CAPÍTULO 8. CONTROLE DE CONCORRÊNCIA

Este capítulo discute em detalhe o problema de controle de concorrência em bancos de dados distribuídos. Inicialmente, para motivar a discussão, são listados vários problemas que poderiam ocorrer se não houvesse qualquer controle de concorrência. Estes problemas são chamados de anomalias de sincronização. Em seguida, um modelo abstrato de transações, já incorporando mecanismos de controle de integridade, é introduzido. O critério fundamental de correção para algorítmos de controle de concorrência, serialização, é o próximo assunto. O corpo principal do capítulo apresenta vários algorítmos para controle de concorrência, grupados em dois métodos principais, bloqueio e pré-ordenação. A apresentação de cada método é acompanhada de uma discussão sobre problemas adicionais, ocasionados pelo método, que possam impedir o término normal das transações. A discussão acerca do uso de bloqueios para bancos de dados centralizados é apresentada em separado do caso distribuído, enquanto que o uso de pré-ordenação cobre apenas o caso distribuído.

8.1 INTRODUÇÃO

Esta seção apresenta exemplos de anomalias de sincronização, e introduz os critérios básicos de correção para controle de concorrência e a notação a ser usada no capítulo. O modelo de processamento de transações adotado nos últimos capítulos também é aqui revisto.

8.1.1 Anomalias de Sincronização

Todo método de controle de concorrência deve evitar certos problemas, chamados de *anomalias de sincronização*, que podem resultar do acesso concorrente irrestrito aos dados. As principais anomalias são:

- perda da consistência do banco
- acesso a dados inconsistentes
- perda de atualizações

Estas anomalias serão ilustradas através de exemplos informais que utilizam um banco de dados centralizado (embora o fato de ser centralizado ou distribuído seja irrelevante para esta discussão). Os esquemas de relação do banco são os seguintes:

CURSOS[CODIGO,NOME,NMATR] representa os cursos oferecidos em um semestre, onde NMATR indica o número de alunos matriculados em cada particular curso;

TURMAS[MATRICULA,CODIGO] indica que alunos (representados pelo número de MATRICULA) estão matriculados em que cursos (representados pelo CODIGO).

Há três critérios de consistência para este banco de dados:

C1. o CODIGO de cada curso é unico.

C2. todo CODIGO usado em TURMAS deve estar listado em CURSOS

C3. para cada curso *c*, NMATR contém o total de alunos matriculados em *c*, conforme indicado em TURMAS.

Considere agora três transações sobre este banco, definidas da seguinte forma:

```
MATRICULE(m,c):
COMECO-DE-TRANSACAO
M1.   LEIA a tupla t de CURSOS com CODIGO=c;
      if t realmente existir
      then  begin
M2.               ESCREVA a tupla (m,c) em TURMAS;
                  incremente de 1 o campo NMATR de t;
M3.               REESCREVA a tupla t EM CURSOS;
            end.
FIM-DE-TRANSACAO

CANCELE(c):
INICIO-DE-TRANSACAO
C1.   REMOVA a tupla de CURSOS com CODIGO=c;
C2.   REMOVA todas as tuplas de TURMAS com CODIGO=c;
FIM-DE-TRANSACAO

LISTE(m)
COMECO-DE-TRANSACAO
L1.   LEIA todas as tuplas de TURMAS com MATRICULA=m;
      liste as tuplas lidas;
L2.   LEIA todas as tuplas de CURSOS tais que o CODIGO foi lido no comando anterior;
      liste todas as tuplas lidas;
FIM-DE-TRANSACAO
```

É importante observar que cada uma destas transações preserva a consistência do banco. De fato, a transação MATRICULE(m,c) primeiro verifica a existência do curso *c* antes de efetivamente matricular *m* em *c*. Da mesma forma, a transação CANCELE(c) retira o curso *c* de CURSOS e todos os alunos matriculados neste curso de TURMAS. No entanto, se estas transações forem executadas concorrentemente, sem nenhum controle, anomalias de sincronização poderão ocorrer.

Por simplicidade, nos exemplos que se seguem, execuções concorrentes serão representadas por seqüências de rótulos correspondendo a comandos que acessam o banco de dados (os rótulos são aqueles dados aos comandos na definição das transações). Comandos que não acessam o banco de dados não influenciam a discussão, sendo, portanto, ignorados. Os valores dos parâmetros das transações são indicados fora da própria seqüência de rótulos. Caso haja mais de uma execução da mesma transação, cada uma das execuções e os rótulos correspondentes serão distinguidos por subscritos.

Assim a seqüência

$M1_1$ $M2_1$ $M3_1$ L1 L2 C1 C2 $M1_2$ $M2_2$ $M3_2$

indica uma execução seqüencial das transações MATRICULE$_1$(m,c), LISTE(m), CANCELE(c) e MATRICULE$_2$(m,c), nesta ordem.

Suponha que o curso INF2045 exista e que o estado inicial do banco de dados seja consistente. Considere uma execução concorrente de MATRICULE(82.3827,INF2045) e CANCELE(INF2045), representada pela seguinte seqüência de comandos:

M1 C1 C2 M2 M3

Esta seqüência viola o segundo critério de consistência do banco. De fato, embora *M1* corretamente determine que o curso INF2045 existe no estado inicial e, portanto, o aluno cuja matricula é 82.3827 pode nele se matricular, o comando *C1* executado imediatamente em seguida remove este curso. Logo, no estado final do banco de dados, a relação associada a TURMAS conterá a tupla (82.3827,INF2045), sem que haja nenhuma tupla na relação associada a CURSOS com CODIGO=INF2045. Isto constitui uma violação do segundo critério de consistência. Além disto, embora não produza erro propriamente, o comando *M3* fica sem ação pois o curso INF2045 não mais existe quando *M3* é executado.

Este é, então, um exemplo de uma execução concorrente de transações que leva a perda de consistência do banco, embora cada transação por si só preserve consistência.

Como um outro exemplo de anomalias de sincronização, suponha que o aluno 82.5694 está inicialmente matriculado no curso INF2045. Considere uma execução concorrente de LISTE(82.5694) e CANCELE(INF2045) representada pela seguinte seqüência:

L1 C1 C2 L2

Neste caso, o resultado apresentado por LISTE é inconsistente pois indica que o aluno 82.5694 está matriculado no curso INF2045, como resultado de *L1*, mas este curso não é listado por *L2*, pois foi cancelado por *C1*. Ou seja, o resultado apresentado por LISTE não satisfaz ao segundo critério de consistência do banco. Temos aqui uma situação de acesso a dados inconsistentes por parte da transação LISTE.

Para concluir esta seqüência de exemplos, suponha novamente que o curso INF2045 exista inicialmente. Considere uma execução concorrente de MATRICULE$_1$(82.5782,INF2045) e MATRICULE$_2$(82.4920,INF2045) representada pela seguinte seqüência:

$M1_1$ $M1_2$ $M2_2$ $M3_2$ $M2_1$ $M3_1$

Esta execução leva a uma perda de atualização pois o valor final de NMATR para o curso INF2045 reflete apenas a matrícula de 82.5782, e não a dos dois alunos. Isto se deve ao fato de *M3* incrementar e reescrever o valor lido por *M1*, e não o valor corrente de NMATR. Isto é, $M1_1$ e $M1_2$ lêem ambas o valor inicial de NMATR para o curso INF2045; $M3_1$ e $M3_2$ ambas incrementam este valor; mas $M3_1$ escreve sobre o valor criado por $M3_2$, em lugar de incrementá-lo.

(O leitor deve se convencer de que o último exemplo também leva à perda de consistência do banco).

Isto completa a nossa discussão sobre anomalias de sincronização. A Seção 8.1.2 introduzirá

uma classe de execuções concorrentes, chamadas de serializáveis, onde estes problemas não ocorrem.

8.1.2 Modelagem do Sistema

O estudo de controle de concorrência será feito assumindo-se o mesmo modelo de SGBD distribuído e de transações usado nos capítulos anteriores. Esta seção recorda os aspectos do modelo relevantes para a discussão sobre controle de concorrência.

A nível lógico, o banco de dados é descrito por um esquema conceitual global consistindo de um conjunto de objetos lógicos. A nível físico, o banco é descrito por uma série de esquemas internos, um para cada nó onde está armazenado; cada esquema interno consiste de um conjunto de objetos físicos. Os mapeamentos do esquema conceitual global para os esquemas internos definem a forma de distribuição do banco e a correspondência entre objetos físicos e objetos lógicos. Estes mapeamentos poderão determinar que certos conjuntos de objetos físicos armazenem cópias dos mesmos dados. Os objetos lógicos são manipulados através de comandos da LMD e os objetos físicos através de ações elementares.

A execução de uma transação é controlada pelo gerente de transações (*GT*) do nó onde foi submetida. A nível lógico, a execução de uma transação processa-se da seguinte forma:

COMEÇO-DE-TRANSAÇÃO: o *GT* ao interceptar este comando cria uma área de trabalho para a transação;

comandos da LMD: consultas puras acessam objetos lógicos do banco de dados trazendo-os para a área de trabalho, se já lá não estiverem. Atualizações sobre os objetos lógicos são mantidas na própria área de trabalho, não se tornando visíveis de imediato a outras transações;

FIM-DE-TRANSAÇÃO: invoca o protocolo bifásico para modificar todas as cópias de todos os objetos lógicos afetados por atualizações executadas pela transação. Os valores dos objetos lógicos são obtidos da área de trabalho.

Suporemos que em cada nó participando do processamento da transação há uma área de trabalho da transação.

Todas as operações a nível lógico são sempre traduzidas em seqüências de operações a nível físico. Mais precisamente, uma execução de um grupo de transações gera, em cada nó onde o banco está armazenado, uma seqüência de ações elementares, que suporemos serem de dois tipos:

R(X) ação de *leitura* que recupera os valores dos objetos físicos cujo nome está no conjunto *X* para a área de trabalho local da transação que gerou a ação;

W(X) ação de *atualização* que escreve os novos valores dos objetos físicos em *X* no banco de dados local.

Assim, a execução de um comando da LMD (que é uma operação a nível lógico) poderá gerar várias ações do tipo *R(X)* para recuperar objetos físicos que ainda não estão na área de trabalho da transação. Porém, um comando da LMD nunca gerará ações elementares do tipo *W(X)* pois o banco de dados não é alterado de imediato. Apenas quando o protocolo bifásico atingir a segunda fase, ações elementares do tipo *W(X)* serão geradas para efetivar alterações nos bancos de dados locais.

Há duas suposições importantes para controle de concorrência a se ressaltar aqui:

1. *controle de concorrência será feito a nível dos objetos físicos. Portanto, um mecanismo de controle de concorrência deverá disciplinar a intercalação das ações elementares de diferentes transações em cada nó.*

2. *A semântica das transações não será levada em conta pelos mecanismos de controle de concorrência.*

Como conseqüência, o controle de concorrência deverá depender apenas das seqüências de operações de leitura/atualização sobre os objetos físicos armazenados nos vários bancos de dados locais, ou seja, das seqüências de operações *R(X)* e *W(X)* executadas contra os bancos de dados locais. Uma execução concorrente *E* de um conjunto *T* de transações pode, então, ser abstraída por um conjunto $L = \{ L_1 ,..., L_n \}$, onde L_i é a seqüência de ações elementares *R(X)* ou *W(X)* executadas contra o banco de dados do nó *i* que foram geradas em *E*. O conjunto *L* é chamado de um *escalonamento global* para *T* e a seqüência L_i é chamada do *escalonamento local* ao nó *i* para *T*. Usaremos $R_i(X)$ ou $W_i(X)$ para indicar ações elementares *R(X)* ou *W(X)* executadas a favor da transação T_i em um escalonamento local. Freqüentemente escreveremos $R_i (x_1 ,..., x_k)$ em lugar de $R_i (\{ x_1 ,..., x_k \})$, e semelhantemente para ações de atualização.

