Sistemas de Gerenciamento de Banco de Dados

Tradução da
Terceira Edição

Mc Graw Hill

Ramakrishnan • Gehrke

Sistemas de Gerenciamento de Banco de Dados
ISBN 978-85-7726-027-0

A reprodução total ou parcial deste volume por quaisquer formas ou meios, sem o consentimento escrito da editora, é ilegal e configura apropriação indevida dos direitos intelectuais e patrimoniais dos autores.

Copyright © 2008 de McGraw-Hill Interamericana do Brasil Ltda.
Todos os direitos reservados.
Av. Brigadeiro Faria Lima, 201 – 17º. andar
São Paulo, SP, CEP 05426-100

Todos os direitos reservados. Copyrigth © 2008 de McGraw-Hill Interamericana Editores, S. A. de C. V.
Prol. Paseo de la Reforma 1015 Torre A Piso 17, Col. Desarrollo Santa Fé, Delegación Alvaro Obregón
México 01376, D. F., México

Tradução da terceira edição do original em inglês Database Management Systems.
© 2003, 2000, 1998 de The McGraw-Hill Companies, Inc.
ISBN da obra original: 0-07-246563-8

Diretor-Geral: *Adilson Pereira*
Editora: *Gisélia Costa*
Supervisora de Produção: *Guacira Simonelli*
Preparação de Texto: *Lucrécia Freitas e Mônica de Aguiar*
Design da Capa: *Mick Wiggins*
Editoração Eletrônica: *Crontec Ltda*.

R165s Ramarkrishnan, Raghu.
Sistemas de gerenciamento de banco de dados [recurso eletrônico] / Raghu Ramarkrishnan,Johannes Gehrke ; tradução: Célia Taniwake. – 3. ed. – Dados eletrônicos. – Porto Alegre : AMGH, 2011.

Editado também como livro impresso em 2008.
ISBN 978-85-63308-77-1

1. Ciência da computação. 2. Bases de dados – Gerenciamento. I. Gehrke, Johannes. II. Título.

CDU 004.658

Catalogação na publicação: Ana Paula Magnus – CRB 10/2052

22.6.1 Tipos de Bancos de Dados Distribuídos

Se os dados são distribuídos, mas todos os servidores executam o mesmo software de SGBD, temos um **sistema de banco de dados distribuído homogêneo**. Se diferentes sites são executados sob o controle de diferentes SGBDs, de forma basicamente autônoma, e são conectados que de algum modo permita o acesso aos dados a partir de vários sites, temos um **sistema de banco de dados distribuído heterogêneo**, também referido como **sistema de múltiplos bancos de dados (multidatabase)**.

O segredo da construção de sistemas heterogêneos é ter padrões bem aceitos para **protocolos de gateway**. O protocolo de gateway é uma API que expõe funcionalidade de SGBD para os aplicativos externos; seus exemplos incluem ODBC e JDBC (consulte a Seção 6.2). Acessando-se servidores de banco de dados por meio de protocolos de gateway, suas diferenças (na capacidade, formato de dados etc.) são mascaradas, e as diferenças entre os diferentes servidores em um sistema distribuído são bastante atenuadas.

Contudo, os gateways não são uma panacéia. Eles acrescentam uma camada de processamento que pode ser dispendiosa e não mascaram completamente as diferenças entre os servidores. Por exemplo, um servidor pode não ser capaz de fornecer os serviços exigidos para o gerenciamento de transação distribuída (consulte as Seções 22.13 e 22.14) e, mesmo que seja capaz, a padronização dos protocolos de gateway até esse nível de interação impõe desafios que ainda não foram resolvidos satisfatoriamente.

O gerenciamento de dados distribuídos, na análise final, tem um custo significativo em termos de desempenho, complexidade do software e dificuldade de administração. Essa observação é particularmente verdadeira para os sistemas heterogêneos.

22.7 ARQUITETURAS DE SGBD DISTRIBUÍDO

Três estratégias alternativas são usadas para separar funcionalidade entre diferentes processos relacionados ao SGBD; essas arquiteturas de SGBD distribuído são chamadas *cliente-servidor*, *servidor colaborador* e *middleware*.

22.7.1 Sistemas Cliente-Servidor

O sistema **cliente-servidor** tem um ou mais processos clientes e um ou mais processos servidores, e um processo cliente pode enviar uma consulta para qualquer processo servidor. Os clientes são responsáveis por questões da interface com o usuário e os servidores gerenciam dados e executam transações. Assim, um processo cliente poderia ser executado em um computador pessoal e enviar consultas para um servidor sendo executado em um computador de grande porte.

