Projeto Físico de Banco de Dados
Capítulo 20

Daniel Antonio Garcia Manzato[1], **Rodrigo Grassi Martins**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6176 – 13.084-971 – Campinas – SP – Brasil

{dmanzato,ra041420}@ic.unicamp.br

Abstract. *This article is a summary of the twentieth chapter – Physical Databse Design and Tuning – of the book "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke.*

Resumo. *Este artigo é um resumo do vigésimo capítulo – Projeto Físico de Banco de Dados – do livro "Database Management Systems" escrito por Raghu Ramakrishnan e Johannes Gehrke.*

1. Introdução ao Projeto Físico de Banco de Dados

O desempenho de um SGBD para as consultas e atualizações mais freqüentes é uma importante métrica de um projeto de banco de dados. Um DBA pode melhorar o desempenho do sistema identificando gargalos e ajustando alguns parâmetros do SGBD, ou ainda acrescentando hardware para amenizar tais gargalos. No entanto, o primeiro passo para se obter um bom resultado final é fazer boas escolhas de projeto, o que será tratado neste capítulo.

Após o projeto dos esquemas conceitual e externo, que representam uma coleção de relações e visões juntamente com um conjunto de restrições de integridade, faz-se necessário tratar as questões de desempenho através do projeto físico, o que deverá resultar em um esquema físico de banco de dados. Posteriormente, conforme os requisitos de usuário evoluem, geralmente é preciso realizar ajustes (ou *tuning*) de todos os ítens de projeto, de forma a se atingir um desempenho satisfatório.

O guia natural para a obtenção de um bom projeto é a natureza dos dados e como se pretende utilizá-los. Faz-se importante entender a carga de trabalho, ou seja, o conjunto de consultas e atualizações que serão submetidas ao banco de dados. Além disso, os usuários também têm certos requisitos de desempenho para determinadas consultas e atualizações, ou quantas transações precisam ser processadas por segundo. Dessa maneira, a descrição da carga de trabalho e os requisitos de usuário são a base para as decisões que deverão ser tomadas durante o projeto físico.

1.1. Cargas de Trabalho *(Workloads)*

A chave para se alcançar um bom projeto físico é uma descrição precisa da carga de trabalho esperada. Esta descrição inclui os ítens a seguir:

1. Uma lista das consultas (com suas freqüências).
2. Uma lista de atualizações (com suas freqüências).
3. Objetivos de desempenho para cada tipo de consulta e atualização.

Para cada consulta da carga de trabalho é preciso ainda identificar:

- Quais relações são acessadas.
- Quais atributos são retidos (na cláusula SELECT).
- Quais atributos possuem condições de junção ou seleção expressas sobre eles (na cláusula WHERE) e quão seletivas estas condições tendem a ser.

Da mesma forma, para cada atualização da carga de trabalho é necessário considerar:

- Quais atributos possuem condições de junção ou seleção expressas sobre eles (na cláusula WHERE) e quão seletivas estas condições tendem a ser.
- O tipo de atualização (INSERT, DELETE ou UPDATE) e a relação a ser atualizada.
- Para os comandos de atualização, quais campos serão modificados.

1.2. Decisões de Projeto Físico e de Ajustes *(Tuning)*

Existem decisões importantes a serem tomadas durante o projeto físico e posteriormente nos ajustes, decisões estas que incluem:

1. Escolha dos índices a serem criados:
    - Para quais relações serão criados índices e qual campo ou combinação de campos serão escolhidos como chaves de busca.
    - Para cada índice, se ele deve ser agrupado ou não.
2. Ajustando o esquema conceitual:
    - Qual a melhor forma normal a ser utilizada (por exemplo, BCNF ou 3NF).
    - Em alguns casos, pode ser interessante desnormalizar o esquema a fim de se obter um ganho de desempenho.
    - Pode-se optar também pelo particionamento vertical, ou seja, decompor ainda mais certos esquemas já decompostos.
    - Visões podem ser utilizadas para mascarar as mudanças no esquema conceitual dos usuários.
3. Ajustes de consultas e transações: Consultas e transações freqüentemente executadas podem ser reescritas para ficarem mais rápidas.

1.3. Necessidade de Ajustes

Detalhes da carga de trabalho são muito difíceis de serem detectados durante o projeto inicial. Conseqüentemente, os ajustes posteriores – baseados nas informações reais de uso do banco – são importantes para a obtenção de um bom desempenho do sistema.

2. Orientações para Seleção de Índices

A escolha de um bom conjunto de índices é uma tarefa difícil. Para uma boa decisão, faz-se necessário o conhecimento das técnicas de indexação existentes e do funcionamento do otimizador de consultas. O guia a seguir ilustra os principais pontos a serem considerados:

Necessidade do índice: A menos que exista uma consulta que possa se beneficiar do índice, a sua construção é desnecessária.

Escolha da chave de busca: Os candidatos naturais à chave de busca são os atributos que aparecem nas cláusulas WHERE, ORDER BY e GROUP BY. Quanto ao tipo de índice a ser utilizado, se a cláusula apresentar uma condição de igualdade a melhor escolha é um índice do tipo hash. Por outro lado, se ocorrerem intervalos de valores a melhor alternativa é um índice do tipo árvore B+.