Como exemplos destes conceitos, considere um banco de dados distribuído armazenado em dois nós. Os esquemas internos são modelados por dois conjuntos de objetos físicos, $D_1 = \{ x_1 ,y_1 \}$ e $D_2 = \{ x_2 \}$, onde x_1 e x_2 armazenam cópias dos mesmos dados.

Um escalonamento global para duas transações T_1 e T_2 neste contexto poderia ser

$L = \{ L_1 , L_2 \}$, onde

$L_1 = R_1(y_1)\ R_2(x_1)\ W_2(x_1)\ W_1(x_1)$

$L_2 = R_1(x_2)\ W_1(x_2)\ W_2(x_2)$

Ou seja, L_1 representa a seguinte seqüência de ações elementares executadas no primeiro nó:

T_1 lê o objeto físico y_1

T_2 lê o objeto físico x_1

T_2 escreve no objeto físico x_1

T_1 escreve no objeto físico x_1

Para o segundo nó, L_2 representa a seguinte seqüência:

T_1 lê o objeto físico x_2

T_1 escreve no objeto físico x_2

T_1 escreve no objeto físico x_1

Tanto a teoria de correção quanto o estudo do comportamento dos métodos de controle de concorrência serão baseados em propriedades de escalonamentos globais.

8.1.3 Critérios de Correção

Esta seção define os critérios de correção que guiarão a discussão sobre controle de concorrência. Serão considerados critérios pertencentes a três classes distintas: critérios para transações, critérios genéricos para o sistema e critérios específicos para os métodos de controle de concorrência.

Os critérios para transações são simples:

T1. Cada transação, quando executada sozinha, sempre termina;

T2. Cada transação, quando executada sozinha, preserva consistência do banco de dados;

Estas suposições afirmam apenas que o usuário especificou corretamente cada transação.

Os critérios genéricos do sistema por sua vez serão os seguintes:

G1. O sistema deve funcionar corretamente para qualquer conjunto de transações acessando qualquer banco de dados;

G2. A resposta do sistema deve ser independente do significado das transações e dos valores dos próprios dados armazenados.

A primeira suposição justifica-se com base no fato de estarmos interessados em construir SGBDDs de uso genérico, e não em sistemas distribuídos para aplicações específicas. Logo, não é razoável supor que as transações são conhecidas "a priori", ou que o sistema seja dependente de um particular conjunto de transações acessando um particular banco de dados. Já a segunda suposição sugere que os métodos de controle de concorrência devam trabalhar apenas com base nos nomes dos objetos físicos lidos e atualizados, conforme comentado no final da seção anterior.

Esta discussão nos coloca em posição de definir intuitivamente os critérios de correção impostos aos métodos de controle de concorrência:

C1. Cada transação submetida ao sistema deve eventualmente terminar.

C2. Cada transação deve ser executada atomicamente, sem interferência das outras transações;

O primeiro critério é claro e resume a idéia de que o método de controle de concorrência deverá prover meios para resolver problemas, como bloqueios mútuos, que possam impedir o término normal das transações. Já o segundo critério, o mais importante de todos, requer uma discussão pormenorizada para esclarecer o que significa "execução atômica sem interferência". Este será o assunto da próxima seção.

*8.2 TEORIA DA SERIALIZAÇÃO

A teoria da serialização se propõe a capturar de forma precisa quando, em uma execução concorrente de um grupo de transações, cada uma delas é executada atomicamente sem interferência. Execuções com esta propriedade são chamadas de serializáveis. O objetivo desta seção será dar uma definição precisa da noção de execução serializável, que é essencial ao entendimento da correção dos métodos de controle de concorrência discutidos nas seções seguintes deste capítulo.

Intuitivamente, uma execução concorrente é serializável se for computacionalmente equivalente a uma execução serial das transações, ou seja, a uma execução em que as transações são processadas seqüencialmente, uma após a outra, em alguma ordem. Para formular precisamente este conceito intuitivo é necessário definir dois conceitos: o que são execuções seriais e quando duas execuções são consideradas computacionalmente equivalentes.

8.2.1 Execuções Seriais

Em termos simples, uma execução é serial se as transações são executadas seqüencialmente. Ou seja, uma execução *E* de *T* modelada por um escalonamento global *L* é *serial* se e somente se

1. para cada escalonamento local de *L*, para cada par de transações T_i e T_j em *T*, ou todas as operações de T_i precedem todas as operações de T_j, ou vice-versa;

2. para cada par de transações T_i e T_j, se as operações de T_i precedem as operações de T_j em um escalonamento local de *L*, então o mesmo é verdade para todos os outros escalonamentos locais de *L*.

Diremos ainda que *L* é um *escalonamento serial* neste caso.

Como exemplo, considere um banco de dados distribuído armazenado em dois nós cujos esquemas internos são modelados por dois conjuntos de objetos físicos, $D_1 = \{ x_1 , y_1 \}$ e $D_2 = \{ x_2 \}$. Suponha que x_1 e x_2 armazenem cópias dos mesmos dados. Um escalonamento serial seria :

$S = \{ S_1 , S_2 \}$, onde

$S_1 = R_1(y_1)\ W_1(x_1)\ R_2(x_1)\ W_2(x_1)$

$S_2 = R_1(x_2)\ W_1(x_2)\ W_2(x_2)$

Como exemplos de escalonamentos não seriais teríamos:

$N = \{ N_1 , N_2 \}$, onde

$N_1 = R_1(y_1)\ R_2(x_1)\ W_2(x_1)\ W_1(x_1)$

$N_2 = R_1(x_2)\ W_1(x_2)\ W_2(x_2)$

e

$N' = \{ N_1' , N_2' \}$, onde

$N_1' = R_1(y_1)\ R_2(x_1)\ W_2(x_1)\ W_1(x_1)$

$N_2' = W_2(x_2)\ R_1(x_2)\ W_1(x_2)$

O primeiro exemplo viola a primeira condição para escalonamentos seriais, enquanto o segundo exemplo viola a segunda condição.

O método de controle de concorrência trivial, que só permite execuções seriais, é obviamente correto dentro dos critérios estabelecidos anteriormente. Não há dúvidas de que cada transação é executada atomicamente sem interferência de outras se este método é seguido.

Também deve estar claro que em uma execução serial *S* anomalias de sincronização não ocorrem. Por suposição, cada transação preserva consistência e termina se executada sozinha. Logo, se o estado inicial do banco for consistente, o estado do banco após a execução da :f/i/-ésima transação em *S* também será consistente. Assim, se o estado inicial do banco for consistente, o estado final também o será, o que significa que *S* preserva consistência. Pela mesma razão, nenhuma transação lê dados inconsistentes em *S* pois o faz de um estado consistente. Finalmente, como cada transação é processada após o término da anterior, obviamente nenhuma atualização é perdida em *S*.

8.2.2 Equivalência de Execuções

Passemos agora para o problema de definir o que significa duas execuções serem computacionalmente equivalentes. Intuitivamente, duas execuções *E* e *E'* de um mesmo conjunto *T* de transações são computacionalmente equivalentes se e somente se as seguintes condições forem satisfeitas, supondo que *E* e *E'* começam no mesmo estado do banco de dados:

1. *E* e *E'* produzem o mesmo estado final do banco de dados;

2. cada transação em *T* lê os mesmos dados em *E* e *E'*.

Mais precisamente, seja *T* um conjunto de transações e *E* um execução de *T* modelada por um escalonamento global *L*. Sejam *R(X)* e *W(Y)* duas ações elementares em algum escalonamento local L_i de *L*. Seja *x* um objeto físico armazenado em um nó *i* tal que $x \in X \cap Y$. Diremos que *R(X) lê x de W(Y)* em L_i se *W(Y)* precede *R(X)* em L_i e não há nenhuma operação *W(Z)* entre *W(Y)* e *R(X)* em L_i tal que $x \in Z$. Seja $y \in X$. Diremos que *R(X) lê o valor inicial de y* em L_i se *R(X)* não lê *y* de nenhum *W(Y)*. Similarmente, seja $z \in Y$. Diremos que *W(Y) cria o valor final de z* em L_i se *W(Y)* é a última operação de atualização em L_i tal que $z \in Y$. Estas duas últimas noções poderiam ser reduzidas à primeira se imaginássemos uma transação inicial que "cria" o estado inicial de *E*, e uma transação final que "lê" o estado final produzido por *E*.

Sejam *E* e *E'* duas execuções para um conjunto *T* de transações. Sejam $L = \{ L_1 ,..., L_n \}$ e $L' = \{ L'_1 ,..., L'_n \}$ escalonamentos globais modelando *E* e *E'*. Diremos que *E* e *E'* são *equivalentes* se e somente, para todo $j \in [1,n]$:

1. L_j e L_j' contém as mesmas ações elementares, e as ações de cada transação ocorrem na mesma ordem relativa em ambas.

2. para cada objeto físico *x*, para cada *R(X)* em L_j tal que $x \in X$, *R(X)* lê o valor inicial de *x* em L_j se e somente se o faz em L_j' ;

3. para cada objeto físico *x*, para cada *W(Y)* em L_j tal que $x \in Y$, *W(Y)* cria o valor final de *x* em L_j se e somente se o faz em L_j' ;

4. para cada *R(X)* em L_j , para cada *W(Y)* em L_j , para cada $x \in X \cap Y$, *R(X)* lê *x* de *W(Y)* em L_j se e somente se o faz em L_j'.

Diremos ainda que *L* e *L'* são *escalonamentos equivalentes*.

Intuitivamente, esta definição garante que cada transação é processada da mesma forma em ambas as execuções pois as operações de leitura lêem os mesmos valores. Além disto, o

estado final do banco de dados é o mesmo, pois o valor final de cada objeto físico foi produzido pela mesma operação de atualização em ambas as execuções.

8.2.3 Execuções Serializáveis

Seja *E* uma execução para um conjunto *T* de transações modelada por um escalonamento *L*. *E* é *serializável* se e somente se for equivalente a uma execução serial. Diremos também que *L* é um *escalonamento serializável*.

Portanto, o critério C2 da Seção 8.1.1.3 pode ser precisamente reformulado como:

C2'. toda execução de um conjunto de transações *T* deverá ser serializável.

Um método de controle de concorrência deverá então permitir apenas execuções serializáveis das transações. Com isto o método estará garantindo que nenhuma anomalia de sincronização aparecerá. A justificativa é simples: em execuções seriais tais anomalias não ocorrem; como as execuções serializáveis são computacionalmente equivalentes às execuções seriais, elas herdam, então, esta propriedade. Se o leitor preferir não analisar a questão em termos de anomalias de sincronização, basta argumentar que execuções seriais são "naturalmente corretas" do ponto de vista da execução das transações. Portanto, gerando apenas execuções que lhes são equivalentes, a propriedade de correção é mantida.

O resto desta seção apresenta uma série de exemplos envolvendo o conceito de serialização. Considere inicialmente um banco de dados centralizado cujo esquema interno é modelado por um conjunto de objetos físicos $D = \{ x , y \}$. Considere duas transações, T_1 e T_2, cujas execuções seqüenciais geram, respectivamente, as seqüências de ações elementares:

$L_1 = R_1(X)\ W_1(X)$

$L_2 = R_2(y)\ W_2(X)$.

Há apenas dois possíveis escalonamentos seriais neste caso:

$S_{12} = R_1(X)\ W_1(X)\ R_2(y)\ W_2(X)$

$S_{21} = R_2(y)\ W_2(X)\ R_1(X)\ W_1(X)$.

:exmp. Além de S_{12} e S_{21}, que são obviamente serializáveis, os seguinte escalonamentos também são serializáveis:

$E_1 = R_1(X)\ R_2(y)\ W_1(X)\ W_2(X)$

$E_2 = R_2(y)\ R_1(X)\ W_1(X)\ W_2(X)$.

