaliação materializada **sempre funciona**, independentemente da complexidade da consulta ou dos tipos de operações envolvidas. É a abordagem mais robusta e confiável.

□ Alto Custo de I/O

O maior problema é o custo elevado de escrita e leitura de resultados intermediários. Cada operação gera disco I/O adicional, que pode dominar o tempo total de execução.

Fórmula de Custo Total

$$\text{Custo Total} = \sum_{i=1}^{n} \text{Custo}_{\text{operação}_i} + \text{Custo}_{\text{I/O intermediário}}$$

Otimização: Buffer Duplo

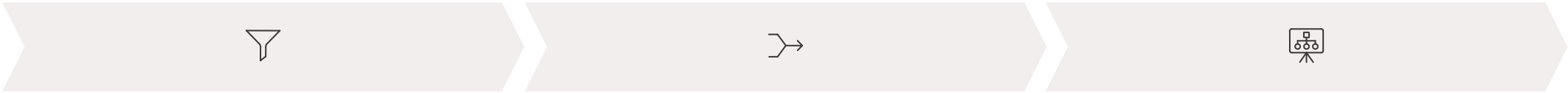
Uma técnica importante para reduzir o impacto do I/O é o **buffer duplo**. Esta estratégia utiliza dois buffers de saída para cada operação:

- **Buffer A:** Sendo escrito para o disco
- **Buffer B:** Sendo preenchido com novos dados

Isso permite **sobreposição** entre operações de escrita em disco e computação, reduzindo o tempo total de execução da consulta.

Pipelining: Processamento Contínuo

A **avaliação em pipeline** representa uma abordagem fundamentalmente diferente. Em vez de processar uma operação completamente antes de iniciar a próxima, o pipelining permite que múltiplas operações executem simultaneamente, passando resultados parciais entre si de forma contínua.



Filtrar

Processa tuplas e passa resultados imediatamente

Junção

Recebe tuplas filtradas e gera junções incrementalmente

Projetar

Aplica projeção conforme tuplas chegam

Vantagens e Limitações

Benefícios do Pipelining

- **Custo computacional reduzido** - não há necessidade de armazenar relações temporárias
- **Menor latência** - resultados começam a aparecer mais cedo
- **Uso eficiente de memória** - apenas dados ativos ocupam RAM

Quando Não é Possível

- **Operações de ordenação** - precisam ver todos os dados antes de gerar saída
- **Junção hash** - requer construção completa da tabela hash
- **Agregações complexas** - podem precisar de dados completos

Requisito chave: Para que o pipelining seja eficiente, os algoritmos de avaliação devem ser capazes de gerar tuplas de saída à medida que recebem tuplas de entrada, sem necessidade de acumular todos os dados primeiro.

CONTROLE POR DEMANDA

Pipelining Controlado por Demanda

No modelo **controlado por demanda** (pull-based), a execução é iniciada pela operação no **topo do pipeline**. Esta operação solicita tuplas conforme necessário, e cada solicitação propaga-se para baixo na árvore de expressões.

Cada operação age como um **iterador**, mantendo seu estado interno entre chamadas para saber exatamente onde parou e o que retornar na próxima solicitação.

Interface do Iterador

1	2	3
open() **Inicializa a operação** - prepara estruturas de dados, abre arquivos, e posiciona ponteiros no início. Para junção merge, isso inclui ordenar as relações e armazenar ponteiros para o início das relações ordenadas.	**next()** **Gera a próxima tupla** - processa dados incrementalmente e retorna uma tupla de resultado. Avança ponteiros internos e atualiza o estado. Na junção merge, continua o processo de mesclagem do ponto onde parou até encontrar a próxima tupla de saída.	**close()** **Finaliza a operação** - libera recursos, fecha arquivos, e limpa estruturas temporárias alocadas durante a execução.

Fluxo de Execução

Quando a operação raiz solicita uma tupla, ela chama `next()` em suas operações filhas. Essas, por sua vez, podem chamar `next()` em suas próprias operações filhas, propagando a solicitação até as folhas da árvore (operações de leitura de tabelas base). As tuplas então "sobem" pelo pipeline, sendo transformadas por cada operação até chegarem ao topo.

Pipelining Controlado por Produtor

O modelo **controlado por produtor** (push-based) inverte a direção de controle. Aqui, os operadores produzem tuplas ativamente e as empurram para seus pais assim que ficam disponíveis, sem esperar por solicitações.

Produção Ativa

Operadores filhos geram tuplas rapidamente e as colocam em buffers compartilhados

Buffer Intermediário

Mantido entre operadores - filho escreve, pai consome conforme capacidade

Controle de Fluxo

Se buffer enche, produtor aguarda; se esvazia, consumidor aguarda

Escalonamento

Sistema coordena operações com espaço disponível para processar

Comparação: Demanda vs. Produtor

Controlado por Demanda

- Controle top-down (puxar)
- Processa sob solicitação
- Mais fácil de implementar
- Menor overhead de sincronização

Controlado por Produtor

- Controle bottom-up (empurrar)
- Produção contínua e agressiva
- Melhor paralelismo potencial
- Requer gestão cuidadosa de buffers

Desafios do Modelo Produtor

O sistema precisa implementar um **escalonador sofisticado** que monitore o estado dos buffers e decida quais operações devem executar a cada momento. Operações que têm espaço em seus buffers de saída e podem processar mais tuplas de entrada recebem prioridade. Este escalonamento dinâmico garante que o pipeline continue fluindo sem bloqueios desnecessários.

Consideração importante: Ambos os modelos de pipelining requerem algoritmos de avaliação especialmente projetados que possam operar incrementalmente, gerando resultados parciais conforme processam os dados de entrada.

Referências

Elmasri & Navathe

Fundamentals of Database Systems

Pearson, 2016

Referência abrangente sobre fundamentos de sistemas de bancos de dados, cobrindo aspectos teóricos e práticos.

Korth, Sudarshan & Silberschatz

Database System Concepts

McGraw-Hill, 2019

Texto fundamental que serviu como base para a maioria dos exemplos apresentados nesta apresentação.

Özsu & Valduriez

Principles of Distributed Database Systems

Springer Nature, 2019

Obra especializada em sistemas de bancos de dados distribuídos, essencial para compreensão avançada.

Nota: Os conceitos e exemplos apresentados baseiam-se principalmente na literatura clássica de sistemas de bancos de dados, em especial *Database System Concepts* e *Fundamentals of Database Systems*.