Utilização de múltiplos atributos na chave de busca: Esta opção deve ser considerada quando: (1) Uma cláusula WHERE inclui condições em mais de um atributo de uma relação; (2) Pode-se chegar numa estratégia apenas de índice (o acesso à relação é evitado) para consultas importantes. Deve-se atentar para a ordem dos atributos na chave de busca, que deve ser a mesma ordem em que os atributos aparecem na cláusula WHERE.

Necessidade do agrupamento: A opção de agrupamento é muito importante, pois influi significativamente no desempenho do acesso à relação considerada. Além disso, apenas um índice por relação pode ser agrupado, o que aumenta ainda mais a importância de tal decisão. Como regra geral, consultas de intervalos tendem a se beneficiar mais do agrupamento. Tendo que decidir sobre várias consultas, cada uma envolvendo diferentes conjuntos de atributos, deve-se considerar a seletividade e a freqüência de cada uma delas.

Índice hash versus árvore: Índices do tipo árvore B+ devem ser considerados em consultas que envolvem intervalos, enquanto que índices do tipo hash nos casos de igualdade. Quando se tem junções por laços aninhados, a opção de hash é a mais adequada para a relação mais interna pois o ganho obtido é significativo (neste caso a chave de busca deve incluir as colunas de junção).

Ponderação do custo de manutenção do índice: Para cada índice criado é necessário um estudo sobre o impacto que o mesmo irá causar no sistema, principalmente com relação à sobrecarga de sua manutenção nas atualizações.

3. Exemplos Básicos de Seleção de Índices

Os exemplos a seguir ilustram como fazer a seleção de índices durante o projeto físico do banco de dados.

```
SELECT E.ename, D.mgr
FROM Employees E, Departments D
WHERE D.dname='Toy' AND E.dno=D.dno
```

As relações utilizadas na consulta acima são Employees e Departments, e as duas condições da cláusula WHERE são condições de igualdades. O guia apresentado sugere então que um índice do tipo hash seja construído nos atributos envolvidos. Não é difícil de se constatar que um índice hash em *D.dname* na relação Departments seja uma boa escolha. Mas e a igualdade *E.dno=D.dno*? Como já existe um índice em Departments e poucas tuplas satisfazem a condição *D.dname*='Toy', um índice adicional nesta relação é desnecessário. Porém, a criação de um índice hash em *E.dno* na relação Employees se faz fundamental, para que as tuplas da mesma sejam diretamente obtidas para cada par da junção realizada, considerando uma junção por laços aninhados com índice.

```
SELECT E.ename, D.dname
FROM Employees E, Departaments D
WHERE E.sal BETWEEN 10000 AND 20000
AND E.hobby='Stamps' AND E.dno=D.dno
```

Na consulta do exemplo acima, exceto a junção, as condições de seleção estão na relação Employees. Portanto, é evidente que uma boa estratégia seria utilizar Departments como a relação mais interna criando um índice hash no atributo *D.dno*. E quanto à relação Employees, qual índice deveria ser construído? Um índice do tipo árvore B+ no atributo *sal* poderia ajudar na seleção de intervalos. E um índice do tipo hash em *hobby* poderia ser útil para resolver a seleção de igualdades. Se algum destes índices estiver disponível, é possível recuperar as tuplas de Employees através dele e utilizar o índice em *D.dno* para fazer a junção e aplicar as seleções e projeções restantes *on-the-fly*. No caso dos dois índices estiverem disponíveis, então o otimizador irá optar pelo índice mais seletivo.

4. Agrupamento e Indexação

Considere novamente o exemplo:

```
SELECT E.ename, D.mgr
FROM Employees E, Departaments D
WHERE D.dname='Toy' AND E.dno=D.dno
```

Na seção anterior ficou definido que uma boa estratégia para esta consulta seria utilizar um índice em *dname* na relação Departments e realizar a junção através de um índice no campo *E.dno* de Employees. Estes índices deveriam ser agrupados? Considerando que o número de tuplas que satisfazem *D.dname*='Toy' é pequeno, não é necessário a criação de um índice agrupado neste caso. Por outro lado, Employees é a relação mais interna de um laço aninhado e *E.dno* não é uma chave candidata. Estes fatores constituem um forte argumento para que o índice em *E.dno* de Employees seja agrupado.

Considere agora:

```
SELECT E.ename, D.mgr
FROM Employees E, Departaments D
WHERE E.hobby='Stamps' AND E.dno=D.dno
```

Esta consulta difere da anterior na condição *E.hobby*='Stamps' no lugar de *D.dname*='Toy'. Considerando que o número de tuplas que satisfazem a nova condição é maior que o caso anterior, uma junção por ordenação-intercalação tende a ser mais eficiente do que uma junção por laço aninhado com índice. Uma junção por ordenação-intercalação poderia tirar vantagem de um índice árvore B+ agrupado no atributo *dno* de Departments para a obtenção das tuplas, evitando assim a ordenação da relação.