Essa arquitetura tornou-se muito popular por vários motivos. Primeiramente, ela é relativamente simples de implementar, devido à separação clara da funcionalidade e porque o servidor é centralizado. Segundo, as máquinas servidoras caras não ficam sub-utilizadas, lidando com interações simples dos usuários, que agora ficam relegadas às máquinas clientes baratas. Terceiro, os usuários podem executar uma interface gráfica com a qual estão familiarizados, em vez da (possivelmente desconhecida e pouco amigável) interface com o usuário do servidor.

Ao escrever aplicativos cliente-servidor, é importante lembrar o limite entre o cliente e o servidor, e manter a comunicação entre eles orientada a conjunto máximo possível. Em particular, abrir um cursor e buscar uma tupla por vezes gera muitas mensagens e isso deve ser evitado. (Mesmo que busquemos várias tuplas e as coloquemos na cache do cliente, mensagens devem ser trocadas quando se avança o cursor, para garantir que a linha corrente seja bloqueada.) Técnicas para explorar a cache da arquitetura

cliente-servidor para reduzir a sobrecarga de comunicação têm sido amplamente estudadas, embora não as discutamos mais.

22.7.2 Sistemas de Servidor Colaborador

A arquitetura cliente-servidor não permite que uma única consulta abranja vários servidores, pois o processo cliente teria de ser capaz de subdividir tal consulta nas subconsultas apropriadas, para serem executadas em diferentes sites e, depois, reunir as respostas das subconsultas. Portanto, o processo cliente seria muito complexo e seus recursos começariam a se sobrepor ao servidor; torna-se mais difícil distinguir entre clientes e servidores. A eliminação dessa distinção nos leva a uma alternativa para a arquitetura cliente-servidor: um sistema de **servidor colaborador**. Podemos ter uma coleção de servidores de banco de dados, cada um capaz de executar transações com dados locais, os quais executam cooperativamente as transações que abrangem vários servidores.

Quando um servidor recebe uma consulta que exige acesso aos dados que estão em outros servidores, ele gera subconsultas apropriadas para serem executadas pelos outros servidores e reúne os resultados para computar as respostas da consulta original. De preferência, a decomposição da consulta deve ser feita usando-se otimização baseada em custo levando em conta o custo da comunicação na rede, assim como os custos do processamento local.

22.7.3 Sistemas de Middleware

A arquitetura middleware é projetada para permitir que uma única consulta abranja vários servidores, sem exigir que todos os servidores de banco de dados sejam capazes de gerenciar tais estratégias de execução em vários sites. Ela é particularmente atraente ao se tentar integrar vários sistemas legados, cujos recursos básicos não podem ser estendidos.

A idéia é que precisamos de apenas um servidor de banco de dados capaz de gerenciar consultas e transações que abranjam vários servidores; os servidores restantes só precisam manipular consultas e transações locais. Podemos considerar esse servidor especial como uma camada de software que coordena a execução de consultas e transações em um ou mais servidores de banco de dados independentes; tal software é freqüentemente chamado de **middleware**. A camada de middleware é capaz de executar junções e outras operações relacionais em dados obtidos de outros servidores, mas normalmente ela própria não mantém dados.

22.8 ARMAZENANDO DADOS EM UM SGDB DISTRIBUÍDO

Em um SGBD distribuído, as relações são armazenadas em vários sites. O acesso a uma relação armazenada em um site remoto acarreta custos de passagem de mensagem e, para reduzir essa sobrecarga, uma única relação pode ser *particionada* ou *fragmentada* entre vários sites, com os fragmentos armazenados nos sites onde são mais freqüentemente acessados ou *replicados* em cada site onde a relação tem alta demanda.

22.8.1 Fragmentação

A **fragmentação** consiste em subdividir uma relação em relações menores ou fragmentos e armazenar os fragmentos (em vez da relação em si), possivelmente em diferentes sites. Na **fragmentação horizontal**, cada fragmento consiste em um subconjunto de *linhas* da relação original. Na **fragmentação vertical**, cada fragmento consiste em um subconjunto de *colunas* da relação original. Os fragmentos horizontais e verticais estão ilustrados na Figura 22.4.