Ambos são equivalentes a S_{12} pois, como $R_2(y)$ lê o valor inicial de *y* em S_{12}, podemos comutar esta ação com $W_1(x)$ obtendo E_1, e depois com $R_1(x)$, obtendo E_2. O leitor deve se convencer que estes são os dois únicos escalonamentos serializáveis além de S_{12} e S_{21}.

Como um segundo exemplo, considere novamente um banco de dados distribuído armazenado em dois nós cujos esquemas internos são modelados por dois conjuntos de objetos físicos, $D_1 = \{ x_1 , y_1 \}$ e $D_2 = \{ x_2 \}$. Suponha que x_1 e x_2 armazenam cópias dos mesmos dados. Um exemplo de um escalonamento serializável seria

$L = \{L_1, L_2\}$, onde

$L_1 = R_2(x_1)\ R_1(y_1)\ W_2(x_1)\ W_1(x_1)$

$L_2 = W_2(x_2)\ R_1(x_2)\ W_1(x_2)$

que é equivalente ao escalonamente serial em que T_2 é executada completamente antes de T_1 ser processada.

Como exemplos de escalonamentos não serializáveis teríamos

$N = \{N_1, N_2\}$, onde

$N_1 = R_1(y_1)\ R_2(x_1)\ W_1(x_1)\ W_2(x_1)$

$N_2 = R_1(x_2)\ W_1(x_2)\ W_2(x_2)$

e

$N' = \{N_1', N_2'\}$, onde

$N_1' = R_2(x_1)\ W_2(x_1)\ R_1(y_1)\ W_1(x_1)$

$N_2' = W_2(x_2)\ R_1(x_2)\ W_1(x_2)$

No primeiro exemplo, N não é serializável pois o próprio escalonamento local N_1 já não o torna equivalente a algum escalonamento serial. Não é possível, intuitivamente, trazer $R_2(x_1)$ para junto de $W_2(x_1)$ em N_1 sem alterar a computação expressa por N_1. O segundo exemplo, N', é interessante pois N_1' e N_2' são por si só seriais, mas as transações aparecem na ordem trocada em cada um. Além disto, não é possível alterar a ordem das ações sem alterar a computação final. Este exemplo ilustra o fato de que serialização não pode ser detetada localmente: mesmo que todos os escalonamentos locais sejam serializáveis, o escalonamento global poderá não o ser.

A caracterização de execuções serializáveis dada pela definição anterior captura corretamente o conceito de atomicidade das transações em um ambiente concorrente, mas ainda não leva a métodos de controle de concorrência. Na verdade é possível provar que apenas testar se um escalonamento é serializável é, provavelmente, computacionalmente intratável (mais precisamente, NP-Completo).

8.2.4 Uma Condição Suficiente para Serialização

Nesta seção será apresentada uma condição suficiente (mas não necessária) para garantir serialização. Esta condição será usada nas seções seguintes para provar a correção de métodos de controle de concorrência.

Seja T um conjunto de transações e L um escalonamento global para T. Seja L_k um escalonamento local de L. Duas ações elementares O_i e O_j de L_k *conflitam* se e somente se elas agem sobre um mesmo objeto físico e uma delas é uma operação de atualização. Operações conflitantes são importantes pois, se a sua ordem relativa for alterada em L_k, o resultado final da execução poderá ser modificado. Considere, por exemplo, as operações *R(X)* e *W(X)*. Suponha que L_k seja da forma '... *R(X)* ... *W(X)* ...'. Logo *R(X)* obviamente não lê o valor de $x \in X$ que foi criado por *W(X)*. Se a ordem das operações for trocada em L_k para '... *W(X)* ... *R(X)* ...' e entre *W(X)* e *R(X)* não houver uma outra operação de atualização para $x \in$

X, *R(X)* passará agora a ler o valor criado por *W(X)*, possivelmente (mas não necessariamente) alterando o estado final do banco de dados. Um cenário semelhante pode naturalmente ser criado para duas operações de atualização. Definiremos ainda que O_i *precede com conflito* O_j em L_k (denotado por $O_i < O_j$) se e somente se O_i ocorre antes de O_j em L_k e O_i e O_j conflitam. Quando mais de uma relação de precedência por conflito estiver em jogo, subscritos serão usados para distinguí-las.

De posse desta relação entre ações elementares, diremos que T_i *precede por conflito* T_j em L (denotado por $T_i < T_j$) se e somente se existir um escalonamento local L_k de L e operações O_i e O_j em L_k tais que O_i e O_j são operações de T_i e T_j respectivamente e $O_i < O_j$. A relação < será chamada de *relação de precedência por conflito* para T induzida por L. Novamente quando mais de uma destas relações estiverem em jogo, subscritos serão usados para distinguí-las.

Podemos, então, mostrar o seguinte:

TEOREMA 1: Seja $T = \{ T_1 ,..., T_m \}$ um conjunto de transações e E uma execução de T modelada por um escalonamento global $L = \{ L_1 ,...,L_n \}$. Se a relação de precedência por conflito para T induzida por L for uma relação de ordem parcial, então E é serializável.

Demostração

Provaremos que, para todo conjunto T de transações, para toda execução E de T modelada por um escalonamento global L, se a relação de precedência por conflito para T induzida por L for uma relação de ordem parcial, então E é serializável. A prova será por indução sobre a cardinalidade de T.

BASE: Suponha que T tenha apenas uma transação. Então o resultado segue trivialmente.

PASSO DE INDUÇÃO: Suponha que o resultado vale para todo conjunto de transações com cardinalidade menor do que n. Seja T um conjunto de transações com cardinalidade n e E uma execução de T modelada por um escalonamento global L. Suponha que a relação $<_L$ sobre T induzida por L seja uma relação de ordem parcial.

Seja T_i uma transação em T tal que para nenhuma transação T_j temos que $T_j <_L T_i$. Construa uma execução F de T, modelada por um escalonamento F, onde T_i é inicialmente executada seqüencialmente e depois as outras transações em T são executadas concorrentemente exatamente como em E. Assim, cada escalonamento local M_k de F é obtido trazendo-se todas as ações elementares de T_i no escanolamento local L_k de L para a esquerda (e respeitando a sua ordem relativa). Por construção, a relação $<_L$ coincide com com a relação $<_M$. Além disto, como não existe T_j tal que $T_j <_L T_i$, não existe uma ação elementar O_j de alguma transação T_j em T, e uma ação elementar O_i de T_i em L_k tais que $O_j <_L O_i$. Ou seja, nenhuma ação elementar O_j que precede alguma ação elementar O_i de T_i em L_k conflita com O_i Assim, E' e E são equivalentes.

Construa agora G retirando as ações elementares de T_i de F. Seja N o escalonamento global representando G. Teremos então que G é uma execução do conjunto de transações $U = T - \{ T_i \}$, que tem cardinalidade menor do que n. Além disto, a relação $<_N$ é um subconjunto da relação $<_L$ pois, por construção, G é uma subseqüência de E. Logo, $<_N$ também é acíclica. Pela hipótese de indução, podemos então concluir que G é serializável.

Seja SG uma execução serial equivalente a G. Construa uma execução serial SF das

transações em T processando T_i primeiro e depois as outras transações em T na mesma ordem que em SG. Por construção e pelo fato de SG e G serem equivalentes, SF e F serão então equivalentes. Mas E e F eram equivalentes. Logo, SF e E são equivalentes, o que prova que E é serializável.

Para ver que a condição apresentada no teorema anterior não é necessária, considere o seguinte escalonamento em um banco de dados centralizado:

$$L = R_1(x)\ W_2(x)\ W_1(x)\ W_3(x)$$

Como $T_1 < T_2 < T_1$, a relação de precedência por conflito para as transações induzida por L não é uma relação de ordem parcial. Mas, por outro lado, L é equivalente ao seguinte escalonamento serial:

$$S = R_1(x)\ W_1(x)\ W_2(x)\ W_3(x)$$

pois os valores de x atualizados por T_1 e T_2 não contribuem nem para a execução de T_3, nem para o estado final do banco de dados já que $W_3(x)$ escreve sobre eles.

Os métodos de controle de concorrência descritos nas seções que se seguem garantirão que a relação < é sempre uma relação de ordem parcial para as transações em processamento e, assim, que todas as execuções são serializáveis.

8.3 MÉTODOS BASEADOS EM BLOQUEIOS - PARTE I

Esta seção discute o uso de bloqueios para controle de concorrência em um ambiente centralizado. Inicialmente os problemas de gerência de bloqueios e tratamento de bloqueios mútuos são abordados. Em seguida, um método de uso de bloqueios para atingir apenas execuções serializáveis, chamado de bloqueio em duas fases, é apresentado. Por fim, a correção do método é provada.

8.3.1 Protocolo de Bloqueio de Objetos

Nesta seção um protocolo de bloqueio/liberação de objetos é discutido bem como as questões de tipos de bloqueio e granularidade dos objetos bloqueados.

Consideremos inicialmente o caso mais simples em que todos os objetos a serem bloqueados são de uma mesma classe (páginas físicas, por exemplo). Suponha ainda que só há um modo de bloqueio, ou seja, que cada objeto só pode estar em dois estados:

bloqueado permite acesso ao objeto apenas pela transação que detém o bloqueio;

livre não permite acesso ao objeto por nenhuma transação.

Neste caso é necessário introduzir apenas duas novas ações elementares ao nosso repertório (que contém até o momento apenas $R(X)$ e $W(X)$):

$B(x)$ bloqueie o objeto cujo nome é x;

$L(X)$ libere todos os objetos cujos nomes estão em X;

Note que $B(x)$ afeta apenas um único objeto, diferentemente das outras ações elementares. Esta opção torna o tratamento de bloqueios mais simples, conforme veremos.

Para acomodar estas novas ações elementares, uma execução de um conjunto de transações será representada agora pela seqüência de ações elementares *R(X)*, *W(X)*, *B(x)* ou *L(X)* processadas contra o banco de dados centralizado. Esta seqüência continuará a ser chamada de um *escalonamento*.

Estas ações são passadas para o gerente de bloqueios, que mantém uma *tabela de bloqueios*, modelada como uma coleção de triplas *(x,T,F)* onde:

x é o nome de um objeto

T é o nome da transação que correntemente bloqueia *x*

F é uma *fila de espera* para *x* contendo os nomes de todas as transações que esperam a liberação de *x*

Suporemos que as filas de espera seguem a política estrita primeiro-a-chegar-primeiro-a-sair. Esta política poderia ser alterada, adotando-se filas com prioridade, por exemplo. Porém, qualquer política adotada deverá garantir que uma transação não fica eternamente na fila de espera.

A noção de um objeto estar bloqueado ou livre é implementada através de um *protocolo de bloqueio e liberação* de objetos definido como (λ indica a fila vazia):

1) Inicialmente a tabela de objetos bloqueados está vazia.
2) Ao receber solicitação para bloquear o objeto *x* para a transação *T* através da ação *B(x)*, pesquise a tabela de bloqueios procurando uma tripla cujo primeiro elemento seja *x*:
   a) se nenhuma tripla for encontrada (ou seja, se *x* está livre), bloqueie *x* para *T*, acrescentando a tripla *(x,T,λ)* à tabela.
   b) caso contrário, acrescente *T* ao final da fila de espera para *x* na tripla encontrada.
3) Ao receber solicitação da transação *T* através da ação *L(X)* para liberar os objetos em *X*, para cada $x \in X$, pesquise a tabela de bloqueios procurando uma tripla cujos dois primeiros elementos sejam *x*, *T*:
   a) se nenhuma tripla for encontrada, ignore a liberação de *x*.
   b) caso contrário, seja *(x,T,F)* a tripla encontrada:
      i) se a fila *F* estiver vazia, retire a tripla da tabela, liberando *x*.
      ii) se a fila *F* não estiver vazia, ou seja, se for da forma *T'.F'* , passe o controle de *x* para *T'* , substituindo a tripla *(x,T,F)* na tabela de bloqueios por *(x,T',F')*.

