INSTITUTO MUNICIPAL
DE ENSINO SUPERIOR DE
SÃO CAETANO DO SUL

Otimização de Consultas SQL

Professor: Aparecido
Disciplina: Recuperação de Informação
Métodos de Otimização SQL em Banco de Dados

4º BN

Ronaldo Boscatto	**08**	**52051-0**
Marcelo Rosalem Daminello	**39**	**52143-5**

INDÍCE

1. Otimização da Consulta

1.1. Introdução

A otimização da consulta apresenta tanto um desafio como uma oportunidade; um desafio para os sistemas relacionais: um desafio porque, a otimização é necessária - pelo menos nos ambientes de grande porte - , a intenção sendo que o sistema tenha um desempenho aceitável; uma oportunidade porque é precisamente uma das forças da abordagem relacional o fato de, dado o alto nível semântico das expressões relacionais, esta otimização é factível em primeiro lugar. Num sistema não-relacional, ao contrário, onde as solicitações do usuário são expressas em nível semântico inferior, toda otimização deve ser feita manualmente pelo usuário humano. Num sistema assim, é o usuário, não o sistema, quem decide que operações a nível de registro são necessárias e em que sequência devem ser executadas - e, se o usuário tomar a decisão errada, não há nada que o sistema possa fazer para melhorar a situação.

Em consequência, a vantagem da otimização não se restringe ao fato dos usuários se preocuparem quanto a melhor expressar suas consultas. Pelo contrário, há uma possibilidade real de que o otimizador o faça melhor que o programador humano, porque o otimizador pode ter informações disponíveis - em relação, por exemplo, aos valores de dados correntes - que o programador pode não ter, e é capaz de avaliar uma gama maior de alternativas do que o programador seria capaz de fazer.

O objetivo geral do otimizador, pois, é escolher uma estratégia para avaliar uma dada expressão relacional. E, mais importante, em geral não há nenhuma garantia de que a estratégia escolhida para a implementação da consulta seja realmente "ótima", qualquer que seja o padrão desta medida; pode ocorrer que seja, mas em geral tudo o que se sabe de certo é que a estratégia "otimizada"é um aperfeiçoamento da versão original não otimizada.

1.2. Um exemplo simples

Começamos com um exemplo simples que ilustra a necessidade (e também um pouco de potencial) de otimização. Consideremos a consulta "Obter nomes de fornecedores que forneçam a peça P2", para o qual a formulação em SQL possível seria:

```
SELECT DISTINCT S.SNAME
FROM S,SP
WHERE S.S# = SP.S#
AND SP.P# = 'P2' ;
```

Suponhamos que o banco de dados contenha 100 fornecedores e 10.000 expedições, das quais apenas 50 relativas à peça P2. Então a seqüência de eventos seria a seguinte:

1.Computar o produto cartesiano das relações S e SP. Este passo envolve a leitura de 10.100 tuplas, mas produz uma relação que consiste apenas em 100 * 10.000 = 1.000.000 de tuplas (e em escrever estas 1.000.000 de tuplas de volta no disco).
2.Restringir o resultado do Passo 1 como especificado pela cláusula WHERE. Este espaço envolve a leitura de 1.000.000 de tuplas, mas produz uma relação que consiste em apenas 50 tuplas (que podemos presumir que serão mantidas na memória central).
3.Projetar o resultado do Passo 2 sobre SNAME de forma a produzir o resultado final desejado(50 tuplas no máximo).

O procedimento seguinte equivale ao primeiro descrito acima (no sentido de que produz o mesmo resultado final)mas que é obviamente mais eficiente:

1.Restringir a relação SP apenas às tuplas da peça P2. Este passo envolve a leitura de 10.000 tuplas, mas produz uma relação consistindo em apenas 50 tuplas, que presume-se, serão mantidas na memória principal.
2.Fazer a junção do resultado do Passo 1 à relação S sobre S#. Este passo envolve a recuperação de apenas 100 tuplas. O resultado contém 50 tuplas (ainda na memória principal).
3.(O mesmo que o 'Passo 3 anterior). Projetar o resultado do Passo 2 sobre SNAME para produzir o resultado final desejado (50 tuplas no máximo).

Se concordarmos em tomar o "número de E/S de tuplas"como medida para o nosso desempenho ,(na prática são as E/S de página que contam, não as E/S de tupla) fica claro que o segundo desses procedimentos é cerca de 200 vezes melhor do que o primeiro. Seria melhor ainda se a relação SP fosse indexada ou tivesse acesso ***hash*** sobre P#__ o número de tuplas lidas no Passo1 seria reduzido de 10.000 para apenas 50, outro aperfeiçoamento bastante expressivo). E, naturalmente, inúmeros outros aperfeiçoamentos são possíveis.

O exemplo anterior, por mais simples que pareça, é sufiente para dar uma idéia de como é necessária a otimização. Fornece , também, uma primeira idéia sobre os tipos de aperfeiçoamentos que podem ser possíveis na prática. Na próxima seção, apresentamos como o problema na sua generalidade pode ser dividido em diversos números de problemas mais ou menos independentes. Esta abordagem sistemática serve como estrutura dentro das estratégias e técnicas individuais de otimização, como descrevem e explicam as duas seções que vêm a seguir.

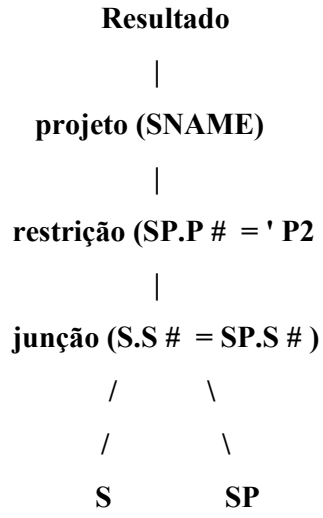
2. O Processo de Otimização: Panorama

2.1. Estágios do processo global de otimização :

2.1.1. Ordenação da consulta em determinada representação interna

O primeiro passo do processo de consulta é convertê-la numa representação interna, eliminando, assim, as considerações puramente de nível externo e, assim, abrindo o caminho para os estágios subseqüentes do processo de otimização. Levanta-se, obviamente, a questão: sobre o que o formalismo deveria basear-se a representação interna? Qualquer que seja o escolhido, deve ser rico o suficiente para representar todas as consultas possíveis na linguagem de consulta de sistema. Deve também ser o mais neutro possível, no sentido de que não prejudicará as opções subseqüentes de otimização. A forma interna usada, tipicamente, é um tipo de *árvore de sintaxe abstrata ou árvore de consulta.*

Por exemplo, a figura abaixo, mostra uma possibilidade de representação de árvore de consulta para a consulta do próximo exemplo.