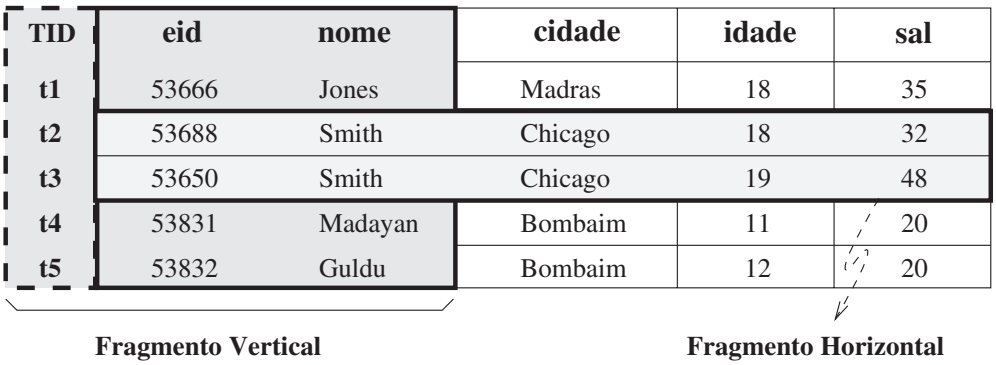

Figura 22.4 Fragmentações horizontal e vertical.

Normalmente, as tuplas pertencentes a determinado fragmento horizontal são identificadas por uma consulta de seleção; por exemplo, as tuplas de funcionários poderiam ser organizadas em fragmentos por cidade, com todos os funcionários de determinada cidade atribuídos ao mesmo fragmento. O fragmento horizontal mostrado na Figura 22.4 corresponde a Chicago. Armazenando fragmentos no site do banco de dados na cidade correspondente, obtemos a localidade de referência — os dados de Chicago provavelmente serão atualizados e consultados a partir de Chicago, e armazenar esses dados nessa cidade os torna locais (e reduz os custos de comunicação) para a maioria das consultas. Analogamente, as tuplas de determinado fragmento vertical são identificadas por uma consulta de projeção. O fragmento vertical na figura resulta da projeção nas duas primeiras colunas da relação de funcionários.

Quando uma relação é fragmentada, devemos ser capazes de recuperar a relação original a partir dos fragmentos:

- **Fragmentação horizontal:** a união dos fragmentos horizontais deve ser igual à relação original. Normalmente, os fragmentos também são obrigados a ser disjuntos.
- **Fragmentação vertical:** a coleção de fragmentos verticais deve ser uma decomposição sem perda de junção, conforme a definição do Capítulo 19.

  Para garantir que uma fragmentação vertical seja sem perda de junção, os sistemas freqüentemente atribuem um único id a cada tupla da relação original, como mostra a Figura 22.4, e anexam esse id na projeção da tupla em cada fragmento. Se considerarmos a relação original contendo um campo de id de tupla adicional que é uma chave, esse campo será adicionado em cada fragmento vertical. É garantido que tal decomposição é sem perda de junção.

Em geral, uma relação pode ser fragmentada (horizontal ou verticalmente) e cada fragmento resultante pode ser ainda mais fragmentado. Para simplicidade na exposição, no restante deste capítulo vamos supor que os fragmentos não são particionados recursivamente dessa maneira.

22.8.2 Replicação

Replicação significa que armazenamos várias cópias de uma relação ou fragmento de relação. Uma relação inteira pode ser replicada em um ou mais sites. Analogamente, um ou mais fragmentos de uma relação podem ser replicados em outros sites. Por exemplo, se uma relação R é fragmentada em $R1$, $R2$ e $R3$, poderia haver apenas uma cópia de $R1$, enquanto $R2$ seria replicada em dois outros sites e $R3$ seria replicada em todos os sites.

A motivação para a replicada é dupla:

- **Maior disponibilidade dos dados:** se um site que contém uma réplica fica inativo, podemos encontrar os mesmos dados em outros sites. Analogamente, se cópias locais de relações remotas estão disponíveis, ficamos menos vulneráveis à falha de links de comunicação.
- **Avaliação de consulta mais rápida:** as consultas podem ser executadas mais rapidamente usando uma cópia local de uma relação, em vez de ir até um site remoto.

Os dois tipos de replicação, chamados de replicação *síncrona* e *assíncrona*, diferem principalmente no modo como as réplicas são mantidas atualizadas quando a relação é modificada (consulte a Seção 22.11).

22.9 GERENCIAMENTO DE CATÁLOGO DISTRIBUÍDO

Monitorar dados distribuídos por vários sites pode ser complicado. Devemos monitorar como as relações são fragmentadas e replicadas, isto é, como os fragmentos da relação são distribuídos entre os vários sites e onde as cópias dos fragmentos são armazenadas —, além do esquema, autorização e informações estatísticas usuais.