Diz-se que uma execução (e o escalonamento que a representa) é *legal* se e somente se obedece ao protocolo de bloqueio/liberação de objetos e uma ação elementar de uma transação T_i que acessa um objeto *x* só é processada depois que *x* for bloqueado para T_i. De agora em diante, quando nos referirmos a um escalonamento com bloqueios, estaremos implicitamente assumindo que é legal.

O protocolo básico de bloqueio/liberação pode ser melhorado incorporando-se *modalidades* (ou *modos*) diferentes de bloqueio. Uma opção seria adotar duas modalidades de bloqueio, *partilhado* e *exclusivo*. Isto significa que agora um objeto poderá estar em três estados:

bloqueado partilhadamente: permite acesso ao objeto por todas as transações que bloqueiam o objeto partilhadamente;

bloqueado exclusivamente: permite acesso ao objeto apenas pela transação que o bloqueia exclusivamente;

livre: não permite acesso ao objeto por nenhuma transação;

Uma forma mais precisa de definir a compatibilidade das modalidades de bloqueio seria através de uma *matriz de compatibilidades* indicando quando duas transações podem bloquear o mesmo dado e quando não o podem fazer:

	partilhado	**exclusivo**
partilhado	SIM	NÃO
exclusivo	NÃO	NÃO

A justificativa para as modalidades de bloqueio acima definidas é simples. Duas ou mais transações poderão ler o objeto *x* simultaneamente, sem perigo de conflito, bloqueando-o em modo partilhado (entrada 'SIM' na matriz de compatibilidades). Por outro lado, se uma transação atualiza *X*, conflitará com qualquer outra transação que acesse *x*. Logo deverá bloquear *x* em modo exclusivo, não permitindo que nenhuma outra transação bloqueie *x* em qualquer modo (entradas 'NÃO' na matriz de compatibilidades).

Cabe observar que a matriz de compatibilidades se refere a ações de transações diferentes, e não se aplica a ações de uma mesma transação. Assim, se uma transação já mantém um objeto *x* bloqueado na modalidade partilhada e desejar bloqueá-lo na modalidade exclusiva, poderá fazê-lo se for a única transação que no momento mantém *x* bloqueado.

O protocolo de bloqueio/liberação deverá então incorporar a política expressa pelas modalidades de bloqueio. As modificações, por serem simples, são omitidas neste texto.

Um segundo melhoramento pode ainda ser incorporado ao protocolo de bloqueio/liberação criando-se uma hierarquia de objetos. Suponhamos que, em lugar de objetos de uma única classe, os objetos sejam organizados sob forma de uma floresta. Se um objeto *x* for um ancestral de *y*, diremos que *x cobre y*. Por exemplo, considere objetos de três tipos: segmentos, páginas e palavras. Um segmento será pai de todas as suas páginas e cada página será pai de todas as suas palavras.

Dentro deste esquema, uma transação pode bloquear um objeto *x* se nenhum dos objetos que *x* cobre estiver bloqueado. Uma transação ao bloquear/liberar um objeto *x*, implicitamente estará bloqueando/liberando todos os objeto que *x* cobre. Por exemplo, um segmento poderá ser bloqueado se nenhuma de suas páginas e nenhuma das palavras de suas páginas estiverem bloqueadas.

A justificativa para este tipo de bloqueio está na economia que proporciona em termos de memória ocupada e tempo adicional gasto na gerência da tabela de bloqueios. Por exemplo, em lugar de bloquear, digamos, 90% das 1000 páginas de um segmento, uma transação bloquearia o segmento inteiro. Sem o uso de bloqueios em objetos hierarquizados seriam necessárias 900 entradas na tabela de bloqueios. Com bloqueios em objetos hierarquizados, apenas uma entrada cumpriria a mesma tarefa (embora 100 páginas fossem implicitamente bloqueadas sem necessidade).

Com isto encerra-se a discussão preliminar sobre bloqueios.

8.3.2 Tratamento de Bloqueios Mútuos

8.3.2.1 Caracterização de Bloqueios Mútuos

O uso de bloqueios, sem preocupações adicionais, poderá levar transações a não terminarem. Mais precisamente, é possível que em determinado ponto da execução concorrente crie-se uma seqüência de transações $T_{i\,0}$,$T_{i\,1}$,..., $T_{i\,m-1}$, $T_{i\,0}$ tal que $T_{i\,j}$ espera por $T_{i\,j+1}$ (soma módulo *m*). Desta forma, nenhuma destas transações terminará e tem-se uma situação de *bloqueio mútuo*. A seqüência é chamada de *seqüência de impasse*.

Bloqueios mútuos são caracterizados definindo-se o *digrafo de espera G*=(*N*,*A*) para o estado corrente da tabela de bloqueios *TB* da seguinte forma:

N é o conjunto de transações que ocorrem na tabela *TB* tanto bloqueando objetos quanto nas filas de espera;

A é o conjunto de arcos (T_i , T_j) tais que há uma tripla (x,T_j ,F) em *TB* tal que T_i ocorre em *F* (ou seja, T_i *espera por* T_j liberar o objeto *x*).

É fácil observar que bloqueios mútuos não ocorrem no ponto em que *G* foi construído se e somente se *G* for acíclico. Caso contrário, cada ciclo de *G* representa uma seqüência de impasse.

As duas formas básicas para tratar o problema de bloqueios mútuos, deteção/resolução e prevenção, serão discutidas nas subseções seguintes.

8.3.2.2 Detecção / Resolução de Bloqueios Mútuos

O tratamento de bloqueios mútuos por detecção / resolução consiste em deixar as transações processarem normalmente e, periodicamente, iniciar um processo independente *P* para detectar / resolver bloqueios mútuos.

Para detetar a existência de bloqueios mútuos, o processo *P* simplesmente constroi o grafo de espera *G*, testando se *G* é acíclico ou não.

Para resolver bloqueios mútuos, o processo *P* age da seguinte forma. Se há ciclos em *G*, transações são selecionadas de tal forma que ao serem retiradas de *G* o novo grafo se torne acíclico. Usualmente para cada ciclo de *G*, é selecionada a transação que consumiu menos recursos até o momento. Cada transação selecionada é reiniciada, liberando primeiro os objetos que bloqueava. Cuidado deve ser tomado, no entanto, para que uma mesma transação não seja reiniciada repetidamente, o que a impediria de eventualmente terminar. Uma técnica para se evitar esta situação seria reiniciar, não a transação que consumiu menos recursos, mas a transação que foi submetida por último. Assim o sistema garantiria que a transação mais antiga sempre termina.

8.3.2.3 Prevenção de Bloqueios Mútuos

A forma mais simples de evitar bloqueios mútuos consiste em liberar um objeto sempre que a transação pedir novo bloqueio. Este método, embora usado em certos casos, é totalmente insatisfatório do ponto de vista de controle de concorrência pois permite a criação de execuções não serializáveis.

Uma outra forma de prevenir bloqueios mútuos consiste em exigir que cada transação bloqueie todos os objetos que irá acessar através de uma única ação indivisível, que é

executada antes da transação acessar o primeiro objeto. A única vantagem deste esquema é a sua aparente simplicidade. Porém, ele exige que todos os objetos que uma transação irá acessar sejam conhecidos inicialmente, o que nem sempre é possível, e que quase sempre leva a bloquear mais objetos do que o necessário. Além disto, este esquema exige para sua implementação que o protocolo de bloqueio/liberação seja modificado para permitir o bloqueio de vários objetos ao mesmo tempo para uma mesma transação (em lugar de apenas um de cada vez, como anteriormente). A modificação necessária não é fácil de implementar e poderá levar mesmo a problemas de bloqueio mútuo. (O leitor deverá tentar modificar o protocolo para perceber este fato. Foi justamente por este problema que decidimos por uma ação elementar que bloqueasse apenas um objeto).

Um terceiro método, satisfatório do ponto de vista de controle de concorrência, seria o seguinte. Quando uma transação T_i pede para bloquear um objeto *x*, que está presentemente bloqueado para T_j, um teste é executado. Se T_i e T_j passarem pelo teste, T_i poderá então ser adicionada à fila de espera de *x*. Caso contrário T_i ou T_j são canceladas. Se a transação T_i que solicita o bloqueio é sempre a escolhida, o método é chamado de *não-preemptivo.*. Se a transação T_j que detém o bloqueio é sempre a escolhida, o método é chamado de *preemptivo.*

O teste escolhido deverá sempre garantir que bloqueios mútuos não irão ser criados ao adicionar T_i à fila de espera de *x*. Em termos do grafo de espera, isto significa que a adição do arco (T_i,T_j) ao grafo não irá criar ciclos. Há vários testes possíveis com esta propriedade. O mais simples seria sempre reiniciar T_i ao solicitar o bloqueio, que geraria um desperdício grande de recursos. Dois testes mais razoáveis, baseados em prioridades dadas às transações, seriam :ol atomic. Versão não-preemptiva: deixe T_i esperar por T_j se e somente se T_j tiver prioridade menor do que T_i; caso contrário cancele T_i. Versão preemptiva: deixe T_i esperar por T_j se e somente se T_j tiver prioridade maior do que T_i; caso contrário cancele T_j.

Estes testes garantem a ausência de bloqueios mútuos. De fato, considere a versão não-preemptiva. Se houvesse um ciclo no grafo de espera, haveria uma transação com prioridade maior do que ela mesma (pois uma transação só espera por outra com prioridade menor). Para o caso da versão preemptiva, o raciocínio é o mesmo, exceto que uma transação só espera por outra com prioridade maior.

No entanto, como não há restrições sobre a forma de associar prioridades às transações, o teste não garante que uma transação termine: ela poderá ser continuamente cancelada. Um esquema que evitaria este problema seria definir a prioridade de uma transação como a data/hora em que a transação foi submetida. A transação com menor data/hora é considerada como a de maior prioridade ou a transação mais *velha* do sistema. Supondo que duas transações não são submetidas no mesmo instante, cada transação receberá uma prioridade única.

Este esquema, acoplado com qualquer um dos testes anteriores, garante que toda transação sempre termina. De fato, ambos os testes garantem que a transação mais velha (de mais alta prioridade) no sistema sempre termina e que toda transação, em um espaço finito de tempo, se tornará a transação mais velha ativa no sistema (pois as mais velhas sempre vão terminando).

Técnicas preemptivas requerem um cuidado adicional. Caso a transação já tenha sido confirmada, ou seja, caso uma decisão já foi tomada para instalar as modifiçãoes produzidas pela transação no banco de dados, a transação não poderá ser cancelada. Para evitar este problema, deve-se garantir que, ao atingir esta fase, a transação detenha todos os bloqueios que precisa. Assim, não esperará por nenhuma outra transação, o que implica em que não

participa de nenhuma seqüência de impasse e, portanto, irá terminar normalmente.

8.3.3 Protocolo de Bloqueio em Duas Fases

Considere agora o problema de usar bloqueios para criar um método de controle de concorrência correto, ou seja, que garanta que, para toda execução concorrente *E* permitida pelo método

1) todas as transações iniciadas em *E* terminam;

2) *E* é serializável.

No caso do uso de bloqueios, violações da primeira condição resultam da criação de bloqueios mútuos ou do cancelamento repetido da mesma transação, o que já foi discutido na seção anterior. Consideraremos, portanto, este problema como resolvido, concentrando a atenção no problema de serialização.

O uso de bloqueios por si só não é suficiente para atingir serialização. Por exemplo, considere o seguinte protocolo:

1) bloqueie cada objeto antes de acessá-lo;

2) libere cada objeto imediatamente após acessá-lo.

Este protocolo é obviamente incorreto pois permitiria a criação de qualquer intercalação das ações das transações. Para transformar qualquer escalonamento sem bloqueios em um escalonamento satisfazendo a este protocolo, basta envolver cada ação de leitura ou atualização $O_i(x_1 ,..., x_k)$ entre as ações $B_i(x_1) ,..., B_i(x_k)$ e $L_i(x_1 ,..., x_k)$.