Para nossos objetivos, contudo, é mais conveniente assumirmos que a representação interna usa um dos formalismos já conhecidos - a saber, a álgebra relacional ou cálculo relacional. Uma árvore de consulta como da figura anterior, pode ser considerada apenas como versão codificada de determinada expressão de um dos dois formalismos. Para assentarmos nossas idéias, partimos do princípio, neste caso, de que o formalismo é a álgebra especificamente. A expressão algébrica da consulta da figura anterior poderia ser :

((S JOIN SP) WHERE P # = ' P2 ') [SNAME]

2.1.2. Conversão à forma padrão

A maioria das linguagens permite que as consultas mais simples sejam expressas em diversas maneiras, distintas apenas superficialmente. Por exemplo, mesmo uma consulta simples como aquela discutida acima - (Obter nomes de fornecedores que forneçam a peça P2) - pode ser expressa em pelo menos sete maneiras aparentemente diferentes em SQL. O próximo passo no processamento da consulta deve ser converter a representação interna numa forma padrão equivalente, com o objetivo de se eliminar estas distinções superficiais e (mais importante) descobrir a representação mais eficiente do que a original , de uma forma específica.

Procedemos , pois, à transformação do resultado do Estágio 1 numa forma equivalente, porém mais eficiente, usando certas regras de transformação bem definidas. Um exemplo importante desta regra de transformação é aquele que permite que qualquer predicado de restrição seja convertido em seu predicado equivalente numa forma normal conjuntiva - isto é, um predicado que consista em um conjunto de comparação simples conectada apenas por Ors. Por exemplo, a cláusula WHERE:

WHERE p OR (q AND r)

pode ser convertida na forma

WHERE (p OR q) AND (p OR r)

A forma conjuntiva normal é interessante por inúmeras razões. Por exemplo, o algoritmo de decomposição da consulta usado em INGRES necessita que o predicado de entrada da consulta seja de forma conjuntiva normal, por motivos que ficarão claros logo a seguir.

Eis outro exemplo da regra de transformação: A expressão algébrica

(A JOIN B) WHERE restriction-on-B

pode ser transformada em sua expressão algébrica equivalente, porém mais eficiente

(A JOIN (B WHERE restriction-on-B))

De forma mais genérica, a expressão

(A JOIN B) WHERE restriction-on-A AND restriction-on-B

é equivalente à expressão

(A WHERE restriction-on-A) JOIN (B WHERE restriction-on-B)

Foi esta a regra que estávamos usando - taticamente - no exemplo introdutório. Exemplo simples, e este exemplo demonstrou claramente porque é conveniente este tipo de transformação. Damos abaixo três exemplos destas regras.

Dada uma determinada expressão que seja suscetível de transformação de acordo com uma das outras três.

Uma seqüência de restrições pode combinar-se numa restrição única; i.e., a expressão

(A WHERE restriction-1) WHERE restriction-2

é equivalente à expressão

A WHERE restriction-1 AND restriction-2

Numa seqüência de projeções, todas, menos a última, podem ser ignoradas; i.e., a expressão

(A [attribute-list-1]) [attribute-list-2]

equivale à expressão

A [attribute-list-2]

Uma restrição de uma projeção equivalente a uma projeção de uma restrição; i.e., a expressão

(A [attribute-list-1]) WHERE restriction-1

equivale à expressão

(A WHERE restriction-1) [attribute-list-1]

Diferentes tipos de transformação também são possíveis durante este estágio de processamento de consulta. Por exemplo, o predicado *A.F1 > B.F2 AND B.F2 = 3* , pode transformar-se na forma mais simples *A.F1 > 3* (substituindo uma junção e uma restrição por uma simples restrição).Dessa forma, o predicado *NOT (p1 AND p2)* , pode ser convertido à sua forma equivalente *(NOT p1) OR (NOT p2)* (esta conversão é realizada no DB2. Nesta última versão, fica claro que, se o predicado p1 for avaliado como falso, então não há necessidade de avaliar-se o predicado p2.

2.1.3. Escolha dos procedimentos de baixo nível

Tendo convertido a representação interna da consulta numa forma (padrão) mais desejável, o otimizador deve então decidir como avaliar a consulta transformada representada pela forma convertida. Neste estágio, considerações como a existência de índices ou outros percursos de acesso, distribuição de valores de dados armazenados, agrupamento físico de registros etc. vão ter sua participação.

A estratégia básica é considerar a expressão da consulta como a especificação de séries de operações (a níveis comparativos) de nível inferior (junção , restrição etc.), com uma certa independência entre elas. Para cada operação deste tipo, o otimizador terá avaliado para a mesma um conjunto de procedimentos de implementação pré-definido, de nível inferior. Por exemplo, haverá um conjunto de procedimentos para a implementação de operação de restrição - uma para o caso em que a restrição seja uma condição de igual num campo único, uma onde o campo da restrição seja indexado, uma onde não é indexado mas o dado fisicamente agrupado no campo da restrição, e assim sucessivamente. Cada procedimento terá uma medição de custo.

Tendo usado a informação do catálogo do sistema relativa ao estado corrente do banco de dados (existência de índices, comparaçõescardinais das relações, etc.) e tendo também usado a informação da interdependência referenciada acima, o otimizador escolherá, então, um ou mais procedimentos para a implementação de cada uma das operações na expressão da consulta. Este processo é , por vezes, chamado de *seleção de percurso de acesso.*

2.1.4. Geração de planos de consulta e escolha do melhor

O estágio final do processo de otimização envolve a construção de um conjunto de planos de consulta candidatos, seguido da escolha do melhor - i.e., o mais barato. Cada plano de consulta é construído pela combinação de um conjunto de procedimentos de implementação candidatos, um para cada operação de nível inferior da consulta. Observemos que normalmente haverá diversos planos razoáveis - provavelmente misturando vários - para cada consulta. De fato, pode não ser uma boa idéia gerar todos os planos possíveis, posto que haverá combinações de vários deles, e a tarefa de escolher o mais barato pode tornar-se bastante cara pra si; certa técnica heurística para manter o conjunto gerado dentro de limites razoáveis seria altamente desejável.