22.9.1 Atribuindo Nomes a Objetos

Se uma relação é fragmentada e replicada, devemos ser capazes de identificar univocamente cada réplica de cada fragmento. A geração de tais nomes únicos exige certo cuidado. Se usarmos um servidor de nomes global para atribuir nomes globalmente únicos, a autonomia local ficará comprometida; queremos que (os usuários de) cada site seja capaz de atribuir nomes a objetos locais sem fazer referência a nomes do sistema.

A solução usual para o problema da atribuição de nomes é usar nomes compostos de vários campos. Por exemplo, poderíamos ter:

- Um campo de *nome local*, que é o nome atribuído de forma local no lugar onde a relação é criada. Dois objetos em diferentes sites poderiam ter o mesmo nome local, mas dois objetos em determinado site não podem ter o mesmo nome local.
- Um campo de *site de nascimento*, identificando o site onde a relação foi criada e onde são mantidas as informações sobre todos os fragmentos e réplicas da relação.

Esses dois campos identificam uma relação univocamente; podemos chamar a combinação de **nome global da relação**. Para identificarmos uma réplica (de uma relação ou do fragmento de uma relação), pegamos o nome global da relação e adicionamos um campo *id-réplica*; chamamos a combinação de **nome global da réplica**.

22.9.2 Estrutura do Catálogo

Um catálogo de sistema centralizado pode ser usado, mas é vulnerável à falha do site que o contém. Uma alternativa é manter uma cópia do catálogo de sistema global, a qual descreve todos os dados de cada site. Embora essa estratégia não seja vulnerável à falha de um único site, ela compromete a sua autonomia, exatamente como a primeira solução, pois cada alteração em um catálogo local agora deve ser divulgada para todos os sites.

Uma estratégia melhor, que preserva a autonomia local e não é vulnerável à falha de um único site, foi desenvolvida no projeto de banco de dados distribuído R*, sucessor do projeto System R da IBM. Cada lugar mantém um catálogo local descrevendo todas as cópias dos dados armazenados nesse site. Além disso, o catálogo no site de nascimento de uma relação é responsável por monitorar onde são armazenadas as réplicas da relação (em geral, de fragmentos da relação). Em particular, uma descrição precisa do conteúdo de cada réplica — uma lista de colunas de um fragmento vertical ou uma condição de seleção para um fragmento horizontal — é armazenada no catálogo do site de nascimento. Quando uma nova réplica é criada ou quando uma réplica é movida entre sites, as informações do catálogo do site de nascimento da relação devem ser atualizadas.

Para se localizar uma relação, o catálogo de seu site de nascimento deve ser pesquisado. As informações desse catálogo podem ser colocadas em cache para acesso mais rápido, mas elas podem se tornar desatualizadas se, por exemplo, um fragmento for movido. Vamos descobrir que as informações colocadas na cache local estão desatualizadas quando as usarmos para acessar a relação e, nesse ponto, devemos atualizar a cache pesquisando o site de nascimento da relação. (O site de nascimento de uma relação é gravado em cada cache local, que descreve a relação e nunca muda, mesmo que a relação seja movida.)

22.9.3 Independência de Dados Distribuídos

Independência de dados distribuídos implica que os usuários devem ser capazes de escrever consultas sem considerar como uma relação está fragmentada ou replicada; é responsabilidade do SGBD computar a relação, conforme necessário (localizando cópias convenientes dos fragmentos, juntando fragmentos verticais e pegando a união dos fragmentos horizontais).

Em particular, essa propriedade implica que os usuários não devem ter de especificar o nome completo dos objetos de dados acessados, durante a avaliação de uma consulta. Vamos ver como os usuários podem acessar relações sem considerar como elas estão distribuídas. O *nome local* de uma relação no catálogo do sistema (Seção 22.9.1) é, na realidade, a combinação de um *nome de usuário* e um *nome de relação* definido pelo usuário. Os usuários podem dar os nomes que desejarem às suas relações, sem considerar as relações criadas por outros usuários. Quando o usuário escreve um programa ou instrução SQL que se refere a uma relação, ele simplesmente usa o nome da relação. O SGBD adiciona o nome de usuário no nome da relação para obter um nome local e, então, adiciona o id do site do usuário como site de nascimento (padrão), para obter um nome global da relação global. Pesquisando o nome global da relação — no catálogo local, se foi colocado na cachê, ou no catálogo do site de nascimento —, o SGBD pode localizar réplicas da relação.

Talvez um usuário queira criar objetos em vários sites ou se referir a relações criadas por outros usuários. Para fazer isso, ele pode criar um **sinônimo** para um nome global da relação usando um comando do estilo SQL (embora atualmente tal comando não faça parte do padrão SQL:1999) e, subseqüentemente, referir-se à relação usando o sinônimo. Para cada usuário conhecido em um site, o SGBD mantém uma tabela de sinônimos como parte do catálogo do sistema nesse site e utiliza essa tabela para localizar o nome global da relação. Note que um programa de usuário é executado sem alteração, mesmo que réplicas da relação sejam movidas, pois o nome de relação global nunca é alterado até que a relação em si seja destruída.