Conforme visto nos exemplos, ao se obter tuplas através de um índice o impacto do agrupamento depende do número de tuplas obtidas, ou seja, do número de tuplas que satisfazem as condições de seleção do índice. Um índice não agrupado é tão bom quanto um agrupado para uma seleção que obtém uma única tupla. Conforme o número de tuplas obtidas aumenta, o índice não agrupado se torna mais caro até mesmo do que uma busca seqüencial na relação inteira. Apesar desta última obter todas as tuplas, cada página é lida somente uma vez, enquanto que uma página pode ser lida tantas vezes quanto for o número de tuplas contidas nesta página, caso um índice não agrupado estivesse sendo utilizado.

5. Índices que Habilitam Estratégias Apenas-de-Índice

Serão mostradas nesta seção consultas para as quais se pode encontrar planos eficientes que evitam a obtenção de tuplas de uma das relações envolvidas. Ao invés disso, estes planos lêem um índice associado (que tende a ser bem menor que a relação original). Um índice que é usado apenas para esta estratégia, denominada estratégia apenas de índice, não deve ser agrupado uma vez que as tuplas da relação indexada nunca são lidas.

A seguinte consulta obtém os gerentes dos departamentos com pelo menos um funcionário:

```
SELECT D.mgr
FROM Departaments D, Employees E
WHERE D.dno=E.dno
```

Note que nenhum atributo de Employees é selecionado. Se um índice no campo *dno* de Employees existisse, poderia se ter a otimização de fazer um laço aninhado com índice usando uma estratégia apenas de índice para esta relação sendo a mais interna. Logo, a decisão correta para este exemplo seria construir um índice não agrupado no campo *dno* de Employees, ao invés de um índice agrupado.

Considere agora a seguinte variação da consulta anterior:

```
SELECT D.mgr, E.eid
FROM Departaments D, Employees E
WHERE D.dno=E.dno
```

Neste caso, para que um possível índice em *dno* de Employees pudesse ser utilizado eficientemente na junção, com Departments sendo a relação mais externa, este índice teria que ser agrupado (já que o índice não contém todas as informações que são selecionadas na consulta, demandando a leitura das tuplas de Employees). Por outro lado, se houvesse um índice árvore B+ em $<dno,eid>$, toda informação necessária à consulta estaria nele presente. Dessa forma, poder-se-ia utilizar este índice para encontrar todos os registros com um mesmo *dno*, já que estarão armazenados juntamente neste índice. (Note que devido a este fato um índice hash não seria apropriado neste caso.) Logo, esta consulta poderia ser resolvida usando-se um laço aninhado com índice com Departments sendo a relação mais externa e uma estratégia apenas de índice para Employees como sendo a relação mais interna.

6. Ferramentas de Auxílio à Seleção de Índices

O número de possíveis índices a serem construídos é muito grande: para cada relação pode-se considerar todos os possíveis subconjuntos de atributos como chaves de índice; é também possível variar a ordem dos atributos para cada índice; finalmente, pode-se agrupar ou não cada índice. Muitas aplicações de larga escala criam dezenas de milhares de relações diferentes, o que torna o ajuste manual deste tipo de esquema um grande desafio.

A dificuldade e a importância da tarefa de seleção de índices motivou o desenvolvimento de ferramentas que auxiliam os administradores de banco de dados a selecionar os índices apropriados para uma dada carga de trabalho.

6.1. Seleção Automática de Índices

Conforme mencionado anteriormente, a seleção automática de índices é um problema difícil. Por exemplo, para uma tabela com 10 atributos, existem 10 possibilidades diferentes de índices com um único atributo, 90 possibilidades diferentes de índices com dois atributos, e 30240 possibilidades diferentes de índices com cinco atributos. Para uma carga de trabalho complexa, envolvendo centenas de tabelas, o número de diferentes configurações de índices é claramente muito grande.

A eficiência das ferramentas de seleção automática de índices pode ser separada em dois fatores: (1) o número de configurações de índices consideradas; (2) o número de chamadas ao otimizador necessárias para o cálculo do custo de cada configuração analisada. Note que reduzir o espaço de busca de índices candidatos é análogo a restringir o espaço de busca do otimizador de consultas a planos apenas de profundidade à esquerda. Em muitos casos, o plano ótimo pode não estar neste espaço restrito, mas considerando todas as análises feitas pode-se chegar a um plano cujo custo é bem próximo ao do plano ótimo. Há portanto um balanço entre a eficiência que se espera da ferramenta de seleção automática de índices e o quão próximo o resultado final estará da configuração ótima de índices.

6.2. Funcionamento dos Assistentes de Ajustes de Índices

Alguns SGBDs disponibilizam tais ferramentas de auxílio nos ajustes de índices. Por exemplo, o DB2 oferece o *DB2 Index Advisor*, enquanto que o SQL Server oferece o *Microsoft SQL Server 2000 Tuning Wizard*.