Apresentaremos nesta seção um exemplo de um protocolo baseado em bloqueios que garante serialização, cuja correção é provada na seção seguinte. O protocolo chama-se *bloqueio em duas fases* e é definido da seguinte forma:

1) Cada transação deverá bloquear cada objeto antes de acessá-lo e liberar todos os objetos que bloqueou até terminar;

2) Uma vez que uma transação liberar um objeto, não mais poderá bloquear outros objetos daí em diante.

O nome deste protocolo advém do fato de que, para cada transação, há uma primeira fase em que os objetos que a transação precisa são gradualmente bloqueados e uma segunda fase em que todos os objetos são gradualmente liberados. O ponto do escalonamento em que se dá a liberação do primeiro objeto da transação é chamado de *ponto de bloqueio* da transação.

Adotando a representação de uma execução através de escalonamentos com as ações *R(X)*, *W(X)*, *B(x)* e *L(X)*, um escalonamento modelando uma execução que satisfaz ao protocolo de bloqueio em duas fases seria:

$$L = B_1(x)\ R_1(x)\ B_2(y)\ R_2(y) W_1(x)\ L_1(x)\ B_2(x)\ L_2(y)\ W_2(x)\ L_2(x)$$

O escalonamento abaixo, por sua vez, viola o protocolo de bloqueio em duas fases:

$$M = B_1(x)\ R_1(x)\ B_2(y)\ R_2(y)\ L_1(x)\ B_2(x)\ L_2(y)\ W_2(x)\ L_2(x)\ B_1(x)\ W_1(x)\ L_1(x)$$

Note que, em *M*, a transação T_1 libera *x* após $R_1(x)$, voltando à bloqueá-lo antes de $W_1(x)$, o

que constitui uma violação da condição 2 do protocolo. Note ainda que L é serializável, mas não F.

Como último exemplo, considere o seguinte escalonamento (sem bloqueios e liberações):

$$N = R_1(x)\ W_2(x)\ W_1(x)\ W_3(x)$$

Este escalonamento é serializável por ser equivalente a

$$S = R_1(x)\ W_1(x)\ W_2(x)\ W_3(x)$$

mas N não satisfará ao protocolo de bloqueio em duas fases, para qualquer adição de bloqueios / liberações. Logo as condições impostas pelo protocolo não são necessárias para serialização (na seção seguinte mostraremos que são suficientes).

Uma implementação centralizada deste protocolo será discutida mais adiante, deixando para a Seção 8.4 a apresentação de implementações distribuídas. Antes, porém, convém provar a correção do protocolo.

*8.3.4 Correção do Protocolo de Bloqueio em Duas Fases

Mostraremos nesta seção que toda execução concorrente seguindo o bloqueio em duas fases é serializável, se todas as transações terminam. Ou seja, o problema de terminação é suposto resolvido por outros métodos.

Recorde que o banco é centralizado e que o ponto de bloqueio de uma transação é o ponto do escalonamento em que se dá a primeira ação $L(x)$ da transação. Seja E uma execução concorrente de um conjunto T de transações modelada por um escalonamento L. Suponha que todas as transações terminem em E e que elas sigam o protocolo de bloqueio em duas fases.

Defina uma relação $\rightarrow$ em T tal que $T_i \rightarrow T_j$ se e somente se o ponto de bloqueio de T_i precede o ponto de bloqueio de T_j. Provaremos que

(1) $\rightarrow$ é uma relação de ordem total em T.

De fato, como E induz uma ordem total das ações das transações, em particular, induz uma ordem total para o conjunto das ações $L(X)$ que primeiro foram executadas pelas transações. Esta última, por sua vez, gera a relação $\rightarrow$ sobre T.

Considere agora a relação de precedência por conflito, $<$, sobre T induzida por L. Mostraremos agora que

(2) se $T_i < T_j$ então $T_i \rightarrow T_j$

Suponha que $T_i < T_j$. Então existem ações $O_i(X)$ e $O_i(X)$ de T_i e T_j, respectivamente, tais que $O_i(X)$ e $O_j(Y)$ conflitam e $O_i(X)$ precede $O_j(Y)$. Logo, existe um objeto $x \in X \cap Y$. Pela condição 1 do protocolo de bloqueio em duas fases, uma ação $B_i(x)$ de T_i terá que preceder $O_i(X)$ e, da mesma forma, uma ação $B_j(x)$ de T_j terá que preceder $O_i(X)$. Pelo protocolo de bloqueio / liberação de objetos, uma ação $L_i(Z)$ de T_i tal que $x \in Z$ terá que suceder $O_i(X)$ e preceder $B_j(x)$. Por definição, o ponto de bloqueio de T_i terá que preceder ou coincidir com $L_i(Z)$. Mas, pela condição 2 do protocolo de bloqueio em duas fases, o ponto de bloqueio de T_j terá que suceder $B_j(x)$. Logo, por transitividade, o ponto de bloqueio de T_i precede o ponto de bloqueio de T_j em L. Assim, $T_i \rightarrow T_j$.

Finalmente, como, por (1), $\rightarrow$ é uma relação de ordem total, temos que $<$ é uma relação de ordem parcial. Logo pelo Teorema 1 da Seção 8.2.4, a execução *E* é serializável, como se queria demonstrar.

8.3.5 Uma Implementação Centralizada do Protocolo de Bloqueio em Duas Fases

Esta seção apresenta uma implementação do protocolo de bloqueio em duas fases que é completamente transparente aos usuários. Assumiremos que o modelo de processamento de transações adotado no caso centralizado é uma simplificação daquele descrito na Seção 8.1.2. Ou seja, as modificações a serem efetuadas nos objetos são mantidas em uma área de trabalho até que a transação complete o processamento. Neste ponto a transação poderá cancelar, descartando-se as modificações, ou terminar corretamente, sendo gerado então uma ação *W(x)* para instalar todas as modificações no banco de dados de forma atômica. Neste cenário, uma implementação possível de bloqueio em duas fases seria

1) as ações de leitura *R(x)* implicitamente geram uma ação prévia de bloqueio *B(x)*, para cada $x \in X$;

2) se a transação foi processada normalmente:

   a) todos os objetos que tiveram seu valor modificado pela transação são bloqueados antes que a ação de atualização final seja executada.

   b) após a ação *W(x)* ser executada, todos os objetos acessados pela transação são liberados.

3) se a transação é cancelada, todos os objetos que mantinha bloqueados são liberados.

Note que nesta implementação os objetos são mantidos bloqueados até o final da transação, ou seja, o ponto de bloqueio se dá ao final da transação. Esta implementação é coerente com o cenário previsto nos últimos capítulos em que uma transação pode ser cancelada durante a execução. De fato, suponha que a liberação de objetos cujo valor foi alterado seja permitida antes da transação terminar. Se a transação for cancelada após liberar um objeto que alterou, todas as outras transações que leram aquele valor teriam que ser canceladas também e as suas modificações desfeitas, mesmo que já tivessem terminado, e assim recursivamente. Ou seja, o cancelamento de uma transação poderia provocar cancelamentos em cascata de outras transações.

Isto conclui a discussão sobre o protocolo de bloqueio em duas fases centralizado.

8.4 MÉTODOS BASEADOS EM BLOQUEIOS - PARTE II

Esta seção discute implementações do protocolo de bloqueio em duas fases e algoritmos para deteção de bloqueios mútuos em um ambiente de bancos de dados distribuídos. As implementações diferirão essencialmente no posicionamento da tabela de bloqueios ao longo da rede. O argumento de correção destas implementações segue diretamente da prova de correção do protocolo de bloqueio em duas fases para o caso centralizado e, portanto, será omitido.

Em todas as implementações assume-se que as transações são processadas conforme descrito na Seção 8.1.2. Em particular, todas as modificações são armazenadas em uma área de trabalho até que a transação execute um comando FIM-DE-TRANSAÇÃO, quando o protocolo

bifásico para confirmar intensões é invocado para instalar as modificações criadas pela transação, ou rejeitá-las, cancelando a transação.

8.4.1 Implementação Básica

Por implementação básica do protocolo de bloqueio em duas fases em um ambiente distribuído entenderemos aquela em que a tabela de bloqueios é distribuída junto com os dados. Mais precisamente, para cada nó onde há um banco de dados local, haverá também uma tabela de bloqueios para controle do acesso aos objetos locais. Esta tabela é implementada como no caso centralizado e gerenciada por uma cópia local do protocolo de bloqueio/liberação de objetos. Desta forma, se os SGBDs locais já usavam uma técnica de bloqueios, nada mais é necessário fazer para controle de concorrência do banco distribuído, exceto deteção de bloqueios mútuos.

Os pedidos de bloqueio/liberação são gerados automaticamente dentro do seguinte esquema:

1. um bloqueio *B(x)* é criado imediatamente antes de uma leitura *R(X)* ser processada localmente, para cada $x \in X$;

2. um bloqueio *B(x)* é criado imediatamente após uma mensagem *PREPARE_SE* ser recebida na primeira fase do protocolo bifásico, para cada objeto *x* que reside localmente e cujo valor foi modificado pela transação;

3. uma liberação *L(X)* é criada imediatamente após uma mensagem *PREPARE_SE* ser recebida na primeira fase do p localmente e cujos valores não foram modificados pela transação;

4. uma liberação *L(X)* é criada após as modificações terem sido instaladas no banco, caso o nó tenha recebido uma mensagem *CONFIRME*, ou após o nó ter recebido uma mensagem *CANCELE*, onde *X* é o conjunto dos objetos que residem localmente e cujos valores foram modificados pela transação;

É interessante observar que a implementação acima não toma conhecimento da existência de cópias. Se porventura os objetos x_1 ,..., x_k representam cópias do mesmo dado em nós diferentes, caberá ao processador de comandos da LMD e ao gerente de transações providenciar para que todas as cópias sejam atualizadas. Para cada nó onde reside uma cópia, o novo valor será enviado durante o protocolo bifásico. Assim sendo, a existência de cópias torna-se transparente ao mecanismo de controle de concorrência.

8.4.2 Implementação por Cópias Primárias

A implementação distribuída de certa forma desperdiça recursos locais por bloquear todas as cópias de um mesmo objeto lógico. Se recursos locais são escassos e é vantajoso economizá-los em troca de um maior tráfego de mensagens, pode-se optar pela implementação usando cópias primárias. A idéia é simples. Suponhamos que os objetos físicos estejam particionados de tal forma que os objetos em cada partição representem cópias dos mesmos dados. Para cada partição, designe um objeto físico como a *cópia primária* daquela partição. Antes de acessar qualquer objeto físico em uma dada partição, a cópia primária correspondente deverá ser bloqueada.

Os pedidos de bloqueio/liberação são gerados automaticamente dentro do seguinte esquema:

1. Antes de uma ação $R_k(X)$ da transação T_k ser processada localmente em *i*, uma mensagem para bloquear cada objeto $x \in X$ é enviada ao nó *j* que mantém a cópia primária x_p correspondente à *x*. O nó *j* tenta bloquear x_p para T_k e, após obter sucesso, envia uma mensagem ao nó *i* confirmando o bloqueio. A ação $R_k(X)$ espera até que todas as cópias tenham sido bloqueadas.

2. um bloqueio *B(x)* é criado imediatamente após uma mensagem *PREPARE_SE* ser recebida na primeira fase do protocolo bifásico, para cada cópia *x* cujo valor foi modificado pela transação (como todas as cópias de cada objeto tem que ser igualmente alteradas pela transação, a cópia primária x_p correspondente a *x* será automaticamente bloqueada para a transação, não havendo necessidade de bloqueá-la explicitamente);

3. uma liberação *L(X)* é criada imediatamente após uma mensagem *PREPARE_SE* ser recebida na primeira fase do protocolo bifásico, onde *X* é o conjunto de todas as cópias primárias que residem localmente e cujos valores não foram modificados pela transação;

4. uma liberação *L(X)* é criada após as modificações terem sido instaladas no banco, caso o nó tenha recebido uma mensagem *CONFIRME*, ou após o nó ter recebido uma mensagem *CANCELE*, onde *X* é o conjunto de todas as cópias primárias que residem localmente e cujos valores foram modificados pela transação;

Note que, ao bloquear apenas a cópia primária, o processamento local em cada nó tende a diminuir pois menos bloqueios são realizados. Porém, para se ler a cópia armazenada em *i*, uma mensagem extra deverá ser enviada ao nó *j* (exceto se *i=j*). Portanto, esta implementação gera um tráfego maior na rede apenas para controle de concorrência.