A escolha do plano mais barato naturalmente necessita de um método para atribuição de custo ao plano em questão. A maioria dos sistemas de E/S em discos envolvidos, sendo que alguns também consideram a utilização da CPU. O problema é que tudo, salvo as consultas mais simples, necessita da geração de resultados intermediários durante a execução. De forma a estimar o número de E/S em disco de forma correta, porém, é necessário estimar os tamanhos dos resultados

intermediários também, e , infelizmente, estes dependem muito dos valores de dados reais. A estimativa correta do custo é um problema difícil.

3. Estimativa do custo de acessos usando índices

O custo estimado que consideramos para expressões da álgebra relacional não levam em conta os defeitos de índices e de funções de hashing no custo de avaliação de uma expressão. A presença dessas estruturas, entretanto, tem uma importância significativa na escolha de uma estratégia de processamento de consultas.

- Índices e funções de hashing permitem acessos rápidos a registros contendo um valor específico na chave de índice.

- Índices (mas não a maioria das funções de hashing) permitem que os registros de um arquivo sejam lidos em uma ordem de classificação. Se um índice permite que registros de um arquivo sejam lidos em uma ordem correspondente à ordem física, esse índice é chamado índice de agrupamento clustering index. Os índices de agrupamento permitem-nos tirar vantagem do agrupamento físico de registros em blocos.

A estratégia detalhada para processamento de consultas é chamada plano de acesso para consulta. Um plano inclui não apenas as operações relacionais a serem executadas, mas também os índices a serm usados, a ordem na qual as tuplas serão processadas e a ordem na qual as operações serão executadas.

Obviamente, o uso de índices impõe a sobrecarga do acesso àqueles blocos contendo o índice. Precisamos levar esses acessos a blocos em conta quando estimar-mos o custo de uma estratégia que envolva o uso de índices.

Consideraremos consultas envolvendo apenas uma relação. Usamos o predicado de seleção para guiar-nos na escolha do melhor índice a ser usado no processamento de consultas.

Como um exemplo de estimativa do custo de uma consulta usando índices, assuma que estamos processando a consulta:

select número-conta
from depósito
where nome-agência = "Perryridge" and nome-cliente="Williams" and saldo>1000
Assuma também que temos as seguintes informações estatísticas sobre a relação depósito:

- 20 tuplas de depósito cabem em um bloco.
- V(nome-agência, depósito) = 50.
- V(nome-cliente, depósito) = 200.
- V(saldo,depósito) = 5.000.
- A relação depósito tem 10000 tuplas.

Vamos assumir que existam os seguintes índices em depósitos:

- Um índice em forma de árvore-B+ para nome-agência com clustering.
- Um índice em forma de árvore-B+ para nome-cliente sem clustering.

Como antes, devemos fazer a hipótese simplificadora de que os valores são distribuídos uniformemente.

Uma vez que V(nome-agência, depósito)=50, esperamos que 10000/50 = 200 tuplas da relação depósito pertençam à agência Perryridge. Se usamos o índice em nome-agência, precisaremos ler essas 200 tuplas e verificar se cada uma satisfaz a cláusula where. Uma vez que o índice tem clustering, a leitura de 200/20 = 10 blocos será necessária para ler as tuplas de depósito. E mais, diversos blocos de índice precisam ser lidos. Assuma que a árvore-B+ do índice armazena 20 ponteiros por nó. Isto significa que a árvore-B+ do índice precisa ter entre 3 e 5 nós folha. Com este número de nós folhas, a árvore interira tem uma profundidade de 2, então 2 blocos de índice precisam ser lidos. Assim, a estratégia acima requer a leitura total de 12 blocos lidos.

Se usamos o índice para nome-cliente, estimamos o número de acessos a blocos como segue. Uma vez que V(nome-cliente, depósito) = 200, esperamos que 10000/200 = 50 tuplas da relação depósito pertençam a Williams. No entanto, uma vez que o índice para nome-cliente não tem clustering, antecipamos que um bloco lido será requerido para cada tupla. Assim, 50 blocos lidos são requeridos apenas para ler as tuplas depósito. Vamos assumir que 20 ponteiros cabem em um nó da árvore-B+ para o índice de nome-cliente. Uma vez que existem 200 nomes de clientes, a árvore tem entre 11 e 20 nós folha. Assim, como no caso da árvore-B+ para o outro índice, o índice para nome-cliente tem profundidade de 2, e 2 blocos de acesso são requeridos para ler os blocos de índice necessários. Assim, essa estratégia requer um total de 52 blocos lidos. Concluímos que é preferível usar o índice para nome-agência.

Observe que, se os dois índices não tivessem clustering, preferiríamos usar o índice para nome-cliente, uma vez que esperamos apenas 50 tuplas com nome-cliente=”Williams”contra 200 tuplas com nome-agência=”Perryridge”.Sem a propriedade de clustering, nossa primeira estratégia poderia requerer o acesso a até 200 blocos para ler os dados, uma vez que, no pior dos casos, cada tupla está em um bloco diferente. Adicionamos isso aos dois acessos a blocos num total de 202 blocos lidos. Entretanto, por causa da propriedade de clustering do índice nome-agência, é realmente menos caro neste exemplo usar o índice para nome-agência.

Um outro modo pelo qual os índices podem ser usados para processar nossa consulta-exemplo é o seguinte. Use o índice para nome-cliente para obter os ponteiros para registros com nome-cliente=”Williams “ em vez dos próprios registros. Digamos que P1 represente o conjunto desses ponteiros. Da mesma forma, use o índice para nome-agência para obter os ponteiros para registros com nome-agência=”Perryridge”. Digamos que P2 represente esse conjunto de ponteiros. Então P1 P2 é um conjunto de ponteiros para registros com nome-agência= “Perryridge”e nome-cliente=”Williams”. Esses registros precisam ser recuperados e testados para ver se saldo > 1000.