Para ilustrar como estas ferramentas funcionam, um algoritmo representativo de dois passos será apresentado a seguir. No primeiro passo, um conjunto de candidatos a índices a serem considerados são selecionados. No segundo passo, utiliza-se a seleção feita no passo anterior para a enumeração das possíveis configurações de índices enumeradas.

Seleção dos Candidatos à Índice: Uma forma de se limitar o grande espaço de busca de possíveis índices é ajustar, primeiramente, cada consulta da carga de trabalho independentemente, e depois considerar uma união de todos os índices selecionados neste primeiro passo como entrada para o segundo passo do algoritmo. Para uma dada consulta, um atributo indexável é um atributo que aparece nas cláusulas WHERE, GROUP BY ou ORDER BY. Um candidato a índice para esta consulta é um índice que contém apenas atributos indexáveis. Pode-se selecionar os candidatos a índices para uma consulta individual através da enumeração de todos os índices com até k atributos indexáveis. Apesar deste método ter a desvantagem de ser bastante custoso, ele garante que o melhor índice com até k atributos estará entre as possibilidades. Existem, no entanto, algoritmos de busca heurística que são mais rápidos por serem menos exaustivos.

Enumeração das Possíveis Configurações de Índices: Nesta segunda fase do algoritmo, usar-se-á os índices candidatos levantados na primeira fase para a enumeração das diferentes configurações de índices. Pode-se novamente optar por uma abordagem exaustiva, desta vez enumerando todas as configurações de até k índices candidatos. No entanto, como na fase anterior, estratégias de busca mais sofisticadas existem para restringir o número de configurações consideradas, ainda que gerando uma configuração final de boa qualidade.

7. Ajustes em Banco de Dados

Após a fase inicial do projeto de banco de dados o funcionamento do sistema deverá gerar valiosas informações que poderão ser utilizadas para refinar o projeto original. Muitas das hipóteses iniciais acerca da carga de trabalho estimada poderão ser substituídas pelos padrões de uso reais observados. Ao passo que algumas das especificações iniciais serão validadas, outras se mostrarão incorretas. Predições iniciais sobre o tamanho dos dados poderão ser substituídas pelas estatísticas reais dos catálogos do sistema (apesar destas informações estarem sempre se modificando conforme o sistema evolui). A monitoração cuidadosa das consultas poderá ainda revelar problemas inesperados, como a não utilização de certos índices criados para a execução de certos planos eficientes.

Dessa forma, os contínuos ajustes no banco de dados são essenciais para a obtenção do melhor desempenho possível. Três tipos de ajustes serão aqui discutidos, a seguir.

7.1. Ajustes de Índices

Quando se constata que algumas consultas e atualizações não são tão importantes como se pensou inicialmente ou, de outra forma, são mais importantes do que se pensou inicialmente, faz-se necessário uma revisão na atual configuração de índices, o que pode acarretar na remoção de índices já existentes ou na criação de novos índices. Deve-se considerar para isso os mesmos fatores que foram analisados no projeto inicial do banco de dados, só que dessa vez levando-se em conta a nova carga de trabalho identificada.

Pode-se também descobrir que o otimizador do sistema não está encontrando alguns dos planos que se pensou. Considere, por exemplo, a seguinte consulta:

```
SELECT D.mgr
FROM Employees E, Departments D
WHERE D.dname='Toy' AND E.dno=D.dno
```

Um bom plano para este caso seria a utilização de um índice em *dname*, para se obter as tuplas de departamentos com *dname*='Toy', e o uso de um índice no campo *dno* da relação Employees, como sendo a relação mais interna da junção. Dessa maneira, uma estratégia apenas de índice poderia ser encontrada pelo otimizador. Prevendo este plano, um índice não agrupado no campo *dno* de Employees poderia ter sido criado. Não obstante, suponha que o otimizador não identificou esta estratégia apenas de índice e, conseqüentemente, as tuplas de Employees estão sendo integralmente lidas, demorando muito mais tempo para a execução da consulta. Neste caso, considerando tal limitação do sistema, o índice não agrupado no campo *dno* de Employees poderia ser removido, dando lugar a um outro índice, dessa vez agrupado.

Um outro procedimento que merece atenção é a reorganização periódica de alguns índices. Um índice estático, como o ISAM, poderia ter desenvolvido longas cadeias de *overflow*, enquanto que um índice dinâmico, como o árvore B+, poderia estar ocupando espaço adicional desnecessário, se a implementação não limpar as páginas nas exclusões. Em ambos os casos, o acesso através destes índices pode se tornar muito lento e a recriação dos mesmos poderia melhorar bastante o problema. No entanto, deve-se verificar se a indisponibilidade temporária da relação – ocasionada enquanto seus índices estiverem sendo recriados – não é um problema.

Finalmente, deve-se lembrar que o otimizador conta com as estatísticas mantidas nos catálogos do sistema, as quais são atualizadas somente quando um utilitário especial for executado. Logo, faz-se necessário garantir esta execução periodicamente para que estas estatísticas estejam sempre suficientemente atualizadas.

