UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE MESTRADO EM INFORMÁTICA

CONTROLE DE CONCORRÊNCIA
COM
TRANSAÇÕES MULTINÍVEIS

Cláudio de Souza Baptista

CAMPINA GRANDE
ABRIL – 1991

CLAUDIO DE SOUZA BAPTISTA

CONTROLE DE CONCORRENCIA COM TRANSAÇOES MULTINIVEIS

Dissertação apresentada ao Curso de Mestrado em Informática da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

AREA DE CONCENTRAÇAO: CIENCIA DA COMPUTAÇAO

STEPHANE PIERRE TURC

Orientador

CAMPINA GRANDE

ABRIL - 1991

B222c Baptista, Claudio de Souza
Controle de concorrencia com transacoes multiniveis / Claudio de Souza Baptista.- Campina Grande, 1991.
112 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Banco de Dados 2. Banco de Dados - Controle de Concorrencia 3. Dissertacao I. Turc, Stephane Pierre, Dr. II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 004.6(043)

CONTROLE DE CONCORRENCIA COM TRANSAÇOES MULTINIVEIS

CLAUDIO DE SOUZA BAPTISTA

DISSERTAÇAO APROVADA EM 04.04.1991

STEPHANE PIERRE TURC, Dr.
Orientador

MARCUS COSTA SAMPAIO, M.Sc
Componente da Banca

CIRANO IOCHPE, Dr.
Componente da Banca

Campina Grande, 04 de abril de 1991

AGRADECIMENTOS

Quero externar meus sinceros agradecimentos

a Deus, pela força espiritual,

aos meus Pais, José Cláudio e Valda, por não medirem esforços em estimular minha formação,

ao meu orientador, Dr. Stéphane Turc, pela maneira profissional, paciente, incentivadora e , sobretudo, amiga com a qual me acolheu durante a elaboração deste trabalho,

aos meus amigos e familiares, que sempre estiveram ao meu lado na horas tristes e alegres.

RESUMO

O controle de concorrência em Bancos de Dados tem recebido muita atenção nos últimos anos, no sentido de oferecer sistemas cada vez mais eficientes e confiáveis. As novas aplicações (CAD, CASE, Informação de Escritórios) têm requerido novas técnicas para o controle de concorrência.

Esta dissertação enfoca o controle de concorrência multinível, que tem sido difundido nos sistemas orientados a objetos, sistemas distribuidos e aplicações não-convencionais, em que fazemos um comparativo das vantagens desta abordagem com relação à abordagem convencional, mostrando o interesse real e prático da utilização desta nova abordagem.

Ademais, apresentamos um protocolo de bloqueios em duas fases multinível no qual propomos algumas hipóteses que se constituem na contribuição maior deste trabalho.

Sumário

1. INTRODUÇÃO

Um Banco de Dados é uma coleção finita de itens de dado os quais possuem valores que formam o estado do Banco de Dados. Para ter acesso a esses itens de dado, utilizamos transações, que são a interface entre os usuários e o sistema de Banco de Dados.

Intuitivamente, podemos dizer que uma transação é uma sequência de ações que interagem com um Banco de Dados, podendo, ou não, alterar o estado do mesmo.

Assumiremos que uma transação, quando executada sozinha e integralmente a partir de um estado inicial consistente do Banco de Dados, isto é, que está de acordo com as regras de integridade do sistema (regras que visam preservar a consistência do sistema de acordo com a semântica da aplicação), deve produzir um estado final também consistente. Ou seja, uma transação deve obedecer às restrições de integridade impostas pelo sistema. Assumiremos também que uma transação, quando executada, sempre termina, seja por intermédio de um aborto, que explicaremos mais tarde, ou através da execução da transação por completo.

No entanto, um Sistema Gerenciador de Banco de Dados, suportando a execução simultânea de várias transações, deve garantir um acesso concorrente aos dados para que se tenha uma melhor performance no atendimento às solicitações das várias aplicações. Para que isto ocorra, faz-se necessária a intercalação de transações que, por sua vez, pode deixar o Banco

de Dados num estado inconsistente, mesmo se cada transação, quando executada isoladamente, ou seja, sem intercalação, produza um estado consistente.

Desta forma, o Sistema Gerenciador de Banco de Dados necessita de mecanismos de controle de concorrência que assegurem uma execução concorrente das transações, evitando que estas produzam um estado inconsistente. Caso tais mecanismos não assegurem um estado consistente, poderemos ter as chamadas anomalias de sincronização.

Exemplificaremos dois tipos de anomalias : atualizações perdidas e acesso a dados inconsistentes [Bernstein 81].

Exemplo 1.1 Atualizações perdidas :

Suponhamos um Banco de Dados para aplicações bancárias onde temos as execuções das seguintes transações :

T_1 :	T_2 :
Read(temp, Conta1)	Read (temp, Conta1)
Add (temp, 100)	Add (temp, 200)
Write(temp, Conta1)	Write(temp, Conta1)

Isto é, ambas as transações, T_1 e T_2, fazem um depósito simultaneamente numa mesma conta bancária (Conta1). Sem o controle de concorrência, as transações podem ser intercaladas de qualquer maneira.