A última observação adquire maior peso se a granularidade dos objetos físicos for pequena. Por exemplo, se os objetos físicos são páginas, para cada página a ser lida uma mensagem teria que ser enviada ao nó que contém a cópia primária daquela página, o que é inaceitável. Uma forma de resolver este problema seria agrupar vários pedidos de bloqueio para um mesmo nó em uma só mensagem. Uma segunda solução seria adotar bloqueio hierarquizado. Por exemplo, segmentos inteiros seriam bloqueados em lugar de páginas. De qualquer forma, o conceito de "cópia" deve estar bem definido na arquitetura do sistema.

8.4.3 Implementação por Bloqueio Centralizado

Em ambas as implementações anteriores, a tabela de bloqueios também é distribuída ao longo da rede. Esta opção torna a deteção de bloqueios mútuos mais difícil pois, para se construir o grafo de espera, será necessário consultar todos os nós em que uma parte da tabela está armazenada. Se bloqueios mútuos são muito frequentes e é necessário detetá-los e resolvê-los rapidamente, uma terceira implementação alternativa torna-se atraente.

Na implementação por bloqueio centralizado, elege-se um *nó coordenador* da rede para conter toda a tabela de bloqueios. Para qualquer objeto que for acessado, uma mensagem de bloqueio deverá ser enviada ao nó coordenador, que responderá então ao nó que solicitou o bloqueio. A implementação é bastante semelhante à do bloqueio por cópias primárias e será omitida.

Esta implementação oferece como vantagem, conforme já mencionado, a facilidade de deteção/resolução de bloqueios mútuos pois toda a tabela de bloqueios reside em um único nó. Assim, a construção do grafo de espera pode ser feita localmente. Por outro lado, esta implementação apresenta os mesmos problemas da implementação baseada em cópias

primárias, acrescidos de dois outros. Primeiro, o tráfego adicional de mensagens para controle de concorrência é canalizado para o nó coordenador, gerando uma sobrecarga localizada na rede. Segundo, e mais grave, a implementação é muito vulnerável a falhas envolvendo o nó coordenador. Se a tabela de bloqueios for perdida ou o nó coordenador por algum motivo não puder ser contactado, todas as transações correntes terão que ser canceladas e um protocolo de eleição do novo nó coordenador deverá ser completado antes do processamento normal ser reiniciado. Isto significa que várias das vantagens advindas do uso de um banco de dados distribuído simplesmente perdem o sentido nesta implementação.

8.4.4 Tratamento de Bloqueios Mútuos no Caso Distribuído

O tratamento de bloqueios mútuos no caso distribuído é muito semelhante ao caso centralizado. As técnicas preventivas discutidas para o caso centralizado também se aplicam ao caso distribuído. A única observação adicional se refere à geração de prioridades através da data/hora em que a transação foi submetida. Como há vários nós, duas transações poderão receber a mesma prioridade em nós diferentes dentro deste esquema. Uma solução consagrada consiste em adotar o par *(n,d)* como a prioridade da transação, onde *n* é o número do nó onde ela foi submetida (assume-se que dois nós não têm o mesmo número) e *d* é a data/hora em que a transação foi submetida. Uma transação que recebeu o par *(n,d)* terá prioridade maior que uma transação que recebeu o par *(n',d')* se e somente se $n < n'$ *ou* $n = n'$ *e* $d < d'$.

Já a deteção/resolução de bloqueios mútuos, quando se opta pela implementação básica ou pela implementação através de cópias primárias, merece comentários especiais. O problema básico está em que cada tabela local de bloqueios gera apenas um subgrafo do grafo de espera. Naturalmente cada um destes subgrafos poderá ser acíclico sem que o grafo completo o seja. Em outros termos, não é possível fazer deteção de bloqueios mútuos apenas localmente. O resto desta seção discute duas formas de implementar deteção de bloqueios mútuos neste caso.

Seja *N* um conjunto de nós. Chamaremos de *subgrafo de espera local a N* ao subgrafo do grafo de espera induzido pelas tabelas de bloqueios residentes em nós pertencentes a *N*. Ao grafo completo chamaremos de *grafo de espera global*. Similarmente, chamaremos de *bloqueio mútuo local a N* a um bloqueio mútuo gerado por um ciclo no subgrafo de espera local a *N*. Chamaremos de *bloqueio mútuo global* a um bloqueio mútuo gerado por um ciclo do grafo de espera global.

A implementação mais simples consistiria em, periodicamente, cada nó *i* que contém uma tabela de bloqueios construir o subgrafo de espera local a *i* e enviá-lo para um nó central designado. O nó central construiria então o grafo de espera global, detetando e resolvendo bloqueios mútuos como no caso centralizado.

O método anterior na verdade induz uma árvore de altura 2, cuja raiz é o nó central e cujas folhas são os outros nós. Uma segunda implementação, que chamaremos de algoritmo *hierárquico*, generaliza esta observação. Supõe-se inicialmente que os nós da rede estejam logicamente organizados em uma árvore (para propósitos do algoritmo apenas). Por exemplo, os nós de um mesmo município são todos filhos de um mesmo *nó municipal*, os nós municipais em um mesmo estado são por sua vez filhos de um mesmo *nó estadual* e assim por diante. Periodicamente (e sincronamente), cada folha *f* constrói o subgrafo de espera local a *f*, baseando-se na tabela de bloqueios local, e tenta detetar e resolver bloqueios mútuos locais a *f*; em seguida envia o subgrafo para o seu pai. Um nó interior *n*, ao receber os

subgrafos dos seus filhos, e após construir o seu próprio subgrafo local, faz a união de todos estes subgrafos e tenta detetar bloqueios mútuos locais a *N*, onde *N* é o conjunto dos nós da subárvore cuja raiz é *n*; em seguida, envia o subgrafo consolidado para seu pai e assim por diante até a raiz.

8.5 MÉTODOS BASEADOS EM PRÉ-ORDENAÇÃO

Esta seção discute controle de concorrência por pré-ordenação para bancos de dados distribuídos. Inicialmente o protocolo genérico é apresentado e em seguida várias implementações discutidas.

8.5.1 Protocolo de Pré-Ordenação

Como o nome indica, neste protocolo uma ordem é imposta "a priori" às transações, e deve ser respeitada pela execução concorrente das transações. Mais precisamente, o *protocolo de pré-ordenação* opera da seguinte forma:

1. Cada transação ao ser iniciada recebe uma *senha* ou *número de protocolo*, única ao longo da rede, de forma transparente aos usuários.

2. Em cada nó há um mecanismo de controle de concorrência local que garante que as ações conflitantes (veja Seção 8.2.4) geradas pelas transações são processadas em ordem de senha.

A correção deste protocolo é imediata. Seja *E* uma execução concorrente de um conjunto *T* de transações. Suponha que *E* tenha seguido o protocolo de pré-ordenação. Seja :f/lt/ a relação de precedência por conflito sobre *T* induzida por *E*. Como as ações conflitantes são executadas em *E* na ordem de senha, temos que se $T_i < T_j$ então a senha de T_i é menor do que a senha de T_j. Logo, < é necessariamente uma relação de ordem parcial sobre o conjunto das transações (pois as senhas impõe uma ordem total às transações), o que implica que *E* é serializável.

O protocolo de pré-ordenação age então de forma completamente diferente do protocolo de bloqueio em duas fases pois, no primeiro, a ordem de serialização das transações é imposta "a priori", enquanto que no segundo é imposta "a posteriori". Esta observação está clara no que concerne ao protocolo de pré-ordenação. Para compreendê-la melhor no caso de bloqueio em duas fases, recordemos que a prova de correção deste indica-nos que toda execução concorrente *E* seguindo o bloqueio em duas fases é sempre serializável e que a execução serial *S* equivalente a *E* é obtida processando-se as transações em ordem dos seus pontos de bloqueio (em *E*). Assim, na execução concorrente *E*, os usuários do sistema tem a ilusão de que tudo se passa como se as transações fossem executadas seqüencialmente na ordem em que atingiram os seus pontos de bloqueio. Porém, esta ordem não é imposta "a priori", dependendo da própria dinâmica das transações e da intercalação mais ou menos fortuita das ações elementares sobre os bancos locais.

É interessante fazer também uma analogia entre o protocolo de pré-ordenação e uma disciplina às vezes usada em estabelecimentos (bem organizados) em que o cliente, ao chegar, apanha uma senha e aguarda a vez para ser atendido. Se houver apenas um empregado e o cliente for atendido sem interrupção, a situação se torna equivalente a permitir apenas execuções seriais. O caso interessante acontece quando vários empregados atendem a

vários pedidos de vários clientes ao mesmo tempo. A disciplina de senhas (isto é, o próprio protocolo de pré-ordenação) deve necessariamente harmonizar todo o trabalho de tal forma que o cliente com senha *i* tenha sempre a ilusão de que foi atendido antes do cliente com senha *j*, se $i<j$ (e, portanto, não reclame do aparente caos reinante).

As próximas subseções desta seção discutirão implementações alternativas do protocolo de pré-ordenação em um ambiente distribuído.

8.5.2 Implementação Básica

Na implementação básica, o protocolo opera exatamente como descrito na seção anterior.

A implementação do primeiro passo do protocolo exige que a geração de senhas ao longo da rede seja tal que duas transações não recebam o mesmo número de senha. Para tal, conforme discutido na Seção 8.4.4., poderemos usar como senha o par *(n,d)* onde *n* é o número do nó onde a transação se originou e *d* é a data/hora em que a transação foi submetida.

A implementação do segundo passo é bem mais difícil pois requer construir um mecanismo de controle de concorrência local que garanta que as ações conflitantes são processadas em ordem de senha. A seguinte estratégia poderia ser usada neste caso. Para cada objeto físico *x* do sistema são mantidas duas variáveis:

R_senha(x) senha da última operação *R(X)* que acessou o objeto

W_senha(x) senha da última operação *W(X)* que acessou o objeto

O mecanismo de controle de concorrência que controla o entrelaçamento das ações elementares em um dado nó seria o seguinte:

1) se a operação é uma leitura *R(X)* com número de senha *s* então:
   a) faça *w* igual ao maior *W_senha(x)* para os objetos $x \in X$.
   b) se $w<s$, então processe a leitura e faça *R_senha(x)* = *s*, para cada $x \in X$.
   c) caso contrário, rejeite a leitura e reinicie a transação.
2) se a operação é uma atualização *W(X)* com número de senha *s* então:
   a) faça *r* igual ao maior valor de *R_senha(x)* para os objetos $x \in X$.
   b) faça *w* igual ao maior valor de *W_senha(x)* para os objetos $x \in X$.
   c) se $max(r,w)<s$, então processe a atualização, e faça *W_senha(x)* = *s*, para cada $x \in X$.
   d) caso contrário, rejeite a atualização e reinicie a transação.

As transações reiniciadas recebem um número de senha maior do que antes.

Esta implementação corretamente processa ações conflitantes em ordem de senha. De fato, seja *E* uma execução seguindo esta implementação e modelada por um escalonamento global *L*. Sejam O_i e O_j ações em um escalonamento local de *L* com senhas s_i e s_j. Suponha que O_i precede O_j e que O_i conflita com O_j. Seja *x* um objeto acessado por O_i e O_j (*x* existe pois estas ações conflitam). Suponha que O_i é uma leitura (logo O_j tem que ser uma atualização). Como O_i precede O_j, temos como resultado dos testes que $s_i < R\_senha(x) < s_j$. Logo, O_i e O_j foram

processadas em ordem de senha, como se queria demonstrar. O caso em que O_i é uma atualização é análogo.