7.2. Ajustes do Esquema Conceitual

Durante o projeto de um banco de dados, pode-se chegar à conclusão de que os esquemas relacionais escolhidos até então não permitem que o sistema alcance as restrições de desempenho para a carga de trabalho considerada, independentemente de qualquer configuração de índices utilizada. Neste caso, faz-se necessário um reprojeto do esquema conceitual, analisando as decisões de projeto físico que serão afetadas por estas modificações.

A necessidade de reprojeto pode ser identificada antes ou depois do sistema ter entrado em produção. No segundo caso, deve-se atentar para a necessidade de mapeamento do conteúdo das relações que já estão populadas com tuplas, o que demanda um esforço adicional considerável. Também neste caso, costuma-se denominar o reprojeto de *evolução de esquema*. De qualquer maneira, os principais fatores a serem considerados no reprojeto são novamente as consultas e atualizações da carga de trabalho e as questões de redundância, que motivam as normalizações.

Os seguintes esquemas serão utilizados nos exemplos desta seção:

Contratos(*cid:* integer, *supplierid:* integer, *projectid:* integer, *deptid:* integer, *partid:* integer, *qty:* integer, *value:* real)
Departamentos(*did:* integer, *budget:* real, *annualreport:* varchar)
Ítens(*pid:* integer, *cost:* integer)
Projetos(*jid:* integer, *mgr:* char(20))
Fornecedores(*sid:* integer, *address:* char(50))

Usar-se-á a convenção comum de se denotar os atributos por um único caracter e as relações por uma seqüência de caracteres. Dessa forma, considere o esquema para a relação Contratos, chamada de CSJDPQV. O significado de uma tupla nesta relação é que o contrato com *cid* C é um acordo que o fornecedor S (com *sid* igual a *supplierid*) fornecerá Q unidades do ítem P (com *pid* igual a *partid*) ao projeto J (com *jid* igual a *projectid*) associado ao departamento D (com *deptid* igual a *did*), e que o valor V deste contrato será igual a *value*.

Existem duas restrições de integridade relacionadas a Contratos, a saber. Um projeto compra um dado ítem usando um único contrato; logo, não podem haver dois contratos distintos nos quais o mesmo projeto compra o mesmo ítem. Esta restrição pode ser representada usando a dependência funcional $JP \rightarrow C$. Além disso, um departamento compra no máximo um ítem de um dado fornecedor. Esta restrição pode ser representada usando a dependência funcional $SD \rightarrow P$. Sabe-se também que o identificador de contato C é uma chave.

O significado das demais relações é trivial, não demandando maiores explicações, principalmente porque o foco maior no exemplos seguintes será dado à relação Contratos.

A seguir serão apresentadas as opções a serem consideradas nos ajustes do esquema conceitual.

7.2.1. Optando por uma Forma Normal mais Fraca

Considere a relação Contratos. As chaves candidatas para esta relação são C (definida) e JP (que funcionalmente determina C). A única dependência não chave é $SD \rightarrow P$, sendo que P é um atributo *primo*, uma vez que faz parte da chave candidata JP. Logo, a relação Contratos não está na forma normal BCNF – pois há uma dependência não chave –, mas está na terceira forma normal.

Usando a dependência $SD \rightarrow P$ para orientar uma possível decomposição, obtém-se as relações SDP e CSJDQV. Esta decomposição é sem perdas, mas não é preservadora de dependências. Adicionando agora a relação CJP, pode-se obter uma decomposição em BCNF que é ao mesmo tempo sem perdas e preservadora de dependências. Considerando que uma decomposição em BCNF é interessante, poderia-se substituir a relação Contratos pelas três relações CJP, SDP, CSJDQV.

Suponha agora que a seguinte consulta seja freqüente: encontrar o número de cópias Q do item P ordenadas pelo contrato C. Esta consulta demanda uma junção das relações decompostas CJP e CSJDQV (ou SDP e CSJDQV), enquanto que poderia ter sido respondida diretamente usando a relação Contratos apenas. O custo adicional para se realizar esta consulta pode ser um fator persuasivo para se permanecer na terceira forma normal, ao invés de se chegar na forma normal BCNF.

7.2.2. Desnormalização

Da mesma maneira que se optou por uma forma normal mais fraca no caso anterior, visando melhorar o desempenho do sistema mediante a carga de trabalho considerada, poderia-se seguir na mesma linha com um passo ainda mais extremo: deliberadamente introduzir alguma redundância. Suponha que uma consulta freqüente seja checar se o valor de um contrato é menor do que o orçamento do departamento contratante. Uma opção seria adicionar um campo orçamento B à tabela Contratos. Dado que *did* é uma chave para Departamentos, teria-se agora a dependência $D \rightarrow B$ em Contratos, implicando nesta relação não mais estar na terceira forma normal. Entretanto, se a consulta em questão for suficientemente importante na carga de trabalho, pode-se optar por este esquema mesmo assim. Este tipo de decisão é subjetiva e deve levar em conta o custo de se ter redundância.