Uma vez que essa técnica requer que os dois índices sejam usados, um total de quatro blocos de índice são lidos. Estimamos o número de blocos que precisam ser lidos do arquivo depósito calculando aproximadamente o número de ponteiros em
P1 P2.

Como V(nome-agência, depósito) = 50 e V(nome-cliente, depósito) = 200, estimamos que uma tupla em 50 x 200 ou uma em 10.000 tenha nome-agência = "Perryridge" e nome-cliente = "Williams". Essa estimativa é baseada em uma hipótese de distribuição uniforme e em uma hipotese adicional qua a distribuição de nomes de agência e nomes de clientes são independentes. Com base nessas tentativas, P1 ⇨ P2 é estimado como tendo apenas um ponteiro. Assim, apenas um bloco de depósito precisa ser lido. O custo total estimado desta estratégia é de cinco blocos lidos.
Não consideramos o uso do atributo saldo e do predicado saldo > 1000 como um ponto de partida para uma estratégia de processamento de consultas por duas razões:
- Não há nenhum índice para saldo.

- O predicado de seleção em saldo envolve uma comparação "maior do que". Em geral, predicados de igualdade são mais seletivos do que predicados "maior do que". Uma vez que temos um predicado de igualdade disponível (na verdade, temos dois), preferimos começar usando tal predicado já que é provável que ele selecione menos tuplas.

A estimativa de custo de acesso usando índices permite estimar o custo completo, em termos de acessos a blocos, de uma estratégia. Para uma dada expressão da álgebra relacional, pode ser possível formular diversas estratégias. A fase de seleção do plano de acesso de um otimizador de consultas escolhe a melhor estratégia para uma dada expressão.
É possível que uma expressão da álgebra relacional (para a qual um bom plano exista) seja preferível uma expressão da álgebra aparentemente mais eficiente, mas que possibilite apenas planos inferiores.

4. Estratégia de Junção

Nesta seção, aplicamos nossas técnicas para estimar o custo de processamento de uma consulta ao problema de estimar o custo de processamento de uma junho.

- A ordem física das tuplas numa relação.
- A presença de índices e o tipo de índice (com clustering ou sem clustering).
- O custo de computar um índice temporário com o único fim de procesar uma consulta.

Vamos iniciar considerando a expressão

depósitoXcliente

e assumindo que

n depósito = 10.000.

n cliente = 200.

Vamos considerar diversos métodos para processar a junção e vamos analisar seu custo respectivo.

O número de acessos a disco requeridos para computar a junção obviamente depende do tamanho do buffer e do algoritimo de substituição de páginas. Computaremos esse número para:

Cenário de pior caso. O buffer consiste em 2 blocos, um contendo um bloco da relaçao de depósito e um contendo um bloco da relação cliente.

Cenário de melhor caso. O buffer é suficientemente grande para acomodar tanto a relação depósito como a relação cliente.

Calcularemos esses números assumindo procedimentos diferentes para computar a junção.

4.1. Iteração Simples

Vamos assumir pelo momento que não temos quaisquer índeces. Se não pretendemos criar um índice, precisamos examinar cada possível par de tuplas td em depósito e tc em cliente. Portanto examinamos 10.000 * 200 = 2.000.000 pares de tuplas.

Suponha que usemos o procedimento da figura abaixo para computar a junção. Lemos cada tupla de depósito uma vez. Isto pode requerer aproximadamente 10.000 acessos a blocos se cada tupla de depósito reside em um bloco diferente. Cada tupla de cliente deve ser referenciada uma vez para cada tupla de depósito. Isto significa que referenciamos cada tupla de cliente 10.000 vezes. No pior caso, cada, cada uma dessas referências requer um acesso a disco. Uma vez que ncliente = 200, poderíamos fazer 2.000.000 de acessos para ler as tuplas de cliente. Ajuntando tudo, no pior caso poderíamos fazer até 2.010.000 acessos a bloco para processar a junção. No cenário de melhor caso, no entanto, podemos ler ambas as relações somente uma vez e executar o processamento. Isso requer no máximo 10.200 acessos a blocos, uma melhoria significativa em relação ao cenário de pior caso.

Se as tuplas de depósito são armazenadas juntas fisicamente, menos acessos são requeridos. Se assumirmos que 20 tuplas de depósito cabem em um bloco, então a leitura de depósito requer 10.000/20=500 acessos a blocos. De maneira semelhante, assumindo que 20 tuplas de cliente cabem em um bloco, então no máximo 10 acessos são requeridos para ler a relação cliente na sua totalidade. Assim, somente 10 acessos por tupla de depósito em vez de 200 são necessários. Isto implica que no cenário do pior caso, no máximo 100.000 acessos a blocos são requeridos para ler tuplas de cliente. Portanto, o custo desta abordagem simples é de 100.500 acessos a blocos. No caso do melhor cenário, no entanto, podemos ler ambas as relações somente uma vez, o que requer no máximo 520 acessos a blocos.

for each tupla d **in** depósito **do**
begin
for each tupla **in** cliente **do**
begin
teste o par (d,c) para ver se uma tupla deve ser adicionada ao resultado
end
end

Se o buffer é demasiadametne pequeno para conter inteiramente ambas as relações na memória principal, podemos ainda obter uma grande economia em acessos a blocos se processarmos as relações numa base por bloco em vez de numa base por tupla. Novamente, assumindo que as tuplas de depósito estão armazenadas fisicamente juntas, podemos usar o procedimento da figura abaixo para computar depósitoXcliente. Esse procedimento executa a junção considerando um bloco inteiro de tuplas de depósito de uma vez. Ainda precisamos ler a relação depósito inteira a um custo de 500 acessos. No entanto, em vez de leremos a relação cliente uma vez para cada tupla de depósito. Assim, no cenário do pior caso, uma vez que existem 500 blocos de tuplas de depósito e 10 blocos de tuplas de cliente, a leitura de cliente uma vez para cada bloco de tuplas de depósito requer 10 x 500 = 5.000 acessos a blocos. Assim, o custo total em termos de acesso a blocos é de 5.500 acessos (5.000 acessos para blocos de cliente mais 500 acessos para blocos de depósito). Claramente, isto é um avanço significativo sobre o número de acessos que eram necessários para nossa estratégia inicial.

for each bloco Bd **of** depósito **do**
begin
for each bloco Bc **of** cliente **do**

```
begin
   for each tupla d in Bd do
    begin
      for each tupla c in Bc do
        begin
          teste o par (d,c) para vers se uma tupla deve ser adicionada ao resultado)
        end
    end
end
end
```

Nossa escolha de depósito para o laço externo e cliente para o laço interno foi arbitrária. Se usamos cliente como a relação para o laço externo e depósito para o laço interno, o custo de nossa estratégia final seria um pouco mais baixo (5.010 acessos a blocos).