Suponhamos que a seguinte sequência seja executada :

S = $Read_1$(temp,Conta1); $Read_2$(temp,Conta1); Add_1(temp,100); Add_2(temp,200); $Write_1$(temp,Conta1); $Write_2$(temp,Conta1)

Suponhamos que o valor do saldo inicial da Conta1 seja 500. Portanto, seguindo a sequencia de execução acima temos : T_1 lê 500, T_2 lê 500, T_1 adiciona 100, logo a variável temp de T_1 fica com o valor 600; depois T_2 adiciona 200 a sua variável temp, que fica com o valor 700. Por último, T_1 grava na conta corrente Conta1 o valor de sua variável temp que é 600, em seguida T_2 grava no mesmo item de dado Conta1, o seu valor de temp que é 700. Logo, concluímos que a atualização de T_1 foi perdida, pois dois clientes depositaram um total de 300 e só foram creditados 200.

■

Exemplo 1.2 Leituras a dados inconsistentes :

Suponhamos o mesmo Banco de dados do Exemplo 1.1 e as seguintes transações, que são submetidas à execução simultaneamente :

T_1 :	T_2 :
Read(temp1, Conta1)	Read(temp1, Conta1)
Sub(temp1, 100)	Read(temp2, Conta2)
Write(temp1,Conta1)	Display(temp1 , temp2)
Read(temp2,Conta2)	
Add(temp2, 100)	
Write(temp2, Conta2)	

Suponhamos a seguinte sequência de execução :

S = $Read_1$(temp1,Conta1); Sub_1(temp1,100); $Write_1$(temp1,Conta1); $Read_2$(temp1,Conta1); $Read_2$(temp2,Conta2); $Display_2$(temp1,temp2); $Read_1$(temp2,Conta2); Add_1(temp2,100); $Write_1$(temp2,Conta2).

Em outras palavras, T_1 faz uma transferência de fundos da Conta1 para Conta2 e T_2 faz uma consulta aos saldos de Conta1 e Conta2.

Suponhamos que os saldos iniciais da Conta1 e Conta2 sejam 500 cada um. Ao executarmos a sequência S acima, temos que T_1 lê o valor 500 da Conta1, subtrai 100 deste valor, neste momento T_2 lê o valor da Conta1, que tem o valor 400 devido ao saque de T_1, e Conta2, que ainda tem o valor 500, e mostra-os ao usuário, em seguida, T_1 faz o depósito do valor 100 na Conta2, que passa a ter o valor 600.

Podemos notar que, ao final da execução, o estado do Banco de Dados é consistente, pois a transferência foi feita corretamente (Conta1 = 400 e Conta2 = 600). Entretanto, T_2 teve acesso a dados inconsistentes, pois para T_2 desapareceram 100. Isto ocorreu porque T_2 não viu todos os dados modificados por T_1. ∎

Para prover esta consistência citada anteriormente, precisamos adotar um critério de correção que indique se uma execução concorrente é correta. Este critério será a **serializabilidade** (do inglês serializability) que será estudada no capítulo 2.

Nos Bancos de Dados Convencionais, cujos modelos de dados são o modelo relacional, o de redes e o hierárquico [Ullman 83], vários métodos para controle de concorrência foram propostos nos últimos anos [Bernstein 81,87].

Entretanto, apesar de esses métodos garantirem execuções concorrentes serializáveis, eles não exploram ao máximo o paralelismo, pois são métodos que atuam sobre modelos de dados pobres semanticamente.

Ademais, os modelos de dados convencionais não atendem aos requisitos das novas aplicações, denominadas de aplicações não-convencionais, tais como : Informação de Escritórios, Projetos Assistidos por Computador (CAD), Concepção de Circuitos Integrados, etc., pois estes requerem interfaces mais amigáveis, transações de longa duração, abstração de dados, versões, dedução lógica, noção de tempo, restrições de integridade, entre outros.

Desta forma, requer-se novos modelos de dados, por exemplo o modelo orientado a objetos [Atkinson 89], em que novas técnicas, inclusive de controle de concorrência, são necessárias.

Esses novos modelos de dados incorporam tipos abstratos de dados. Com isso, essa abordagem consegue, através da exploração da semântica destes tipos, utilizando a comutatividade das operações e abstração de estado, um aumento significativo na performance do controle de concorrência [Schwarz 84a, 84b].

Além do ganho de paralelismo com operações tipadas, podemos aumentar ainda mais a concorrência se, ao invés de considerarmos as operações tipadas como indivisíveis, permitirmos que haja concorrência entre as suboperações que implementam uma operação tipada. Evidentemente, a coerência deve ser mantida e, portanto, deve haver sincronização entre as suboperações. Desta forma, nos dirigimos para o controle de concorrência com

transações multiníveis [Weikum 87, Beeri 88, Moss 82].