7.2.3. Escolha de Decomposições

Considere novamente a relação Contratos. Poder-se-ia deixá-la na terceira forma normal, aceitando a redundância relacionada ao fato dela não estar na forma normal BCNF, ou decompô-la nesta última, resolvendo o problema da redundância. Neste segundo caso, dois métodos poderiam ser empregados:

- Decompô-la nas três relações SDP, CSJDQV e CJP, conforme descrito anteriormente; neste caso, a terceira relação CJP tem como propósito único preservar a dependência $JP \rightarrow C$.

- Utilizar apenas as duas primeiras relações SDP e CSJDQV, sem adicionar a relação CJP, mesmo que esta decomposição não seja preservadora de dependências.

Note que a segunda opção não impede a checagem da restrição $JP \rightarrow C$, apenas a torna mais custosa. Uma *assertion* poderia ser criada para este fim:

```
CREATE ASSERTION checkDep
CHECK ( NOT EXISTS
( SELECT *
FROM SDP PI, CSJDQV CI
WHERE PI.supplierid=CI.supplierid
AND PI.deptid=CI.depit
GROUP BY CI.projectid, PI.partid
HAVING COUNT(cid) > 1 ) )
```

Enquanto esta *assertion* é custosa – pois envolve uma junção seguida de ordenação (para fazer o agrupamento) –, o sistema poderia simplesmente checar se JP é uma chave primária da tabela CJP, mantendo um índice nela. Esta diferença de custo na checagem da integridade dos dados é a motivação para a preservação de dependências nas decomposições. Por outro lado, se as atualizações não são tão freqüentes, este custo adicional poderia ser aceitável, ou seja, poderia se optar por não manter a tabela CJP (e, conseqüentemente, o seu índice).

7.2.4. Decomposição Vertical de Relações BCNF

Suponha agora que se tenha optado por decompor Contratos em SDP e CSJDQV. Como este esquema já está na forma normal BCNF, não há nenhum motivo, do ponto de vista de normalização, para decompô-lo ainda mais. Não obstante, considere que as seguintes consultas são freqüentes:

- Encontrar os contratos do fornecedor S.
- Encontrar os contratos realizados pelo departamento D.

Estas consultas poderiam levar à decisão de se decompor CSJDQV em CS, CD e CJQV. Esta decomposição é sem perdas e possibilita a execução das duas consultas através de relações bem menores. Uma outra razão para se considerar este tipo de decomposição é a diminuição de *hot spots* em um ambiente onde há concorrência. Sendo estas consultas freqüentes e a maioria das atualizações envolvendo alterações na quantidade de produtos e valores relacionados aos contratos, esta decomposição melhoraria o desempenho através da redução das contenções de *locks*. Ou seja, os *locks* exclusivos se concentrariam em sua maioria na relação CJQV, não conflitando com as leituras em CS e CD.

Em suma, ao se decompor uma relação faz-se necessário considerar quais consultas serão afetadas, especialmente se a única motivação para tal decomposição for a melhoria de desempenho. Por exemplo, se outra consulta importante fosse encontrar o valor total de contratos de um fornecedor, uma junção das relações CS e CJQV seria necessária, o que poderia contra-indicar esta decomposição.

7.2.5. Decomposição Horizontal

As decomposições consideradas até agora foram as chamadas decomposições verticais, onde a relação original é substituída por um conjunto de relações que são projeções da primeira. Algumas vezes, no entanto, faz-se necessário considerar um outro tipo de decomposição, baseada em seleções de tuplas da relação original, onde as relações resultantes apresentam os mesmos atributos da original. Este tipo de decomposição é chamado de decomposição horizontal.

De maneira intuitiva, esta técnica pode ser útil quando diferentes subconjuntos de tuplas são consultados em ocasiões distintas. Por exemplo, considere que diferentes regras sejam empregadas nos contratos grandes, definidos como sendo aqueles com valores maiores que 10000. (Pode-se pensar, neste caso, como ilustração, que tais contratos sejam aplicáveis a uma política de recompensa.) Esta diferenciação resultaria num número de consultas à relação Contratos com condições na forma $value > 10000$.

Uma forma de se lidar com esta situação seria através da construção de um índice árvore B+ agrupado no campo *value* de Contratos. Alternativamente, poder-se-ia substituir a relação Contratos por outras duas chamadas ContratosGrandes e ContratosPequenos, ou seja, realizando a decomposição horizontal. Note que se a seleção mencionada for a única motivação para a criação do índice, no primeiro caso, a decomposição horizontal tem a vantagem de oferecer todos os benefícios do índice sem a sobrecarga de gerenciamento do mesmo. Esta alternativa se torna ainda mais atraente se outras consultas importantes à relação Contratos demandarem índices agrupados em outros campos, além de *value*.