Uma vantagem importante do uso da relação menor (cliente) no laço interno é que pode ser possível armazenar a relação inteira na memória principal temporariamente. Isto acelera o processamento de consultas significativamente uma vez que é preciso ler a relação de laço interno apenas uma vez. Se cliente é realmente pequeno o suficiente para caber inteiro na memória principal, nossa estratégia requer apenas 500 blocos para ler depósito mais 10 blocos para ler cliente para um total de apenas 510 acessos a blocos.

4.2. Junção por Intercalação

Nos casos em que nenhuma relação cabe na memória principal, á ainda possível processar a junção eficientemente se ambas as relações estiverem armazenads na ordem dos atributos da junção.

Suponha que as relações *cliente* e *depósito* estejam ordenadas por *nome-cliente.* Nesse caso podemos executar a operação de *junção por intercalação (merge join).* Associamos um ponteiro a cada relação. Esses ponteiros apontam inicialmente para a primeira tupla da respectiva relação. À medida que o algoritmo é executado, os ponteiros se movem através da relação. É lido um grupo de tuplas de uma relação com memo valor nos atributos da junção. Então as tuplas (se houver) correspondentes da outra relação são lidas. Umavez que as relações estão ordenadas, as tuplas com o mesmo valor nos atributos da junção estão em ordem consecutiva. Isto nos permite ler cada tupla apenas uma vez. No caso, em que as tuplas das relações são armazenadas fisicamente juntas, esse algoritmo nos permitecomputar a junção lendo cada bloco exatamente uma vez.

A figura abaixo mostra como o esquema de junção por intercalação é aplicado a nosso exemplo de *depósito x cliente.* Nesse caso, existe um total de 510 acessosa blocos. Este método é tão bom quanto o método anterior de junção apresentado para o caso especial no qual a relação *cliente* inteira cabe dentro da memória principal. Em vez disso, é suficiente manter todas as tuplas com o mesmo valor para os atributos da junção na memória principal. Isto é viável mesmo que ambas as relações sejam grandes.

Uma desvantagem do método de junção por intercalação é que ambas as relações precisam estar classificadas fisicamente.

pd := endereço da primeira tupla de *depósito*;
pc := endereço da primeira tupla de *cliente*;
while (pc ≠ nulo) **do**
begin
tc := tupla para a qual pc aponta;
sc := {*tc*}
ajuste pc para apontar para a próxima tupla de *cliente*;
pronto := false;
while (**not** pronto and pc ≠ nulo) **do**
begin;
tc' := tupla para a qual pc aponta;
if *tc'* [*nome-cliente*] = *tc*[*nome-cliente*]
then begin
si := *sc* UNIÃO {*tc*};
ajuste *pc* para apontar para a próxima tupla de *cliente*;
end
else *pronto* := *true;*
td := tupla para a qual pd aponta;
ajuste *pd* para apontar para a próxima tupla de *depósito*;
while (*td*[*nome-cliente*] < [*nome-cliente*] **do**
begin
td := tupla para a qual *pd* aponta;

```
    ajuste pd para apontar para a próxima tupla de depósito;
  end
 while (td[nome-cliente] = tc[nome-cliente]) do
  begin
    for each t in sc do
        begin
          compute t x td e adicione ao resultado;
        end
   ajuste pd para apontar para a próxima tupla de depósito;
   td := tupla para a qual pd aponta;
  end
 end.
```

4.3. Uso de um Índice

As três estratégias consideradas dependem das técnicas físicas usadas para armazenamento das relações. A junção por intercalação requer uma ordenação. A iteração orientada a blocos requer que as tuplas de cada relação estejam armazenadas juntas fisicamente. Apenas a terceira estratégia, iteração simples, pode ser aplicada se hover tuplas sem sem clustering. O custo da iteracação simples de nosso exemplo *depósito x cliente* é de 2 milhões de acessos a blocos. Quando um índice é usado, mas sem qualquer hipótese sendo feita sobre o armazenamento físico, a ligação pode ser computada com significamente menos acessos a blocos.

Freqüentemente, os atributos da junção formam uma chave de busca para um índice de uma das relações da junção. Nesse caso, podemos considerar uma estratégia de junção que usa tal índice. A estratégia simples da Figura 5.1 pode ser feita mais eficientemente se existir um índice de *cliente* usando *nome-cliente.* Dada a tupla *d* em *depósito,* não é mais necessário ler a relação *cliente* inteiro. Em vez disso, o índice é usado para buscar as tuplas em *cliente* para as quais o valor *nome-cliente* é *d[nome-cliente]*.

Ainda precisamos de 10.000 acessos para ler *depósito*. Entretanto, para cada tupla de *depósito* apenas uma busca de índice é necessária. Se assumirmos (como antes) que *nclientes* = 200, e que 20 ponteiros cabem em um bloco, então essa pesquisa requer no máximo o acesso a dois blocos de índices, mais um bloco de acesso para ler a tupla de *cliente* propriamente dita. Fazemos o

acesso a tr6es blocos por tupla de *depósito* em vez de 200. Adicionando isto aos 10.000 acessos para ler *depósito,* descobrimos que o custo total dessa estratégia é de 40.000 acessos.