Esta abordagem multinível explora o paralelismo interno das transações, que são formadas com operações tipadas. Cada subtransação é uma unidade atômica, podendo em caso de falha ser recuperada e reexecutada sem comprometer a transação como um todo. Assim, estas transações multiníveis são indicadas para modelos de dados orientados a objetos, sistemas distribuídos e aplicações não-convencionais.

Esta dissertação tem como objetivo inicial apresentar um estudo sobre o controle de concorrência nos Bancos de Dados Convencionais, apresentando a teoria e os diversos mecanismos existentes. A partir daí, estudamos a abordagem multinível, enfatizando sempre as vantagens desta abordagem com relação aos métodos convencionais. Como objetivo final e contribuição deste trabalho, apresentamos um protocolo de bloqueio em duas fases para transações multiníveis. Neste protocolo, incluímos o estudo de algumas hipóteses que ainda não foram abordadas pelos pesquisadores da área [Weikum 87, Moss 85, Badrinath 90].

O interesse de propor novas hipóteses visa aprimorar o modelo de transações multiníveis, no sentido de prover um melhor atendimento aos novos requisitos das aplicações não-convencionais, preocupando-se com performance e confiabilidade. Para isso, estas hipóteses visam deixar o modelo de transações menos restritivo.

A organização do restante desta dissertação dá-se da seguinte forma : o capítulo 2 apresenta um estudo do estado da

arte do controle de concorrência nos Bancos de Dados Convencionais; o capítulo 3 aborda o controle de concorrência sobre operações tipadas, enfocando a comutatividade; o capítulo 4 enfoca a abordagem multinível; o capítulo 5 apresenta o protocolo de bloqueios em duas fases multinível; por último, o capítulo 6 faz as conclusões do trabalho e sugere as possíveis pesquisas que poderão dar continuidade a esta dissertação.

2. CONTROLE DE CONCORRENCIA PARA BANCOS DE DADOS CONVENCIONAIS

2.1 Introdução

Neste capítulo, abordaremos o controle de concorrência nos Bancos de Dados Convencionais. A consolidação dos sistemas de Bancos de Dados Convencionais, cujos modelos são o hierárquico o de redes e o relacional [Ullman 83] viabilizou a realização de diversos estudos no sentido de melhorar o controle de concorrência entre transações. Por exemplo, Bernstein apresenta vários métodos de controle de concorrência para esta abordagem convencional. [Bernstein 81]

O objetivo deste capítulo é, inicialmente, apresentar a teoria do controle de concorrência para a abordagem convencional, e aí se encontram a definição do modelo de transação, schedule (isto é, uma execução concorrente de várias transações) e a teoria da serializabilidade. Em seguida, expomos as principais técnicas utilizadas para prover um controle de concorrência nesta abordagem.

A organização deste capítulo está disposta da seguinte forma : a seção 2.2 enfoca o modelo de transação utilizado, a seção 2.3 apresenta a definição de um schedule, a seção 2.4 aborda a teoria da serializabilidade, a seção 2.5 apresenta o protocolo de bloqueios em duas fases [Eswaran 76], a seção 2.6 o mecanismo de timestamp [Bernstein 87, Korth 89, Ceri 84], a seção 2.7 apresenta o mecanismo otimista [Kung 81], a seção 2.8 apresenta o protocolo baseado no teste do grafo de serialização

[Bernstein 87, Sugihara 87] e, por último, a seção 2.9 faz uma breve conclusão da teoria apresentada no presente capítulo.

2.2 O Modelo de Transação

Enfocamos nesta seção o modelo de transação adotado. Consideremos um domínio D = { x, y, z, ... } composto do conjunto finito de itens de dado (objetos). Então, as operações sobre os objetos permitidos no modelo com x ∈ D e i ∈ N, são :

$R_i(x)$: é a operação de leitura do objeto x pela transação i;

$W_i(x)$: é a operação de gravação do objeto x pela transação i;

C_i : é a confirmação (Commit) da transação i, ou seja, quando a transação i é executada totalmente;

A_i : é o aborto (Abort) da transação i, ou seja, quando a execução da transação i é interrompida antes de todas as operações terem sido processadas pelo sistema.

Antes de darmos a definição do nosso modelo de transação, apresentamos o conceito de **conflito** que será de grande utilidade. Duas operações estão em conflito se elas operam sobre um mesmo item de dado, digamos x, e pelo menos uma delas é uma operação de gravação, isto é W(x).

O modelo de transação é baseado em gravações imediatas, isto é, uma operação de write atua diretamente no Banco de Dados, ao invés de gravações deferidas onde uma operação de write é

armazenada em uma área privada (buffer) e só depois é transferida para o Banco de Dados. Para um maior aprofundamento neste assunto, Turc faz uma comparação entre modelo com gravações imediatas e modelo com gravações deferidas. [Turc 90]

Podemos então definir uma transação T_i, $i \in N$ como um par $(OP_i, <_i)$ [Hadzilacos 88], onde :

1. $OP_i \subset \{ R_i(x), W_i(x), C_i, A_i \}$ com $x \in D$

2. $<_i$ é uma ordem parcial sobre OP_i

3. Quaisquer duas operações de OP_i em conflito são ordenadas por $<_i$, isto é, se a, b $\in OP_i$ estão em conflito, então ou a $<_i$ b ou b $<_i$ a.