A decomposição horizontal pode, até certo ponto, ser mascarada através da criação de uma visão. Para o exemplo acima, a visão Contratos poderia ser criada da seguinte forma:

```
CREATE VIEW Contratos(cid, supplierid, projectid, deptid,
partid, qty, value)
AS ((SELECT *
FROM ContratosGrandes)
UNION
(SELECT *
FROM ContratosPequenos))
```

O motivo desta mascaragem poder ser transparente somente até certo ponto é o seguinte: qualquer consulta que lide unicamente com contratos grandes deve utilizar diretamente a relação ContratosGrandes e não a visão Contratos. Expressar este tipo de consulta utilizando a visão com a seleção $value > 10000$ é equivalente a utilizar diretamente ContratosGrandes, porém menos eficiente. Ou seja, apesar de ser possível a mascaragem da alteração no esquema conceitual através da criação de visões, os usuários preocupados com desempenho deverão estar cientes da mudança.

7.3. Ajustes de Consultas e Visões

Ao se notar que uma consulta está sendo executada muito mais lentamente do que o esperado, deve-se analisá-la com atenção. Em geral, pode-se resolver o problema simplesmente reescrevendo esta consulta, possivelmente com algum ajuste de índice também.

Este tipo de análise também é válido para visões que apresentam consultas lentas. Como as consultas que definem uma determinada visão podem ser tratadas independentemente, ou seja, ajustadas por si próprias, a discussão de ajustes de consultas desta seção também será válida para o ajuste de visões.

Deve-se, primeiramente, entender o plano que está sendo utilizado pelo otimizador para as consultas sob análise. Para isso, pode-se utilizar as ferramentas que normalmente são disponibilizadas pelos sistemas SGBD. Após este entendimento, pode-se encontrar maneiras de melhorar o desempenho tendo em mente o novo plano que se deseja que o otimizador passe a considerar; por exemplo, através da criação de diferentes configurações de índices ou até mesmo co-agrupando duas relações para consultas de junção.

Algumas vezes, no entanto, o otimizador não consegue encontrar o plano previsto para uma determinada consulta. Isto pode acontecer por várias razões, entre elas:

- Condições de seleção envolvendo valores nulos.
- Condições de seleção que envolvem expressões aritméticas ou de *strings*, ou expressões que utilizam o conectivo OR.
- Inabilidade de reconhecer um plano sofisticado, como por exemplo uma estratégia apenas de índice para uma consulta agregada que envolve a cláusula GROUP BY.

Dessa maneira, antes de se modificar a atual configuração de índices, deve-se tentar reescrever as consultas para melhor utilizar os índices já existentes, possivelmente evitando estas possibilidades. Por exemplo, considere a seguinte consulta que apresenta o conectivo OR:

```
SELECT E.dno
FROM Employees E
WHERE E.hobby='Stamps' OR E.age=10
```

Mesmo havendo índices tanto em *hobby* quanto em *age* um otimizador poderia falhar em reconhecer a melhor estratégia caso considerasse, por exemplo, as condições da cláusula WHERE acima como um bloco único que não casasse com nenhum dos dois índices. Neste caso, uma busca seqüencial seria feita em Employees, aplicando as seleções *on-the-fly*. Este problema poderia ser resolvido através da reescrita da consulta acima, de forma que a mesma fosse uma união de duas consultas, uma com a cláusula WHERE *E.hobby*='Stamps'e a outra com a cláusula WHERE *E.age*=10.

Um outro ponto a se considerar é a reescrita de consultas para evitar o uso de operações caras, como DISTINCT, GROUP BY e HAVING. O uso de DISTINCT na cláusula SELECT acarreta na duplicação do processo de eliminação, o que pode ser bastante custoso. Pode-se evitar o uso desta operação em muitas situações como, por exemplo, em consultas a relações simples onde dados duplicados não fazem diferença ou há uma chave candidata da relação entre os atributos mencionados na cláusula SELECT. Quanto a GROUP BY e HAVING, algumas vezes a consulta pode ser substituída por outra sem estas cláusulas, eliminando desta forma a operação de ordenação demandada. Considere, por exemplo, a seguinte consulta:

```
SELECT MIN (E.age)
FROM Employees E
GROUP BY E.dno
HAVING E.dno=102
```

Esta consulta é equivalente a:

```
SELECT MIN (E.age)
FROM Employees E
WHERE E.dno=102
```

Uma outra orientação é a respeito de consultas complexas, que em geral são escritas em passos, normalmente utilizando relações temporárias. Freqüentemente elas podem ser reescritas sem a utilização da relação temporária, o que as tornam mais rápidas. Considere a seguinte consulta para o cálculo da média de salário dos departamentos gerenciados por 'Robinson':

```
SELECT *
INTO Temp
FROM Employees E, Departments D
WHERE E.dno=D.dno AND D.mgrname='Robinson'

SELECT T.dno, AVG(T.sal)
FROM Temp T
GROUP BY T.dno
```

Esta consulta pode ser reescrita da seguinte forma:

```
SELECT E.dno, AVG(E.sal)
FROM Employees E, Departments D
WHERE E.dno=D.dno AND D.mgrname='Robinson'
GROUP BY E.dno
```

Como a segunda consulta não materializa a relação intermediária Temp, ela tende a ser mais rápida. Além disso, o otimizador pode até encontrar um plano mais eficiente, como uma estratégia apenas de índice que nunca obtém tuplas de Employees, caso haja um índice árvore B+ composto em *<dno,sal>*. Ou seja, a reescrita de consultas para evitar a utilização desnecessária de relações temporárias não somente evita a materialização das mesmas, como também abre um leque maior de possibilidades de otimização para o otimizador explorar. Não obstante, em algumas situações pode ser que o otimizador não consiga encontrar um bom plano para uma consulta complexa (como uma consulta aninhada com correlação). Nestes casos, reescrever a consulta para utilizar relações temporárias visando orientar o otimizador a encontrar um bom plano pode ser mais vantajoso.