Embora esse custo de 40.000 acessos possa parecer alto, precisamos lembrar que encontramos estratégias mais eficientes apenas quando assumimos que as tuplas estavam armazenadas fisicamente juntas. Se essa hipótese não valer para as relações que estão sento juntadas, então a estratégia que acabamos de apresentar é altamente desejável. Realmente, a economia de 160.000 acessos é suficiente para justificar a criação do índice. Mesmo que crie o índice com o único propósito de processar esta única consulta e eliminá-la depois, podemos executar menos acessos do que se usasse a estratégia da iteração simples.

4.4. Junção com Hashing

Pode ser compensador construir um índice especificamente para o uso na computação de uma junção, mesmo que esse índice não seja retido após a computação da junção. Em vez de construir um índice em forma de árvore-B+, é freqüentemente preferível usar o hashing para um índice do tipo "use uma vez" construído para auxiliar a computação de uma única junção.

Uma função de hashing *h* é usada para as tuplas de ambas as relações sobre os atributos da função. Os buckets resultantes, que contêm ponteiros para tuplas das relações, são usados para limitar o número de pares de tuplas que devem ser comparados. Se *d* é uma tupla de *depósito e c* uma tupla de *cliente,* então *d* e *c* devem ser testadas apenas se *h(c) = h(d).* Se *h(c)* DIFERENTE *h(d),* então *c* e *d* devem ter valores diferentes para *nome-cliente.* Entretanto, se *h(c) = h(d)* precisamos testar *c* e *d,* uma vez que é possível que eles tenham valores para *nome-cliente* que dão o mesmo valor de hashing.

A figura abaixo mostra os detalhes do algoritmo de junção com hashing como aplicado ao nosso exemplo de *depósitos x clientes.* A função de hashing *h* deveria ter as boas propriedades de aleatoriedade e uniformidade. Usaremos essas propriedades para estimar o custo da execução da junção com hashing. Na Figura 5.4 assumimos que:

- *h* é uma função de hashing mapeando valores de *nome-cliente* em {0, 1, ..., max}.
- *Hc0, Hc1... Hcmax* representam buckets de ponteiros para tuplas de *cliente,* cada um inicialmente vazio.
- *Hd0, Hd1... Hdmax* representam buckets de ponteiros para tuplas de *depósito,* cada um iniciando vazio.

Usamos em seguida essas propriedades para estimar o custo da execução de uma junção com hashing.

A distribuição de ponteiros para buckets de hashing nos dois laços **for** do algoritmo requer uma leitura completa de ambas as relações. O custo desta operação requer 510 acessos a blocos se as tuplas de *depósito* e as tuplas de *cliente* estiverem armazenadas juntas fisicamente. Uma vez que os buckets contém apenas ponteiros, assumimos que eles cabem na memória principal, assim nenhum acesso a disco é necessário para fazer acesso aos buckets.

A parte final do algoritmo varre os valores assumidos por *h.* Digamos que *i* sejaum valor de *h.* O laço **for** final externo computa

rd x rc

onde *rd* é o conjunto de tuplas de *depósito* cujo valor de hashing as coloca no bucket *i* e *rc* é o conjunto de tuplas de *cliente* cujo valor de hashing leva-as ao bucket *i.* Esta junção é computada usando a iteração simples, uma vez que esperamos que *rd* e *rc* sejam suficientemente pequenos para caber na memória principal. Uma vez que o hashing de uma tupla leva-a exatamente em um bucket, cada tupla é lida apenas uma vez pelo laço **for** final externo. Como observamos anteriormente, isto requer 510 acessos a bloco. Assim, o custo total estimado de uma junção com hashing cujo contra domínio seja grande o suficeiente para asseurar que os buckets contenham um número suficientemente pequeno de ponteiros para que *rc* e *rd* caibam na memória principal. O otimizador não deve escolher uma função de hashing que tenha um contra domínio tão grande que os buckets fiquem vazios. Isto gastaria espaço e forçaria o algoritmo de junção com hashing a incorrer em desperdício processando buckets vazios.

for each tupla *c* **in** *cliente* **do**
begin
i := h(c[nome-cliente]);

```
    Hci := Hci UNIÃO {ponteiro para c};
  end
for each tupla d in depósito do
  begin
    i := h(d[nome-cliente]);
    Hci := Hdi UNIÃO {ponteiro para d};
  end
for i := 0 to max do
  begin
    rc := □ ;
    rd := □ ;
        for each ponteiro pc in Hci do
          begin
            c := tupla para a qual pc aponta
            rc := rc UNIÃO {c}
          end
        for each tupla d in rd do
          begin
            for each tupla d in rc  do
              begin
                teste o par (d,c) para ver se uma tupla
                poderia ser adicionada ao resultado
              end
          end
  end.
```

4.5. Junção Tripla

Vamos agora considerar uma junção envolvendo três relações:

agência x depósito x client

Assuma que *ndepósito* = 10.000, *ncliente* = 200 e *nagência* = 50. Não apenas temos uma escolha de estratégia para o processamento de junção, mas temos também uma escolha de qual

junção computaremos primeiro. Existem muitas estratégias possíveis a considerar. Analisaremos diversas delas a seguir e deixaremos outras como exercícios para o leitor.

- **Estratégia 1**. Compute a junção depósitoxcliente usando uma das técnicas apresentadas anteriormente. Uma vez que nome-cliente é uma chave de cliente, sabemos que o resultado dessa junção tem no máximo 10.000 tuplas (o número de tuplas em depósito). Se construirmos um índice em agência para nome-agência, podemos computar

  agência x (depósito x cliente)

considerando cada tupla *t* de (*depósito x cliente*) e buscando a tupla em agência com um valor de *nome-agência* igual a *t[nome-agência]*. Uma vez que *nome-agência* é uma chave de agência, sabemos que precisamos examinar apenas uma tupla de *agência* para cada uma das 10.000 tuplas em (*depósito x cliente*). O número exato de acessos a blocos requeridos por esta estratégia depende do modo pelo qual computamos (*depósito x cliente*) e do modo pelo qual agência é armazenada fisicamente. Diversos exercícios examinam os custos de várias possibilidades.

- **Estratégia 2**. Compute uma junção tripla sem construir qualquer índice. Isto requer a verificação de 50*10.000*200 possibilidades, ou num total de 100.000.000.

- **Estratégia 3**. Em vez de executarmos duas junções, executamos um par de junções de cada vez. Essa técnica primeira envolve a construção de dois índices:

  Sobre *agência* usando *nome-agência*
  Sobre *cliente* usando *nome-cliente*.