4. $A_i \in OP_i$ se, e somente se $C_i \notin OP_i$, isto é, uma transação ou aborta ou confirma.

5. Se a $\in \{ A_i, C_i \}$ então $\forall$ b $\in OP_i - \{a\}$ => b $<_i$ a; isto é, o aborto A_i ou confirmação C_i são sempre a última operação de uma transação. Em outras palavras, uma transação termina ou com um Commit ou com um Abort.

Uma transação deve obedecer às seguintes propriedades [Haerder 83] :

P1. **ATOMICIDADE** => Uma transação é atômica se executa tudo ou nada, ou seja, se houver um Commit, então todos os writes devem ser processados completamente e seus resultados podem ser retornados ao usuário; por outro lado, se for abortada, então todos os writes feitos até o aborto devem ser desfeitos como se a

transação não tivesse sido executada.

P2. CONSISTENCIA => Uma transação deve sempre obedecer às regras de integridade para, quando executada sozinha e por completo, nunca deixar o Banco de Dados num estado inconsistente.

P3. ISOLAÇÃO => Eventos ocorridos dentro de uma transação devem ser escondidos das demais transações executadas concorrentemente, até que a primeira confirme.

P4. DURABILIDADE => Também chamada de persistência. Quando uma transação confirma, suas atualizações não podem ser desfeitas a não ser por outra transação confirmada, e uma vez abortada seus efeitos não podem reaparecer.

2.3 Schedules

Também conhecido na literatura de Banco de Dados como Histórias [Bernstein 87] ou Logs [Hadzilacos 88], um Schedule S é um conjunto parcialmente ordenado de operações pertencentes a um conjunto de transações.

Exemplo 2.1 : A execução concorrente das transações abaixo constitui um schedule.

$S = R_1(x); R_2(y); W_1(x); C_1; W_2(y); C_2$

■

Então podemos definir um Schedule Completo S, sobre as transações T_i, com $1 < i < N$ e N = número de transações de S,

como sendo uma ordem parcial (OP_S, $<_S$), onde :

1. $OP_S = \bigcup_{i=1}^{N} OP_i$; isto é, OP_S é o conjunto de todas as operações OP_T das transações.

2. $<_S \supset <_i$, $1 \leq i \leq N$, isto é, existe uma compatibilidade entre a ordem do Schedule e a ordem das transações. Ou seja, a ordem do Schedule, $<_S$, não vai mudar a ordem estática entre operações de uma transações, $<_i$.

3. Todas as operações em conflito em OP_S são ordenadas por $<_S$.

Um Schedule (incompleto) é um prefixo de um Schedule completo, representando uma possível execução incompleta de transações [Hadzilacos 88].

A necessidade do Schedule incompleto dá-se porque um Schedule Completo contém todas as transações completadas e muitas vezes precisamos de um Schedule que analise passos intermediários da execução das transações, como se num determinado momento nós parássemos a execução para analisar a ordem das operações. Em outras palavras, o Schedule está relacionado com transações **ativas**, que são aquelas que ainda nem confirmaram nem abortaram.

Formalmente, dizemos que uma transação é ativa se e somente se $C_i \notin OP_S$ e $A_i \notin OP_S$. Utilizamos grafos direcionados para representar graficamente o comportamento (ordem) de um Schedule, onde os arcos indicam a ordem definida por $<_S$. Por

exemplo, $R_1(x)$ -> $W_3(x)$, lê-se $R_1(x)$ precede $W_3(x)$ ou $W_3(x)$ segue $R_1(x)$.

Exemplo 2.2 : Sejam as transações :

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(y) \rightarrow W_2(x) \rightarrow W_2(y) \rightarrow C_2$

Então um Schedule Completo S seria :

$$S = \begin{array}{l} R_1(x) \rightarrow W_1(x) \rightarrow C_1 \\ \qquad\qquad\quad \downarrow \\ R_2(y) \rightarrow W_2(x) \rightarrow W_2(y) \rightarrow C_2 \end{array}$$

Um Schedule Incompleto, ou prefixo de S, S´ seria:

$$S' = \begin{array}{l} R_1(x) \rightarrow W_1(x) \rightarrow C_1 \\ \qquad\qquad\quad \downarrow \\ R_2(y) \rightarrow W_2(x) \end{array}$$

∎

Existem também Schedules com ordem total das operações tais como :

$R_1(x) \rightarrow R_2(x) \rightarrow W_1(x) \rightarrow C_1 \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$,

podendo-se suprimir os arcos para tais Schedules :

$R_1(x)\ R_2(x)\ W_1(x)\ C_1\ W_2(y)\ W_2(x)\ C_2$

Denotamos também **Com(S)** como sendo um Schedule Completo sobre o conjunto de transações confirmadas num dado instante.