Talvez os usuários queiram executar consultas em réplicas específicas, especialmente se for usada replicação assíncrona. Para suportar isso, o mecanismo do sinônimo

pode ser adaptado para também permitir que os usuários criem sinônimos para nomes globais de réplica.

22.10 PROCESSAMENTO DE CONSULTA DISTRIBUÍDA

Discutiremos primeiramente, por meio de exemplos, os problemas envolvidos na avaliação de operações da álgebra relacional em um banco de dados distribuído e, depois, destacaremos a otimização de consulta distribuída. Considere as duas relações a seguir:

Marinheiros(*id-marin:* `integer`, *nome-marin:* string, *avaliação:* integer, *idade:* real)
Reservas(*id-marin:* `integer`, *id-barco:* `integer`, *dia:* `date`, *nome-resp:* string)

Assim como no Capítulo 14, suponha que cada tupla de Reservas tenha 40 bytes de comprimento, que uma página pode conter 100 tuplas de Reservas e que temos 1000 páginas dessas tuplas. Analogamente, suponha que cada tupla de Marinheiros tenha 50 bytes de comprimento, que uma página pode conter 80 tuplas de Marinheiros e que temos 500 páginas dessas tuplas.

Para estimarmos o custo de uma estratégia de avaliação, além de contarmos o número de E/S de página, precisamos contar o número de páginas enviadas de um site para outro, pois os custos de comunicação representam um componente significativo do custo global em um banco de dados distribuído. Também devemos alterar nosso modelo de custo para levar em conta o custo do envio das tuplas resultantes para o site onde a consulta é feita, a partir do site onde o resultado é montado! Neste capítulo, denotamos o tempo gasto para ler uma página do disco (ou para gravar uma página no disco) como t_d e o tempo gasto para enviar uma página (de qualquer site para outro site) como t_s.

22.10.1 Consultas sem Junção em um SGBD Distribuído

Mesmo operações simples, como percorrer uma relação, seleção e projeção, são afetadas pela fragmentação e pela replicação. Considere a consulta a seguir:

```
SELECT  M.idade
FROM    Marinheiros M
WHERE   M.avaliação > 3 AND M.avaliação < 7
```

Suponha que a relação Marinheiros seja fragmentada horizontalmente, com todas as tuplas tendo um valor de avaliação menor do que 5 em Xangai e todas as tuplas tendo um valor de avaliação maior do que 5 em Tóquio.

O SGBD deve responder a esta consulta avaliando-a nos dois sites e pegando a união das respostas. Se a cláusula `SELECT` contivesse AVG (*M.idade*), a combinação das respostas não poderia ser feita simplesmente pegando-se a união — o SGBD precisaria calcular a soma e a contagem de valores de *idade* nos dois sites e usar essa informação para calcular a idade média de todos os marinheiros.

Por outro lado, se a cláusula `WHERE` contivesse apenas a condição $M.avaliação > 6$, o SGBD deveria reconhecer que essa consulta poderia ser respondida apenas executando-a em Tóquio.

Como outro exemplo, suponha que a relação Marinheiros fosse fragmentada verticalmente, com os campos *id-marin* e *avaliação* em Xangai e os campos *nome-marin* e *idade* em Tóquio. Nenhum campo seria armazenado nos dois sites. Portanto, essa fragmentação vertical seria uma decomposição com perda, exceto pelo fato de um campo contendo o id da tupla de Marinheiros correspondente ser incluído pelo SGBD nos

dois fragmentos! Agora, o SGBD precisa reconstruir a relação Marinheiros juntando os dois fragmentos no campo de id de tupla comum e executando a consulta nessa relação reconstruída.

Finalmente, suponha que a relação Marinheiros inteira fosse armazenada em Xangai e Tóquio. Poderíamos responder a qualquer uma das consultas anteriores executando-a em Xangai ou Tóquio. Onde a consulta deve ser executada? Isso depende do custo do envio da resposta para o site da consulta (que pode ser Xangai, Tóquio ou algum outro site), assim como do custo da execução da consulta em Xangai e Tóquio — os custos do processamento local podem diferir, dependendo do índices que estejam disponíveis em Marinheiros nos dois sites, por exemplo.