Em seguida consideramos cada tupla *t* em *depósito*. Para cada *t*, buscamos as tuplas correspondentes em cliente e as tuplas correspondentes em *agência*. Assim, examinamos cada tupla de *depósito* exatamente uma vez..

A estratégia 3 representa uma forma que não havíamos considerado anteriormente. Esta não corresponde diretamente a uma operação da álgebra relacional. Em vez disso, combina duas operações e uma operação especial. Com a estratégia 3, é frequentemente possível executar uma junção de três relações mais eficientemente do que usando duas junções de duas relações. Os custos

relativos dependem do modo pelo qual as relações estão armazenadas, da distribuição de valores dentro das colunas e da presença de índices. Os exercícios oferecem oportunidade de computar esses custos em diversos exemplos.

5. Estratégias de Junção para processadores paralelos

As estratégias de junção que consideramos até agora assumem um único processador está disponível para computar a junção. Nesta seção, consideramos o caso em que diversos processadores estão disponíveis para a computação paralela de uma junção. Assumimos um ambiente de multiprocessamento no qual os processadores são parte de um sistema de computadores, todos dividindo uma única memória principal.

Numerosas arquiteturas tem sido propostas para processadores paralelos para aplicações de banco de dados. Muitas dessas máquinas de banco de dados são discutidas em referências de notas bibliográficas. Consideraremos uma arquitetura simples com os seguintes recursos:

- Todos os processadores tem acesso a todos os discos.

- Todos os processadores compartilham a memória principal.

As técnicas apresentadas a seguir para o processamento paralelo de junções podem ser adaptadas as outras arquiteturas nas quais cada processador tem sua própria memória particular.

5.1. Junção Paralela

Nas técnicas que discutimos para processar junções em um único processador, a eficiêcncia é obtida pela redução do número de pares de tuplas que precisam ser testados. A meta de um algoritimo de junção paralela é dividir os pares a serem testados entre os diversos processadores. Cada processador então computa parte da junção. No passo final, os resultados de cada processador são coletados para produzir o resultado final.

Idealmente, o trabalho geral de computar a junção é particionado igualmente entre todos os processadores. Se tal divisão é feita sem qualquer sobrecarga, uma junção paralela usando N processadores tomará 1/N do tempo que a mesma junção tomaria em um único processador. Na prática, o aceleramento é menos dramático por diversas razões:

- Ocorre sobrecarga ao se particionar o trabalho entre os processadores.

- Ocorre sobrecarga ao se coletar os resultados computados para cada processador para produzir o resultado final.

- O esforço feito para dividir o trabalho igualmente é apenas uma aproximação, assim alguns processadores podem ter mais trabalho do que outros. O resultado final não pode ser obtido até que o último processador tenha de fato terminado.

- Os processadores podem competir por recursos compartilhados do sistema. Isto resulta em demoras à medida que os processadores esperam que outros processadores liberem os recursos.

Vamos considerar novamente nossos exemplos de depósitos x cliente, assumindo que temos *N* processadores *P1, P2, ... ,PN*. Dividimos depósito em *N* participações de igual tamanho: *depósito 1, depósito2,...,depósitoN*. (Para simplificar, assumimos que o tamanho da relação *depósito* é um múltiplo de *N*). Então cada processador *Pi* computa depósitoixcliente em paralelo. No passo final, computamos a união dos resultados parciais computados por cada processador.

O custo desta estratégia depende de diversos fatores:

- A escolha do algoritimo de junção usado por cada processador.

- O custo de montagem do resultado final.

- Os atrasos impostos pela contanção de recursos. Embora cada processador use sua própria partição de depósito, todos os processadores fazem acesso a cliente. Se a memória principal não é

suficientemente grande para guardar a relação cliente inteira, os processadores precisam sincronizar seus acessos a cliente para reduzirem o número de vezes que cada bloco de cliente precisa ser lido do disco.

O potencial de contenção da memória principal ao armazenar tuplas de cliente sugere que tomemos alguns cuidados na divisão do trabalho entre os processadores para reduzir a contenção. Existem muitos modos de fazer isso. Uma técnica simples é usar uma versão paralela do algoritimo de junção com hashing.

Escolhemos uma função de hashing cujos limiters são *{1,2,...,N}*. Isto permite atribuir cada um dos *N* processadores para exatamente um dos buckets do hashing. Uma vez que o laço final externo for do algoritimo atua sobre os buckets, cada processador a iteração que corresponde ao seu bucket atribuído. Nenhuma tupla é atribuída a mais de um bucket, assim não há contenção para as tuplas de cliente. Uma vez que cada processador considera um par de tuplas por vez, o total da requisição de memória particular pelo algoritimo é suficientemente baixo e a contenção de espaço na memória principal é improvável.

5.2. Junção Múltipla em duto (PIPELINED)

Nesta seção, exploramos a possibilidade de computar diversas junções em paralelo. Esta é uma questão importante, uma vez que muitas consultas do mundo real, particularmente aquelas expressas em termos de uma visão, envolvem diversas relações.

Vamos considerar uma junção de quatro relações.

r1 x r2 x r3 x r4

Claramente, podemos computar *t1 r1 x r2* em paralelo com *t2 r3 x r4*. Quando essas duas computações estiverem completas, computamos.

t1xt2

Um paralelismo ainda maior pode ser feito ajustando um duto (*"pipeline"*) que permite que as três computações sejam feitas em paralelo. Digamos que o processador

execute a computação *r1 x r2* e digamos que *P2* execute *r3 x r4*. À medida que *P1* computa tuplas em *r1 x r2*, ele deixa essas tuplas disponíveis para o processador *P3*. Da mesma forma, à medida que o processador *P2* computa tuplas em *r3 x r4*, ele torna essas tuplas disponíveis para *P3*. Assim, *P3* tem disponíveis algumas tuplas de *r1 x r2* e *r3 x r* antes que os processadores *P1* e *P2* tenham acabado totalmente seus serviços. *P3* pode usar essas tuplas disponíveis para iniciar a computação de *(r1 x r2) x (r3 x r4)* antes mesmo que *r1 x r2* e *r3 x r4* tenham sido completamente computadas.