2.4 Serializabilidade

A teoria da serializabilidade é utilizada para definir

um critério de correção de Schedules que intercalam várias transações [Bernstein 87]. Antes de definirmos o que é um Schedule Serializável, precisamos de duas outras definições: **Schedule serial e equivalência de Schedules.**

Um Schedule S é uma execução serial, ou simplesmente serial, se as transações são executadas uma após outra. Formalmente, sejam : tr(S) o conjunto de transações que estão em S, a e b´ $\in$ OP_i - { A_i, C_i } então
podemos dizer que um Schedule Completo S é serial se :

$\forall$ T e T´ $\in$ tr(S) $\forall$ a $\in$ OP_T, $\forall$ b´$\in$ OP_T· a $<_S$ b´
ou $\forall$ a $\in$ OP_T $\forall$ b´ $\in$ OP_T· b´ $<_S$ a.

Desde que assumimos em nosso modelo de transação que a propriedade de consistência é válida para uma transação quando esta é executada isoladamente, então um Schedule serial é consistente.

Exemplo 2.3 Sejam as transações :

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

Então existem duas possíveis execuções seriais (permutação de N, onde N é o número de transações) :

$S_1 = R_1(x)\ W_1(x)\ C_1\ R_2(x)\ W_2(y)\ W_2(x)\ C_2$

ou $S_2 = R_2(x)\ W_2(y)\ W_2(x)\ C_2\ R_1(x)\ W_1(x)\ C_1$

isto é :

$S_1 = T_1 ; T_2$

$S_2 = T_2 ; T_1$

onde $T_i;T_j$ significa que T_i é executado inteiramente antes de T_j. ∎

Definiremos a seguir a equivalência de conflito entre Schedules.

Dizemos que um Schedule S_1 é **computacionalmente equivalente** a um Schedule S_2, denotado por $S_1 \equiv S_2$, se

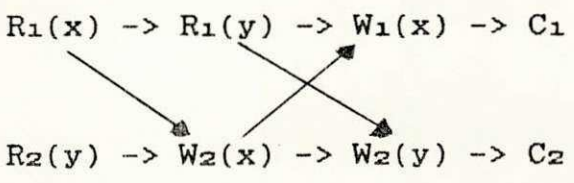
1. Iniciam com o mesmo estado do Banco de Dados e resultam no mesmo estado final;

2. $tr(S_1) = tr(S_2) \Rightarrow OP_{a1} = OP_{a2}$, isto é, eles têm o mesmo conjunto de transações e, consequentemente, têm o mesmo conjunto de operações;

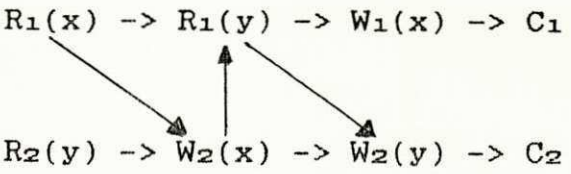
3. $\forall a_1 \in T_1$ e $b_2 \in T_2$, onde ou a_1 ou b_1 é $w(x)$, isto é, a_1 conflita com b_2, e $A_1, A_2 \notin S$, então $a_1 <_{a1} b_2$ se e somente se $a_1 <_{s2} b_2$ ou $b_2 <_{s1} a_1$ se e somente se $b_2 <_{s2} a_1$, isto é, a ordem de operações em conflito de transações não abortadas é a mesma em S_1 e S_2.

Exemplo 2.4: Sejam S_1,S_2 e S_3 os seguintes Schedules :

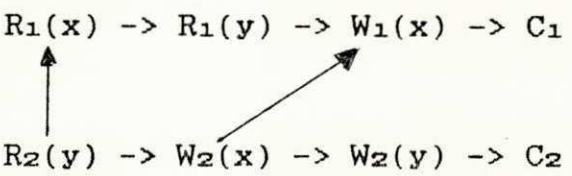
S_1 =



S_2 =



S_3 =



Então, $S_1 \equiv S_2$ mas S_3 não é equivalente a nenhum.

■

Agora podemos definir um Schedule Serializável. Um Schedule é **serializável** se é computacionalmente equivalente a um Schedule Serial, isto é, produz o mesmo resultado e o mesmo efeito no Banco de Dados, que uma execução serial do conjunto de transações tr(S) [Bernstein 87].