22.10.2 Junções em um SGBD Distribuído

As junções de relações em site diferentes podem ser muito dispendiosas e consideraremos agora as opções de avaliação que devem ser levadas em conta em um ambiente distribuído. Suponha que a relação Marinheiros estivesse armazenada em Londres e que a relação Reservas estivesse armazenada em Paris. Consideraremos o custo de várias estratégias para calcular $Marinheiros \bowtie Reservas$.

Busca Quando Necessário

Poderíamos fazer uma junção de loops aninhados orientada a página em Londres, com Marinheiros como a relação externa e, para cada página de Marinheiros, buscar todas as páginas de Reservas de Paris. Se colocarmos na cache, em Londres, as páginas buscadas de Reservas até que a junção esteja completa, as páginas serão buscadas apenas uma vez; mas suponha que as páginas de Reservas não sejam colocadas na cache, apenas para ver como as coisas podem ficar complicadas. (A situação pode ficar muito pior se usarmos uma junção de loops aninhados orientada a tupla!)

O custo é de $500t_d$ para percorrer Marinheiros, mais (para cada página de Marinheiros) o custo de percorrer e enviar todas as tuplas de Reservas, que é de $1000(t_d + t_s)$. Portanto, o custo total é de $500t_d + 500.000(t_d + t_s)$.

Além disso, se a consulta não foi submetida em Londres, devemos adicionar o custo do envio do resultado para o site da consulta; esse custo vai depender do tamanho do resultado. Como *id-marin* é uma chave para Marinheiros, o número de tuplas no resultado é de 100.000 (o número de tuplas em Reservas) e cada tupla tem 40 + 50 = 90 bytes de comprimento; assim, 4000/90 = 44 tuplas resultantes cabem em uma página e o tamanho do resultado é de 100.000/44=2273 páginas. O custo do envio da resposta para outro site, se necessário, é de 2273 t_s. No restante desta seção, supomos que a consulta é feita no site onde o resultado é calculado; se não for, o custo do envio do resultado para o local da consulta deverá ser somado ao custo.

Neste exemplo, observe que, se o site de consulta for não for Londres nem Paris, o custo do envio do resultado será maior do que o custo do envio de Marinheiros e de Reservas para o site da consulta! Portanto, seria mais barato enviar as duas relações para o site da consulta e calcular lá a junção.

Como alternativa, poderíamos fazer uma junção de loops aninhados indexados em Londres buscando todas as tuplas de Reservas correspondentes para cada tupla de Marinheiros. Suponha que tenhamos um índice de hashing não agrupado na coluna *sid* de Reservas. Como existem 100.000 tuplas de Reservas e 40.000 tuplas de Marinheiros, cada marinheiro tem 2,5 reservas em média. O custo para localizar as 2,5 tuplas de Reservas que correspondem a determinada tupla de Marinheiros é de $(1{,}2 + 2{,}5)t_d$, supondo 1,2 E/S para localizar o bucket apropriado no índice. O custo total é o custo

da varredura de Marinheiros, mais o custo para localizar e buscar as tuplas de Reservas correspondentes para cada tupla de Marinheiros, $500t_d + 40.000(3{,}7t_d + 2{,}5t_s)$.

Os dois algoritmos buscam as tuplas de Reservas exigidas a partir de um site remoto, conforme for necessário. Claramente, essa não é uma boa idéia; o custo do envio de tuplas domina o custo total, mesmo para uma rede rápida.

Envio para um Site

Podemos enviar Marinheiros de Londres para Paris e realizar lá a junção, enviar Reservas para Londres e executar lá a junção ou enviar ambas para o site onde a consulta foi feita e lá calcular a junção. Observe, novamente, que a consulta poderia ter sido feita em Londres, Paris ou, talvez, em um terceiro site, digamos, Timbuktu!

O custo para varrer e enviar Marinheiros, salvá-la em Paris e, depois, fazer a junção em Paris é de $500(2t_d + t_s) + 4500t_d$, supondo que seja usada a versão da junção sort-merge descrita na Seção 14.4.2, e que temos um número adequado de páginas de buffer. No restante desta seção, supomos que a junção sort-merge é o método usado quando as duas relações estão no mesmo site.

O custo para enviar Reservas e fazer a junção em Londres é de $1000(2t_d + t_s) + 4500t_d$.

Semijunções e Bloomjoins

Considere a estratégia de enviar Reservas para Londres e calcular lá a junção. Algumas tuplas em (na instância corrente de) Reservas não se juntam com nenhuma tupla de (na instância corrente de) Marinheiros. Se pudéssemos identificar de algum modo as tuplas de Reservas que, com certeza, não se juntam com nenhuma tupla de Marinheiros, poderíamos evitar seu envio.