Há três problemas ainda com esta implementação: armazenamento das variáveis *R_senha* e *W_senha*, relacionamento com o protocolo bifásico para confirmar intensões, e problemas de terminação.

Para o primeiro destes problemas, duas soluções são sugeridas. Suponhamos, inicialmente, que os objetos físicos sejam páginas. A primeira solução consiste em armazenar as variáveis *R_senha(x)* e *W_senha(x)* diretamente na página *x*. Esta solução força o protocolo a ler cada página $x \in X$ apenas para obter o valor de *R_senha(x)* e *W_senha(x)*, mesmo que a ação *R(X)* ou *W(X)* venha a ser rejeitada. Se a percentagem de ações rejeitadas for baixa, este custo adicional será pequeno no cômputo total, pois os acessos às páginas são de qualquer forma necessários ao próprio processamento das ações.

Uma segunda solução para o primeiro problema seria manter em uma tabela à parte um certo número de valores das variáveis *R_senha(x)* e *W_senha(x)* e "esquecer" os valores mais antigos. Desta forma, memória adicional seria economizada e os valores das variáveis estariam disponíveis sem acessar as próprias páginas. Naturalmente que não é possível simplesmente esquecer valores de senha, o que força a usar um esquema mais elaborado.

Mais precisamente, a solução proposta baseia-se nas seguintes estruturas de dados:

R_tabela	contém pares *(x,R_senha(x))* para os objetos *x* mais recentemente usados
W_tabela	contém pares *(x,W_senha(x))* para os objetos *x* mais recentemente usados
R_max	maior valor de *R_senha(x)* que foi expurgado de *R_tabela*
W_max	maior valor de *W_senha(x)* que foi expurgado de *W_tabela*

As tabelas são gerenciadas da seguinte forma. Quando um objeto *x* é lido, um par *(x,s)* é adicionado a *R_tabela*, onde *s* é a senha da ação de leitura. Se já houver um par *(x,s')* em *R_tabela* inicialmente, *s*' é simplesmente substituído por *s*. Se *R_tabela* estiver cheia, um par *(y,t)* é selecionado de acordo com alguma política como, por exemplo, selecionar sempre o par que foi referenciado pela última vez há mais tempo. O par *(y,t)* é expurgado da tabela, dando lugar a *(x,s)*.

No entanto, o par *(y,t)* não pode ser abandonado sem qualquer cuidado adicional, pois a senha *t* de *y* é necessária para o protocolo de pré-ordenação. Para contornar este problema, a variável *R_max* é mantida. Assim, ao expurgar *(y,t)*, faz-se *R_max := max(R_max,t)* .

A implementação do protocolo é modificada da seguinte forma. Para se obter o valor de *R_senha(x)*, pesquisa-se a *R_tabela* primeiro. Se existir um par *(x,r)* em *R_tabela*, *r* é tomado como o valor de *R_senha(x)*. Caso contrário, toma-se *R_max* como o valor de *R_senha(x)*. Note que o valor verdadeiro de *R_senha(x)* neste segundo caso é menor ou igual a *R_max*. O tratamento de *W_senha(x)* é semelhante.

Assim sendo, os testes aplicados na implementação corretamente garantirão que ações conflitantes são processadas em ordem de senha. Porém, algumas ações serão rejeitadas desnecessariamente pois *R_max* e *W_max* são uma estimativa conservativa de *R_senha* e *W_senha*. A solução sugerida aqui representa então um balanço entre gastar menos memória para armazenar *R_senha* e *W_senha* ou reiniciar transações com mais freqüência.

Passemos agora para o segundo problema, o relacionamento com o protocolo bifásico.

Sabemos que durante a primeira fase do protocolo bifásico, mensagens de *PREPARE_SE* são enviadas para cada nó participante do processamento da transação. Desta forma o protocolo bifásico determina se a transação deve ser aceita ou cancelada. Se aceita, mensagens *CONFIRME* são enviadas em seguida para que os nós atualizem o banco de dados através de ações de atualização *W(X), que necessariamente ter&aoi. que ser executadas*. Portanto, a decisão de aceitar ou rejeitar *W(X)* (implicando em votar *SIM* ou *NÃO*) deverá ser tomada quando a mensagem de *PREPARE_SE* chegar e não quando *W(X)* for efetivamente processada. Para tal, o mecanismo de controle de concorrência local deve ser modificado para fazer os testes necessários à aceitação de *W(X)* quando a mensagem de *PREPARE_SE* (que deverá vir com o número de senha e o conjunto *X*) for recebida, e não ao processar *W(X)*.

Além disto, uma vez que um nó votou *SIM* aceitando a transação, ele terá que garantir que *W(X)* será necessariamente executada. Isto significa que nenhuma ação *R(Y)* ou *W(Z)* que invalide a decisão tomada ao processar *PREPARE_SE* poderá ser executada entre receber *PREPARE_SE* e processar *W(X)*. Esta regra pode ser implementada mudando-se *W_senha(x)* para $\propto$, para todo $x \in X$, quando *PREPARE_SE* for aceito, e atualizando-se *W_senha(x)* para o valor correto depois de processar *W(X)*, para todo $x \in X$. Com *W_senha(x)* = ∞, para todo $x \in X$, todas as ações que conflitam com *W(X)* serão rejeitadas. Uma outra forma de implementar a regra seria combinar bloqueios com as senhas de tal forma a evitar que ações com número de senha maior do que a senha de *W(X)* e que conflitem com *W(X)* sejam processadas indevidamente entre *PREPARE_SE* e *W(X)*. Tais ações ficariam bloqueadas até que *W(X)* fosse executada, em lugar de serem rejeitadas como na solução anterior. Na verdade, estas duas soluções poderão ser combinadas usando mudança de senha para bloquear ações.

Finalmente é necessário examinar o problema de terminação. No contexto desta implementação uma transação T_i poderá não terminar se for *reiniciada ciclicamente*. De fato, neste protocolo, erros de sincronização ocorrem quando uma ação chegou "atrasada", por assim dizer, com relação a outra ação com que conflita. A única forma de corrigir um erro é reiniciar a transação T_i com uma senha maior do que a anterior (se a transação for repetida com a mesma senha fatalmente será reiniciada novamente).

A solução para este problema consiste em reiniciar a transação T_i com uma nova senha tal que seja garantidamente maior do que a senha de qualquer outra transação processada concorrentemente com T_i. Desta forma, todas as ações de T_i passarão pelos testes do protocolo o que significa que T_i irá necessariamente terminar. No entanto, como não é simples impor que a senha de T_i possua esta propriedade em um ambiente distribuído, é preferível apenas incrementar a senha de T_i. suficientemente para aumentar a sua chance de terminar. Dentro do método de gerar senhas proposto nesta seção, isto significa adiantar o relógio local do nó onde a transação foi submetida.

A política de aumento de senha ainda tem um outro fator determinante. Existe a possibilidade do processamento de várias transações sincronizar-se de tal forma a causar o *reinício mútuo* de todas elas. Ou seja, cria-se uma seqüência de transações $T_{i\,0}$,$T_{i\,1}$,..., $T_{i\,m-1}$,$T_{i\,0}$ tal que $T_{i\,j}$ força o reinício de $T_{i\,j+1}$ (soma módulo *m*). Se o aumento da senha for o mesmo para todas estas transações, corre-se o risco de se repetir a situação indefinidamente. Note que esta é a contra-partida de bloqueio mútuo quando a forma de corrigir erros de sincronização é baseada no reinício de transações. No entanto, existe uma forma simples de minimizar a chance de uma situação de reinício mútuo se formar: basta randomizar o incremento dado às senhas quando reiniciar transações. Este esquema simples naturalmente não evita completamente o problema de reinício cíclico, mas o torna pouco provável.

8.5.3 Implementação Conservativa

Uma segunda forma de implementar o protocolo de pré-ordenação, desta feita bem simples, seria processar as ações localmente em ordem de senha, quer elas conflitem ou não, mas de tal forma que uma transação nunca seja rejeitada. Como não há necessidade de detetar conflitos, o controle de concorrência pode ser feito a nível lógico ou físico com igual simplicidade. Por esta razão esta será a única implementação descrita a nível lógico, ou seja, a nível dos subcomandos atuando sobre os objetos lógicos, embora possa ser convertida para atuar a nível físico, ou seja, a nível das ações elementares como todas as outras.

Nesta implementação, em cada nó *i* onde o banco é armazenado e para cada gerente de transações *g* que envia subcomandos para *i* são mantidas duas filas:

R_fila(g) fila contendo os subcomandos que apenas lêem do banco local enviados por *g* para o nó *i*;

W_fila(g) fila contendo os subcomandos que modificam o banco local enviados por *g* para o nó *i*;

Exige-se que um gerente de transações *g* envie os subcomandos para as filas *R_fila(g)* e *W_fila(g)* em ordem de senha. Localmente, o mecanismo de controle de concorrência procederia da seguinte forma:

1. espere até que todas as filas contenham algum comando;

2. escolha o subcomando com menor senha e processe-o completamente.

A prova de correção desta implementação é muito simples. Como todas as ações são processadas em ordem de senha, em particular as que conflitam o são. Logo, caímos no caso geral do método de pré-ordenação.

Fazendo uma breve comparação com a implementação básica, a implementação conservativa é mais simples e, ao contrário da implementação básica, nunca força transações a serem reiniciadas. Por outro lado, implicitamente bloqueia transações pois sempre garante que a escolha de um comando se deu na ordem correta, quer ele gere conflitos ou não (daí o nome de implementação conservativa).

O problema levantado na implementação anterior acerca do relacionamento com o protocolo bifásico para confirmar intensões também ocorre nesta implementação, sendo resolvido de forma semelhante. Além deste, a implementação gera um tipo diferente de problema de terminação e causa problemas de comunicação.

O protocolo, conforme descrito, pode ficar paralizado em um nó *i* simplesmente porque um gerente de transações nunca manda nenhum comando para *i*, o que pode bloquear transações indefinidamente. Para resolver este problema os gerentes de transações deverão periodicamente enviar *mensagens de controle* a todos os nós com que se comunicam indicando a senha corrente que pretendem dar às suas transações (que no esquema desta seção é o par *(n,d)*, onde *n* é o número do nó onde a transação se originou e *d* é a data/hora em que a transação foi submetida).

Esta solução, por sua vez, cria a necessidade de comunicação adicional entre os gerentes de transações e nós onde o banco está armazenado, que pode se tornar proibitiva em uma rede de grandes proporções. Para solucionar este último problema, um gerente de transações *g* poderá enviar uma mensagem de controle a um nó *i* contendo uma senha *n* maior do que a sua

corrente indicando que não pretende mandar para *i* nenhum subcomando com senha menor do que *n*. Esta mensagem de controle não precisa ser repetida até que as senhas dadas por *g* cheguem a *n*. Naturalmente, deverá haver um mecanismo adicional para revogar esta decisão, caso *g* tenha necessidade de enviar a *i* um subcomando com senha menor do que *n*.

8.5.4 Implementação Baseada em Versões Múltiplas

Nas duas implementações até agora discutidas, uma ação de leitura é rejeitada sempre que chegar atrasada, ou seja, sempre que o valor do objeto que ela deveria ter lido já tiver sido substituído por um mais novo. Uma forma de minimizar o reinício de transações por este motivo seria manter uma série histórica com todas as *versões* do valor de cada objeto. Quanto maior a série, maior é a chance do valor que uma ação de leitura precisa estar disponível. Por outro lado, maior é o desperdício de memória por força do mecanismo de controle de concorrência.

Consideremos inicialmente o caso extremo em que a série histórica com todas as versões de cada objeto é mantida. As seguintes estruturas de dados serão então mantidas para cada objeto *x*:

R_seq(x) seqüência de todas as senhas das ações de leitura para o objeto *x*.