Exemplo 2.7 Sejam as transações :

$T_1 = R_1(x)\ W_1(x)\ C_1$

$T_2 = R_2(y)\ W_2(x)\ C_2$

Portanto os dois Schedules seriais possíveis são :

$S_1 = T_1\ ;\ T_2$ e

$S_2 = T_2\ ;\ T_1$

Os Schedules serializáveis são :

S_1 e S_2, pois todo Schedule serial é serializável;

$S_3 = R_1(x)\ R_2(y)\ W_1(x)\ W_2(x)\ C_1\ C_2$

$S_4 = R_2(y)\ R_1(x)\ W_1(x)\ W_2(x)\ C_1\ C_2$

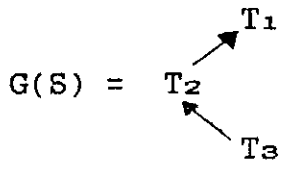
■

Uma outra maneira de verificarmos se um Schedule é serializável é através do grafo obtido através do Schedule, o qual chamamos de **Grafo de Serialização para o Schedule**, denotado por **G(S)**. Este grafo é dirigido e é definido como G(S) = (Com(S), ->), onde Com(S) são os nodos do grafo representando o conjunto de transações que confirmaram em S, e -> são os arcos, onde se T_i -> T_j, então todas as operações de T_i, que conflitam com operações de T_j, ocorreram antes das respectivas operações de T_j.

Exemplo 2.8: Seja o seguinte Schedule S :

$$S = \begin{array}{ll} R_1(x) \rightarrow W_1(x) \rightarrow C_1 & (T_1) \\ R_2(x) \rightarrow W_2(x) \rightarrow W_2(y) \rightarrow C_2 & (T_2) \\ R_3(y) \rightarrow W_3(y) \rightarrow C_3 & (T_3) \end{array}$$

Então o grafo de serialização para S seria :



■

Uma vez definido o grafo de serialização, podemos dar uma nova definição do que seja um Schedule Serializável. Dizemos que um Schedule S é serializável se e somente se o grafo de serialização para S, G(S), é **acíclico** [Bernstein 87, Hadzilacos 88, Korth 89, Casanova 85, Ceri 84].

2.5 Protocolo de Bloqueio em Duas Fases

O Protocolo de Bloqueio em Duas Fases ou Protocolo 2PL (do inglês Two Phase Locking) foi proposto inicialmente por Eswaran e utiliza a técnica de bloqueios para sincronizar o acesso a dados compartilhados, fazendo esta sincronização de acordo com a teoria da serializabilidade. [Eswaran 76]

Antes de descrevermos este protocolo falaremos um pouco da técnica de bloqueios que é o fundamento deste protocolo. Um bloqueio é solicitado por uma transação, quando esta quer ter acesso a um determinado item de dado. Uma vez atendido o pedido de bloqueio, isto significará que nenhuma outra transação, com operação conflitante com o objeto bloqueado, poderá ter acesso a este objeto bloqueado, até que a transação que detém o bloqueio libere-o. Existem dois tipos de bloqueios : o bloqueio **compartilhado**, também chamado readlock, e o bloqueio **exclusivo**, também conhecido como writelock.

Os bloqueios compartilhados são aqueles sobre operações não conflitantes. Por exemplo, se uma transação faz um readlock num objeto, outras transações podem fazer readlocks no objeto, pois não há conflito, uma vez que as transações estão apenas consultando os objetos, sem modificá-los. Este bloqueio propicia uma maior concorrência ao sistema.

Por outro lado, os bloqueios exclusivos são solicitados quando uma transação quer modificar um objeto do Banco de Dados, ou seja, são os writelocks. Daí, nenhuma outra transação pode ter acesso ao objeto enquanto a transação que possui o bloqueio não

liberar este objeto.

O Protocolo 2PL, como o próprio nome já diz, consiste de duas fases : a primeira fase chamada **fase de crescimento** e a segunda fase denominada **fase de liberação**. Durante a fase de crescimento a transação obtém os bloqueios nos objetos que vão ser por ela utilizados. Na fase de liberação a transação tende a liberar todos os bloqueios sobre os objetos à medida que não vai mais utilizá-los. Podemos deduzir que, uma vez liberado um único bloqueio de um objeto, a transação não pode mais tentar bloquear quaisquer objetos [Casanova 85, Korth 89].

Para provar a correção do Protocolo 2PL é suficiente provarmos que um Schedule produzido por este protocolo é serializável. Ou seja, pelo teorema da Serializabilidade [Bernstein 87], precisamos provar que o grafo G(S), onde S é o Schedule, é acíclico.

Seja S um Schedule produzido pelo 2PL Scheduler. Sejam $L_i(x)$ a operação de solicitação de bloqueio do objeto x pela transação i ; $Op_i(x)$ uma operação de read ou write sobre o objeto x pela transação i; e $U_i(x)$ a operação de liberação do bloqueio do objeto x pela transação i. Então temos as seguintes propriedades:

P1. Se $Op_i(x) \in Com(S)$, então $L_i(x)$ e $U_i(x) \in Com(S)$ e $L_i(x) < Op_i(x) < U_i(x)$; isto é, uma transação antes de ter acesso a um objeto deve bloqueá-lo e, após usá-lo, deve liberá-lo.