8. Impacto de Concorrência

Em um ambiente com vários usuários simultâneos, diversos pontos adicionais precisam ser considerados. Transações obtém *locks* sobre as páginas que utilizam, podendo ficar bloqueadas aguardando por estes *locks*. Para reduzir a duração destes bloqueios e conseqüentemente obter um melhor desempenho, duas abordagens serão discutidas, a seguir.

8.1. Reduzindo Durações de *Locks*

Adiamento das Requisições de *Locks*: As transações devem ser ajustadas para utilizarem variáveis de programas locais, postergando para o último momento as alterações à base de dados. Este procedimento adia a requisição dos *locks* correspondentes e reduz o tempo de duração dos mesmos.

Transações Mais Rápidas: Quanto mais rápida for uma transação, mais depressa seus *locks* serão liberados. Diversas técnicas podem ser empregadas para este fim, como por exemplo o ajuste de índices, visto anteriormente, e a utilização de paralelismo no meio de armazenamento (por exemplo, índices sendo persistidos em discos distintos de suas relações).

Substituição de Transações Longas por Curtas: Às vezes muitas tarefas são realizadas numa mesma transação, retendo diversos *locks* por muito tempo. A quebra destas transações longas por transações menores pode ser vantajosa em termos de desempenho, desde que as aplicações possam lidar com o fato de não mais existir atomicidade entre as tarefas de diferentes transações.

Construção de *Warehouse*: Existem consultas complexas que retém *locks* compartilhados por muito tempo. Em geral, muitas destas consultas envolvem a análise estatística de tendências de negócios que poderiam ser executadas sobre uma cópia dos dados levemente desatualizada. Este conceito levou à popularidade das *data warehouses*, que são bases de dados que complementam a base principal, mantendo uma cópia dos dados utilizados em consultas complexas. Com isso, a sobrecarga de consultas de longa duração sobre a base de dados principal é aliviada.

Nível de Isolamento mais Baixo: Em diversas situações, tais como consultas gerando informações agregadas ou resumos estatísticos, pode-se usar um nível de isolamento SQL mais baixo, como REPEATABLE READ ou READ COMMITTED. Isto acarreta em sobrecargas menores de *locks*, apesar do programador da aplicação ter que ponderar as vantagens e desvantagens relacionadas a cada projeto.

8.2. Reduzindo *Hot Spots*

Adiamento de Operações em *Hot Spots*: Conforme mencionando anteriormente, o adiamento das requisições de *locks* é importante para se reduzir o tempo de duração dos mesmos. Este procedimento se torna ainda mais interessante para as requisições envolvendo objetos freqüentemente utilizados.

Otimização dos Padrões de Acesso: O padrão de atualização de uma relação também pode ser importante. Por exemplo, se tuplas são inseridas na relação Employees na ordem de *eid* e se há um índice árvore B+ neste campo, cada inserção vai para a página da última folha da árvore B+. Isso ocasiona *hot spots* ao longo do caminho da raíz à folha mais à direita. Um índice hash poderia resolver este problema, já que o processo de hash sorteia o *bucket* no qual o registro será inserido. Outra alternativa seria indexar a relação por um outro campo.

Partição de Operações nos *Hot Spots*: Considere uma transação de entrada de dados que anexa novos registros a um arquivo. Ao invés de se anexar cada registro por transação, obtendo um *lock* na última página para cada inserção, poder-se-ia substituir esta transação por várias outras, cada qual escrevendo os registros em um arquivo local e periodicamente anexando este lote de uma só vez ao arquivo principal. Apesar de aumentar o número de tarefas, esta prática reduz a contenção de *locks* na última página do arquivo principal.

Escolha do Índice: Se uma relação é atualizada freqüentemente, índices árvore B+ podem se tornam gargalos no controle de concorrência, uma vez que todos os acessos através deles precisam passar pelas raízes. Dessa forma, a raiz e as páginas de índice imediatamente abaixo dela se tornam *hot spots*. Se o SGBD utilizar

protocolos especializados de *locks* para índices de árvores e, em particular, definir *locks* com granulosidade fina, este problema será amenizado. Felizmente diversos sistemas atuais já utilizam estas técnicas. Índices hash, por outro lado, não apresentam este tipo de problema, a menos que a distribuição dos dados esteja agrupada, com muitos registros concentrados em poucos *buckets*. Neste caso, as entradas de diretórios para estes *buckets* podem se tornar *hot spots*.

Referências

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, third edition.