Duas técnicas, *Semijunção* e *Bloomjoin*, foram propostas para reduzir o número de tuplas de Reservas a ser enviadas. A primeira delas é chamada de **Semijunção**. A idéia é proceder em três etapas:

1. Em Londres, calcular a projeção de Marinheiros nas colunas de junção (neste caso, apenas o campo *id-marin*) e enviar essa projeção para Paris.
2. Em Paris, calcular a junção natural da projeção recebida do primeiro site com a relação Reservas. O resultado dessa junção é chamado de **redução** de Reservas com relação a Marinheiros. Claramente, apenas as tuplas de Reservas que estão na redução serão juntadas com as tuplas da relação Marinheiros. Portanto, enviamos a redução de Reservas para Londres, em vez da Reservas inteira.
3. Em Londres, calcular a junção da redução de Reservas com Marinheiros.

Vamos calcular o custo do uso dessa técnica para nosso exemplo de consulta de junção. Suponha que tenhamos uma implementação simples e direta de projeção, baseada primeiro na varredura de Marinheiros e criemos uma relação temporária com tuplas que têm apenas um campo *id-marin*, depois ordenamos a relação temporária e percorramos essa relação temporária ordenada para eliminar duplicatas. Se supusermos que o tamanho do campo *id-marin* é de 10 bytes, o custo da projeção será de $500t_d$ para percorrer Marinheiros, mais $100t_d$ para criar a relação temporária, $400t_d$ para ordená-la (em duas passagens), $100t_d$ para a varredura final, $100t_d$ para gravar o resultado em outra relação temporária, dando um total de $1200t_d$. [Como *sid* é uma chave, nenhuma duplicata precisa ser eliminada; se o otimizador for suficientemente bom para reconhecer isso, o custo da projeção será de apenas $(500 + 100)t_d$.]

Portanto, o custo do cálculo da projeção e do seu envio para Paris é de $1200t_d + 100t_s$. O custo do cálculo da redução de Reservas é de $3 \cdot (100 + 1000) = 3300t_d$, supondo que seja usada uma junção sort-merge. (O custo não reflete o fato de Marinheiros já ser ordenada; o custo diminuiria ligeiramente, se a junção sort-merge refinada explorasse isso.)

Qual é o tamanho da redução? Se todo marinheiro mantiver pelo menos uma reserva, a redução incluirá cada tupla de Reservas! O esforço investido no envio da projeção e na redução de Reservas é um desperdício total. Na verdade, por causa dessa observação, notamos que a Semijunção é particularmente útil em conjunto com uma seleção em uma das relações. Por exemplo, se quisséssemos calcular a junção das tuplas de Marinheiros com um valor de *rating* maior do que 8 com a relação Reservas, o tamanho da projeção em *sid* para as tuplas que satisfazem a seleção seria de apenas 20% da projeção original, ou seja, 20 páginas.

Vamos agora continuar o exemplo de junção supondo que temos a seleção adicional em *rating*. (O custo do cálculo da projeção de Marinheiros cai um pouco, o custo do seu envio cai para $20t_s$ e o custo da redução de Reservas também cai um pouco, mas ignoraremos essas reduções, por simplicidade.)

Supomos que apenas 20% das tuplas de Reservas são incluídas na redução, graças à seleção. Assim, a redução contém 200 páginas e o custo de seu envio é de $200t_s$.

Finalmente, em Londres, a redução de Reservas é juntada com Marinheiros, a um custo de $3 \cdot (200 + 500) = 2100t_d$. Observe que, usando essa técnica, são mais de 6500 E/S de página *versus* cerca de 200 páginas enviadas. Em contraste, enviar Reservas para Londres e fazer a junção lá custa $1000t_s$ mais $4500t_d$. Com uma rede de alta velocidade, o custo da Semijunção pode ser maior do que o custo do envio de Reservas em sua totalidade, mesmo que o custo do envio seja menor ($200t_s$ *versus* $1000t_s$).

A segunda técnica, chamada de **Bloomjoin**, é muito parecida. A principal diferença é que um vetor de bits é enviado na primeira etapa, em vez da projeção de Marinheiros. Um vetor de bits de tamanho (algum escolhido) k é calculado por meio do hashing de cada tupla de Marinheiros no intervalo de 0 a $k - 1$, e configurando o bit i como 1 se alguma tupla for mapeada por hashing em i, e 0, caso contrário. Na segunda etapa, a redução de Reservas é calculada por meio do hashing de cada tupla de Reservas (usando o campo *sid*) no intervalo de 0 a $k - 1$, usando a mesma função de hashing utilizada para construir o vetor de bits, e descartando as tuplas cujo valor de hashing i corresponder a um bit 0. Como nenhuma das tuplas de Marinheiros é mapeada por hashing para tal i, nenhuma delas poderia se juntar com alguma tupla de Reservas que não esteja na redução.