P2. Se $Op_i(x)$ e $Op_j(x) \in Com(S)$, com $i \neq j$, são

operações conflitantes então ou $U_i(x) < L_j(x)$ ou $U_j(x) < L_i(x)$, isto é, existe uma ordem total entre as operações em conflito garantindo um acesso exclusivo ao objeto.

P3. Seja S um Schedule completo produzido por um 2PL Scheduler. Se quaisquer $Op_i(x)$ e $Op_i(y) \in Com(S)$, então $L_i(x) < U_i(y)$, isto é, uma transação deve solicitar todos os bloqueios antes de começar a liberar bloqueios.

Para provarmos a serializabilidade de um Schedule produzido por um 2PL Scheduler, podemos supor que o grafo G(S) tem um ciclo, T1 -> T2 -> ... -> Tn -> T1. Pela propriedade **P3**, T1 deve solicitar todos os bloqueios antes de começar a liberá-los, porém se houver tal ciclo cria-se uma contradição pois : $L_i(x) < U_i(x) < L_i(y)$, portanto este ciclo não é permitido pelo protocolo. Portanto, como G(S) não contém ciclos, pela segunda definição de serializabilidade temos que S é serializável [Casanova 85].

O ponto de transição da fase de crescimento para fase de liberação é chamado **ponto de bloqueio**. Ele determina a ordem de serialização do Schedule. Isto ocorre porque quando uma transação T_i atinge seu ponto de bloqueio, todas as outras transações que necessitem dos objetos bloqueados por T_i devem esperar até que sejam liberados estes objetos, fazendo com que a transação T_i que conseguiu atingir seu ponto de bloqueio primeiro seja precedente na ordem de execução. No grafo de serialização, se existir um arco de T_i para T_j quer dizer que T_i atingiu seu ponto de bloqueio antes de T_j.

Resumindo, poderíamos dizer que se duas transações conflitam, aquela que atingir o ponto de bloqueio primeiro terá precedência de execução em relação a outra.

Dessa forma, o Schedule ordena as transações pelos seus respectivos pontos de bloqueio, determinando uma ordem total sobre as operações em conflito.

Com o intuito de prover uma maior concorrência pode-se utilizar as chamadas **conversões de bloqueio**, isto é, uma transação obtém um bloqueio compartilhado sobre um objeto e depois eleva esse bloqueio para o modo exclusivo, ou vice-versa, uma transação possui um bloqueio sobre um objeto no modo exclusivo e depois abaixa este bloqueio para o modo compartilhado. É importante notar que estas conversões não são arbitrárias, ou seja, a elevação deve acontecer na fase de crescimento , enquanto que o abaixamento deve acontecer apenas na fase de liberação.

Este maior grau de paralelismo é propiciado porque uma transação, ao invés de bloquear um objeto no modo exclusivo logo no início de sua execução, faz um bloqueio compartilhado sobre o objeto, permitindo que outras transações tenham acesso concorrente ao objeto bloqueado e, apenas quando a transação quiser modificar o objeto bloqueado, ela solicita uma conversão de bloqueio para o modo exclusivo.

O grande problema deste protocolo de bloqueio em duas fases é que ele é susceptível a bloqueios mútuos, conhecidos como **deadlocks**, que simplesmente paralisam a execução das transações

envolvidas no deadlock, pois cria-se um impasse devido as transações que ficam esperando por liberações que nunca ocorrerão. Além disso, um deadlock impõe um bloqueio infinito dos objetos que estão bloqueados pelas transações em deadlock, de forma que nenhuma outra transação pode ter acesso a estes objetos. Na próxima seção faremos um estudo deste problema.

2.5.1 Deadlocks

Um sistema está em deadlock quando se tem, pelo menos, uma sequência de transações, T_1, T_2,...,T_n, onde T_i espera por uma liberação de $T_{(i+1 \bmod n)}$, $i = 1,...,n$; de forma que nenhuma das T_i terminará sem que haja uma interferência do scheduler para resolver o impasse.

Exemplo 2.7 Sejam os seguintes trechos de transações, considerando bloqueios exclusivos :

$T_1 = L_1(x)\ L_1(y)$

$T_2 = L_2(y)\ L_2(x)$

e seja o seguinte Schedule :

$S = L_1(x)\ L_2(y)\ L_1(y)\ L_2(x)$,

então o Schedule S provocou um deadlock, pois a transação T_1 bloqueia o objeto x e fica esperando pelo bloqueio do objeto y que está bloqueado pela transação T_2, que, por sua vez, está esperando pelo bloqueio do objeto x, que está com a transação T_1, daí criando-se o impasse. Devemos notar que no Schedule acima as

operações de pedido de bloqueio $L_1(y)$ e $L_2(y)$ não chegam a ser executadas e ficam esperando.

■

Existem duas técnicas para lidar com deadlocks : uma que previne os deadlocks, isto é, não os deixa ocorrer, e a outra técnica que detecta quando um deadlock ocorre e soluciona o impasse. A técnica de prevenção é mais utilizada quando a incidência de deadlocks é muito grande, enquanto que a técnica de detecção/resolução é adequada quando esta incidência é pequena.