Versões(x) seqüência contendo todas as versões já criadas para *x*, representadas por pares da forma *(s,V)*, onde *s* é a senha da ação que criou a versão *V* de *x*.

De posse destas estruturas, o mecanismo de controle de concorrência local passa a ser o seguinte:

1) Suponha que a ação é uma leitura *R(X)* com senha *r*. Para cada $x \in X$, escolha o par *(s,V)* em *Versoes(x)* tal que *s* é a maior senha em *Versoes(x)* menor do que *r*.

   a) *V* será então o valor de *x* usado no processamento de *R(X)*.

   b) *r* será inserida em *R_seq(x)* na posição correta.

2) Suponha que a ação é uma atualização *W(X)* com senha *w*. Para cada $x \in X$, seja *t(x)* a menor senha em *Versoes(x)* maior do que *s*.

   a) se existe algum $x \in X$ e existe alguma senha *r* em *R_seq(x)* tal que $w < r \leq t(x)$, então rejeite a atualização;

   b) caso contrário, processe a atualização, criando um novo par *(w,V)* em *Versoes(x)*, onde *V* é o valor de *x* dado por *W(X)*, para cada $x \in X$.

Como este protocolo é mais sofisticado, convém dar um exemplo. Para efeitos do exemplo é conveniente imaginar que a senha dada a uma transação represente o instante em que foi submetida. Considere um banco de dados com apenas um objeto físico *x*. Suponha que em um dado momento *R_seq(x)* e *Versoes(x)* sejam:

R_seq(x) = (5,8,15,...,92)

Versoes(x) = ((4,V_1), (10,V_2), (20,V_3), ... , (100,V_{20}))

Suponha que o protocolo receba uma ação de leitura *R(x)* com senha *r=12*. Como temos que

10 < r < 20, o valor lido de *x* será V_2. Ou seja, apesar da ação ter chegada bastante "atrasada" (a versão corrente de *x* foi criada por uma transação com senha *100*), é possível encaixar a ação na ordem correta, fornecendo-lhe a versão corrente no instante em que foi submetida. A senha *r=12* é inserida em *R_seq(x)* criando-se *R_seq(x) = (5,8,12,15,...,92).*

Suponha agora que o protocolo receba uma ação de atualização *W(x)* com senha *w=7*. Esta ação terá que ser rejeitada pela seguinte razão. Observe *R_seq(x)* e *Versoes(x)*. Uma leitura *R(Y)* com senha *r=8* já foi processada (segundo elemento de *R_seq(x)*) e leu a versão V_1 criada por uma ação com senha *4* (primeiro elemento da seqüência *Versoes(x)*). Se o protocolo aceitasse a ação *W(x)* estaria criando uma nova versão V_1' com senha *7*, que deveria então ter sido lida por *R(Y)*. Ou seja, a ação *W(X)* seria processada depois de *R(Y)*, quando deveria ter sido processada antes. Logo, *W(x)* tem que ser rejeitada. Este fato pode ser detetado pesquisando-se a menor senha em *Versoes(x)*, *t=10* neste caso, maior do que *w=7*. Como há alguma senha em *R_seq(x)* entre *s=7* e *t=10*, *r=8* neste caso, *W(x)* terá que ser rejeitada.

Esta implementação tem a propriedade interessante de nunca rejeitar ações de leitura e diminuir a rejeição de ações de atualização. Em outras palavras, o volume de transações reiniciadas nesta implementação é necessariamente menor que o da implementação básica. Por outro lado, a memória adicional necessária a torna proibitiva: manter todas as versões de todos os objetos não é possível nem razoável para um banco de dados. Uma forma de tornar esta implementação atraente seria manter as últimas versões criadas dentro de um esquema semelhante àquele sugerido na implementação básica. Haveria uma tabela de versões (e de senhas de operacoes de leitura) mantendo um certo número de versões recentes, além do próprio banco de dados armazenando a última versão de cada objeto. Quando a tabela ficar completamente ocupada, espaço é obtido expurgando-se a entrada mais antiga. Assim, a um custo adicional de memória razoável, poder-se-ia diminuir o volume de transações reiniciadas, com benefício positivo para o sistema.

Por fim, lembramos que os problemas de terminação e relacionamento com o protocolo bifásico persistem nesta implementação, sendo resolvidos da mesma forma que antes.

8.6 COMPARAÇÃO ENTRE OS MÉTODOS

Nas seções anteriores, dois métodos básicos de controle de concorrência e várias implementações foram discutidas. Esta seção reúne comentários indicando em que situações uma ou outra implementação se torna mais adequada.

A Tabela 8.1 resume as principais características das implementações dos protocolos de bloqueio em duas fases e de pré-ordenação, considerando apenas bancos de dados distribuídos.

É extremamente difícil comparar efetivamente a performance das várias implementações sugeridas neste capítulo. Por outro lado, pode-se adiantar alguns comentários que, embora simples, ajudam a avaliar as diversas alternativas.

Tabela 8.1 - Resumo das Características dos Métodos de Controle de Concorrência

Implementação		***Características***			
		Estrutura de Dados	**Mensagens Adicionais**	**Existência de cópias**	**Problemas Terminação**
Bloqueio	*Básica*	tabela de bloqueios distribuída	não são necessárias	não reconhece	bloqueios mútuos difíceis de detectar
	Cópias Primárias	tabela de bloqueios para cópias primárias	para bloquear cópias primárias	reconhece	bloqueios mútuos difíceis de detectar
	Centralizada	tabela de bloqueios centralizada	para bloquear em um nó coordenador	não reconhece	bloqueios mútuos fáceis de detectar
Pré-Ordenação	*Básica*	listas de senhas	não são necessárias	não reconhece	reinício cíclico e mútuo, de fácil solução
	Conservativa	filas de subcomandos	para evitar bloqueios eternos	não reconhece	bloqueios eternos
	Versões Múltiplas	listas de senhas e versões	não são necessárias	não reconhece	reinício cíclico e mútuo, de fácil solução

Inicialmente, convém lembrar que a performance de um mecanismo de controle de concorrência pode ser avaliada através de várias medidas diferentes, das quais o tempo de resposta das transações será aqui utilizado. O tempo de resposta das transações é influenciado por parâmetros intrínsicos à população de transações - taxa média de chegada, número de objetos acessados para leitura e atualização, taxa média de serviço de processamento e de serviço de entrada/saída - e por parâmetros do próprio sistema - número de objetos do banco de dados, alocação dos objetos nos bancos locais, grau de multiprogramação, topologia do sistema, etc. No que concerne especificamente a mecanismos de controle de concorrência, os seguintes parâmetros tornam-se importantes:

custo adicional de comunicação, que pode ser estimado pelo número médio de mensagens passadas apenas para fins de controle de concorrência;

custo adicional de processamento local, que pode ser medido pelo tempo médio de processamento gasto apenas em controle de concorrência;

custo adicional de processamento das transações, estimado pelo tempo médio que uma transação passa bloqueada, ou pelo número médio de vezes que a transação é reiniciada;

Usaremos estes três parâmetros como indicativo da adequação de um mecanismo de controle de concorrência a dois cenários extremos:

pessimista: caracterizado por uma elevada percentagem de ações conflitantes

otimista: caracterizado por uma baixa percentagem de ações conflitantes

Não é muito claro, e não há muitos experimentos disponíveis para guiar a intuição, o que significa "baixa percentagem de ações conflitantes". Um indicador aceito considera como baixa uma percentagem de ações conflitantes que está abaixo de 5%. Se considerarmos que a freqüência de conflitos será muito baixa quando as transação acessam poucos objetos em um banco de dados de tamanho médio, este limite será satisfeito por uma vasta gama de aplicações. Por outro lado, uma alta percentagem de conflitos significa algo acima de 10%. Se a freqüência exceder este limite, o volume de bloqueios mútuos e reinícios de transações poderá ser tão elevado que o trabalho útil do sistema decairá consideravelmente.

Consideremos inicialmente o problema de escolher um mecanismo de controle de concorrência para um cenário pessimista. Dado que a percentagem de conflitos é alta, o objetivo básico neste caso será diminuir o custo de resolver conflitos. Como a forma mais dispendiosa de resolver conflitos é reiniciar transações, deve-se escolher um mecanismo que minimize o volume de transações reiniciadas.

Com estas características temos duas implementações do protocolo de pré-ordenação, a implementação conservativa e aquela utilizando versões múltiplas. A implementação conservativa nunca reinicia transações, mas gera um volume de mensagens adicionais substancial e implicitamente bloqueia transações com freqüência. Se estas características forem inaceitáveis, pode-se optar pela implementação utilizando versões múltiplas, gastando-se mais memória em troca. Das implementações do protocolo de bloqueio em duas fases, apenas a centralizada poderá ser adequada, pois em um cenário pessimista bloqueios mútuos serão muito frequentes e a implementação centralizada é a única que permite fácil deteção/resolução de bloqueios mútuos.

Para um cenário otimista, qualquer implementação é, em princípio, adequada, exceto a implementação conservativa do protocolo de pré-ordenação pois força as operações a serem processadas em ordem de senha, quer elas conflitem ou não. A implementação básica, dentre as do protocolo de pré-ordenação, é a mais indicada neste cenário. Quanto às implementações do protocolo de bloqueio em duas fases, a básica é a que se comporta melhor em termos de mensagens adicionais, sendo interessante neste cenário.

Isto conclui a nossa breve comparação entre as diversas implementações. O leitor interessado deverá consultar as referências deste capítulo para maiores detalhes.

NOTAS BIBLIOGRÁFICAS

Apresentaremos aqui apenas algumas referências selecionadas dentre uma vasta literatura. Este capítulo baseia-se na maior parte no tutorial sobre controle de concorrência de Bernstein e Goodman [1980,1981], que analisa um grande número de algoritmos para controle de concorrência. A teoria de serialização é abordada em detalhe em Papadimitriou, Bernstein e Rothnie [1977], Papadimitriou [1979], Bernstein, Shipman e Wong [1979] e Casanova [1980]. Gray [1978] descreve em bastante detalhe uma implementação do protocolo de bloqueios para o caso centralizado. Gray et all. [1976] analisa o uso de bloqueio com várias granularidades. Ries e Stonebraker [1977,1979] e Ries [1979] analisam o efeito do uso de objetos com granularidades diferentes sobre o desempenho do mecanismo de bloqueios. Korth [1982,1983] descreve variações do método de bloqueio. Thomas [1978,1979] analisa uma outra técnica para controlar bloqueio a cópias múltiplas. Eswaran et all. e Klug [1983] exploram a estratégia de bloqueios a nível lógico (bloqueios por predicados ou expressões da álgebra relacional). Fussel et all. [1981], Gligor e Shattuck [1980], Leung e Lay [1979] e Rypka e Lucido [1979] investigam o problema de bloqueio mútuo em BDDs. O algoritmo

hierárquico para deteção de bloqueios mútuos em BDDs foi primeiramente proposto por Menascé e Muntz [1979]. Obermack [1980] propôs uma outra forma interessante para deteção de bloqueios distribuídos, que não foi incluída no texto por ser não ser diretamente compatível com o modelo de processamento de transações usado. Métodos de pré-ordenação são discutidos em Bernstein e Goodman [1980], Rosenkrantz, Stearns e Lewis [1978] e Le Lann [1978]. Reed [1979], Stonebraker [1979] e Bernstein e Goodman [1983] analisam algoritmos baseados em versões múltiplas. O método de controle de concorrência usado no SDD-1 é bastante interessante, combinando uma pré-análise das transações com o algoritmo conservativo de pré-ordenação. Este método é descrito por Bernstein et all. [1978a,1978b,1980]. Bernstein e Shipman [1980] provam a correção deste método. Análises comparativas do desempenho de alguns algoritmos de controle de concorrência podem ser encontradas em Molina [1978], Menascé e Nakanishi [1979] e Nakanishi [1981].