O custo da redução de Reservas é apenas o custo de sua varredura, ou seja, $1000t_d$. O tamanho da redução de Reservas provavelmente será quase igual ou um pouco maior do que o tamanho da redução na estratégia da Semijunção; em vez de 200, supomos que esse tamanho é de 220 páginas. (Supomos que a seleção em Marinheiros é incluída, para permitir uma comparação direta com o custo da Semijunção.) Portanto, o custo do envio da redução é de $220t_s$. O custo da junção final em Londres é de $3 \cdot (500 + 220) = 2160t_d$.

Assim, na comparação com a Semijunção, o custo do envio dessa estratégia é praticamente o mesmo, embora pudesse ser mais alto se o vetor de bits não fosse tão seletivo quanto a projeção de Marinheiros, em termos da redução de Reservas. Normalmente, contudo, a redução de Reservas não é maior do que 10% a 20% do que o tamanho da redução na Semijunção. Em troca por esse custo de envio ligeiramente mais alto, a Bloomjoin atinge um custo de processamento significativamente menor: menos de $3700t_d$ *versus* mais de $6500t_d$ para a Semijunção. Na verdade, a Bloomjoin tem um custo de E/S menor e um custo de envio menor do que a estratégia de envio

de tudo de Reservas para Londres! Esses números indicam por que a Bloomjoin é um método de junção distribuída atraente; mas a sensibilidade do método quanto à eficácia do hashing do vetor de bits (na redução de Reservas) deve ser lembrada.

22.10.3 Otimização de Consulta Baseada em Custo

Vimos como a distribuição dos dados pode afetar a implementação de operações individuais como seleção, projeção, agregação e junção. Em geral, é claro, uma consulta envolve várias operações, e a otimização de consultas em um banco de dados distribuído apresenta os seguintes desafios adicionais:

- Os custos de comunicação devem ser considerados. Se temos várias cópias de uma relação, também precisamos decidir qual cópia vamos usar.
- Se sites individuais estão sob o controle de diferentes SGBDs, a autonomia de cada site deve ser respeitada ao se fazer um planejamento de consulta global.

Basicamente, a otimização de consultas ocorre como em um SGBD centralizado, conforme descrito no Capítulo 12, com informações sobre as relações em sites remotos obtidas a partir dos catálogos de sistema. Naturalmente, existem mais métodos alternativos a considerar para cada operação (por exemplo, considerar as novas opções para junções distribuídas), e a métrica do custo também deve levar em conta os custos de comunicação, mas o processo de planejamento global fica basicamente inalterado, se consideramos a métrica do custo como o custo total de todas as operações. (Se considerássemos o tempo de resposta, o fato de que certas consultas podem ser executadas em paralelo, em diferentes sites, exigiria alterar o otimizador, como vimos na Seção 22.5.)

No plano global, a manipulação local de relações no site onde elas estão armazenadas (visando a computar uma relação intermediária para ser enviada a algum lugar) é encapsulada em um plano local *sugerido*. O plano global inclui vários desses planos locais, os quais podemos considerar como subconsultas sendo executadas em diferentes locais. Na geração do plano global, os planos locais sugeridos fornecem estimativas de custo realistas para o cálculo das relações intermediárias; os planos locais sugeridos são construídos pelo otimizador, principalmente para fornecer essas estimativas de custo local. Um site está livre para ignorar o plano local sugerido a ele, se for capaz de encontrar um plano mais barato, usando informações mais atualizadas dos catálogos locais. Assim, a autonomia do site é respeitada na otimização e na avaliação de consultas distribuídas.

22.11 ATUALIZAÇÃO DE DADOS DISTRIBUÍDOS

A visão clássica de um SGBD distribuído é a de que, do ponto de vista do usuário, ele deve se comportar exatamente como um SGBD centralizado; os problemas que surgem com a distribuição dos dados devem ser transparentes para o usuário, embora, é claro, eles devam ser tratados na implementação.

Com relação às consultas, essa visão de um SGBD distribuído significa que os usuários devem ser capazes de fazer consultas sem se preocuparem com onde e como as relações estarão armazenadas; já vimos as implicações desse requisito na avaliação de consultas.

Com relação às atualizações, essa visão significa que as transações devem continuar a ser ações atômicas, independentemente da fragmentação e da replicação de dados. Em particular, todas as cópias de uma relação modificada devem ser atualizadas antes

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.