Essa junção em duto, que mostra um "fluxo" de tuplas de *P1* para *P3* e de *P2* para *P3*. Em nossa máquina paralela assumida, as tuplas são passadas via memória principal partilhada. Esta técnica é aplicável a outras arquiteturas paralelas.

Os processadores *P1* e *P2* estão livres para usar qualquer um dos algoritimos de junção que consideramos antes. A única modificação é que quando uma tupla *t* é adicionada ao resultado, *t* precisa ser tornada disponível para *P3* colocando tna fila. Além disso, uma entrada especial de fila consistindo em *ENDP1* e *ENDP2*, respectivamente, é feita após o término da computação.

pronto1: = false
pronto2: = false
de1: = □ ;
de2: = □ ;
resultado: = □ ;
while not *pronto 1* **or not** *pronto 2* **do**
begin
if fila esta vazia then espere até que a fila não esteja vazia;
t: = entrada mais alta na fila;
if *t = ENDP1* **then** *pronto 1 := true*
else if *t = ENDP2* **then** *pronto 2: = true*
else if *t* é de *P1* **then**
begin
de 1: = de1 U{t};
resultado: = resultado U({t}x de 2);
end

```
            else /*t é de P2*/
               begin
                     de 2: = U{t};
                resultado: = resultado U (de 1x {t});
               end
end
```

O conceito ilustrado pela computação em duto de uma junção quádrupla *r1x r2 x r3 x r4* pode ser estendido para manipular junções com n relações.

6. Organização Física

As técnicas que consideramos para a computação de junção paralela aumentam a taxa em que ocorrem acessos a disco. Para a junção paralela dupla da Seção 6.1, vimos que, escolhendo cuidadosamente o modo pelo qual a relação é particionada, poderíamos reduzir a contenção de disco. Entretanto, para aquela técnica, assim como para a técnica da junção em duto da seção precedente, o disco provavelmente será o gargalo.

A fim de reduzir a contenção nos acessos a disco, o banco de dados pode ser particionado em diversos discos. Isto permite diversos acessos a disco a serem servidos em paralelo. Entretanto, a fim de explorarmos o potencial de acessos paralelos a disco, precisamos escolher uma boa distribuição de dados entre os discos.

O algoritimo de junção paralela dupla requer diversos processadores para fazer o acesso a relações em paralelo. A fim de reduzir a contenção, é útil distribuir as tuplas de relações individuais entre os diversos discos. Esta técnica é chamada fatiamento de disco (disk striping). Vamos considerar um exemplo de fatiamento de disco particularmente bem adequado à versão paralela da junção com hashing que apresentamos na Seção 6.1.

Usamos a função de hashing do algoritimo de junção com hashing que atribui tuplas ao disco. Todos os grupos de tuplas que partilham um bucket são atribuídos para o mesmo disco. A cada grupo é designado um disco separado se possível. De qualquer forma, os grupos são distribuídos uniformemente entre os discos disponíveis. Essa forma de fatiamente permite que a

junção paralela dupla com hashing explore acessos a disco paralelos. No caso em que cada grupo é atribuído a um disco separado, não há contençõ para acessos a disco!

A técnica de fatiamento de disco é menos útil para a junção em duto da Seção 6.2. Para essa junção, é desejável que cada relação seja mantida em um disco e que relações distintas sejam atribuídas a discos separados até o grau possível. No esquema para a Figura 9.5, para computação de *(r1 x r2) x (r3 x r4),* se cada relação está em um disco diferente, a contenção é eliminada pelos processadores *P1* e *P2.*

É claro que a organização física ótima difere para consultas diferentes. O administrador do banco de dados precisa escolher uma organização física que acredite ser boa para a composição esperada de consultas do banco de dados. O otimizador de consultas do sistema de banco de dados precisa escolher entre as várias técnicas paralelas e sequenciais que consideramos, estimando o custo de cada técnica em uma dada organização física.

7. Estrutura do Otimizador de Consultas

Discutimos apenas algumas das muitas estratégias de processamento de consultas usadas em vários sistemas comerciais de banco de dados. Como a maioria dos sistemas implementa apenas umas poucas estratégias, o número de estratégias, o número de estratégias a ser considerado pelo otimizador de consultas é limitado. Outros sistemas consideram um grande número de estratégias. Para cada estratégia, é computado um custo estimado.

Alguns sistemas reduzem o número de estratégias que necessitam ser levadas em consideração fazendo uma estimativa heurística de uma boa estratégia. Seguindo isto, o otimizador considera cada estratégia possível, mas finaliza tão logo ele determina que o custo é maior do que a melhor das estratégias consideradas anteriormente. Se o otimizador inicia com uma estratégia que provavelmente é de baixo custo, apenas umas poucas estratégias competitivas requererão uma análise completa de custo. Isto pode reduzir as despesas gerais do otimizador de consultas.

A fim de simplificar a tarefa de seleção de estratégia, uma consulta pode ser dividida em diversas subconsultas. Isto não apenas simplica a seleção de estratégia, mas também permite ao otimizador de consultas reconhecer casos em que uma subconsulta particular aparece diversas vezes na mesma consulta. Se tais subconsultas são computadas apenas uma vez, o tempo é economizado na fase de otimização de consulta e na execução da própria consulta. O reconhecimento de subconsults iguais é análogo ao reconhecimento de subexpressões iguais em muitos compiladores com otimização para linguagens de programação.

Claramente, o exame de uma consulta em busca de subconsultas iguais e a estimativa de custo de um grande número de estratégias impõem uma sobrecarga de trabalho substancial no processamento de consultas. Entretanto, o custo adicional da otimização de consulta é normalmente mais do que compensador pela economia no tempo de execução de uma consulta. A economia é ampliada nas aplicações processadas regularmente e reexecutam as mesmas consultas em cada execução. Por isso, a maioria dos sistemas comerciais incluem otimizadores relativamente complexos. As notas bibliográficas dão referências a descrições de otimizadores de consultas de sistemas de banco de dados.

Bibliografia :

Livros: **Sistemas de bancos de dados**

Herry F. Korth

Abraham Silberchatz

2º Ed - SP