Ambas as técnicas imprimem um "overhead" de execução ao mecanismo de controle de concorrência, o qual deve ser o mínimo possível.

2.5.1.1 Prevenção de Deadlocks

Existem várias técnicas para se prevenir deadlocks, citamos algumas delas :

. Bloquear todos os objetos a serem usados, antes de iniciar a execução, através de uma pré-compilação. Se não se conseguir todos os bloqueios de que se necessita, então a transação deve ser repetida. Esta é a maneira mais simples de implementar a prevenção de deadlocks porém, existem dois problemas: o primeiro é o baixo paralelismo que esta técnica pode prover, pois mesmo que uma transação utilize um objeto por pouco tempo, este é bloqueado durante toda a execução da transação (2PL Estrito). O outro problema é mais sério, pois

pode ser que uma transação fique esperando indefinidamente por um objeto que está sempre alocado.

. Um outro método é impor uma ordenação total de todos os objetos e exigir que as transações bloqueiem objetos somente nesta ordem.

. Outros métodos utilizam preempção e repetição de transações. Para isso, deve-se atribuir um timestamp (senha) único a cada transação [Lamport 78]. Se uma transação é repetida, ela permanece com seu timestamp anterior. Existem dois métodos que utilizam esta técnica [Bernstein 87]:

1. **wait - die** (espera - morre) => E uma técnica não-preemptiva, onde uma transação T_i espera por outra T_j que retém o objeto desejado se T_i tiver um timestamp menor que o timestamp de T_j , caso contrário T_i é abortada e depois repetida com o mesmo timestamp.

Exemplo 2.8 Sejam T_1 e T_2 transações com timestamp $TS(T_1) = 5$ e $TS(T_2) = 10$. Se T_1 quiser um objeto bloqueado por T_2, então como $TS(T_1) < TS(T_2)$, T_1 espera. Por outro lado, se T_2 quiser um objeto bloqueado por T_1 então, como $TS(T_2) > TS(T_1)$, T_2 é repetida com o mesmo timestamp.

∎

2. **wound - wait** (fere - espera) => E uma técnica preemptiva onde se uma transação T_i quiser um objeto bloqueado por outra transação T_j e o timestamp (TS) de T_i for maior que o timestamp de T_j então T_i espera pela liberação do objeto, caso contrário, T_j é repetida, ou seja T_j é "ferida" por T_i.

Exemplo 2.9 Seja T_1 com $TS(T_1) = 5$ e T_2 com $TS(T_2) = 10$. Se T_1 quiser um objeto bloqueado por T_2, como $TS(T_1) < TS(T_2)$, então o objeto terá preempção de T_2 e T_2 será repetida. Por outro lado, se T_2 quiser um objeto bloqueado por T_1, como $TS(T_2) > TS(T_1)$ então T_2 esperará pela liberação do objeto bloqueado por T_1.
■

Um problema com esses métodos que utilizam o conceito de preempção/repetição é que podem ocorrer algumas repetições desnecessárias.

2.5.1.2 Detecção e Resolução de Deadlocks

Se o sistema permitir a ocorrência de deadlocks, deve prover um mecanismo para detectar e resolver o impasse. Para isso, um algoritmo é executado periodicamente para determinar se ocorreu deadlock; caso positivo, deve resolvê-lo. Uma maneira bastante rústica de implantar este mecanismo é o **timeout**, isto é, o scheduler determina um tempo limite que uma transação pode esperar por um bloqueio. Se este limite for esgotado, então o scheduler supõe que há um deadlock e, portanto, aborta a transação.

Este método pode incorrer num erro, pois ele pode supor que uma transação está em deadlock e portanto, abortá-la, quando na verdade ela está esperando por uma liberação do bloqueio do objeto de outra transação. O problema desta técnica é especificar qual o tempo ideal para o timeout.

Outra técnica para detectar o deadlock é quando o scheduler contrói um grafo dirigido chamado grafo **wait-for**, que consiste num par G = (N,A) onde N é o conjunto de nodos, que são todas as transações, e A é o conjunto de arcos, onde cada arco é um par ordenado (T_i,T_j), que indica que a transação T_i está esperando que a transação T_j libere um objeto de que T_i precisa. O deadlock é detectado quando o grafo contém um ciclo.

Exemplo 2.10 Sejam os seguintes trechos de transações, considerando bloqueios exclusivos :

T_1 = $L_1(x)$; $L_1(y)$; ...

T_2 = $L_2(y)$; $L_2(x)$; ...

Supondo o seguinte Schedule de execução :

S = $L_1(x)$;$L_2(y)$;$L_2(x)$; $L_1(y)$;

então o grafo wait-for seria :

G = T_1 -> T_2 -> T_1,

havendo um ciclo e consequentemente um deadlock.

■

O problema desta técnica é saber com que frequência deve-se aplicar o algoritmo de detecção do ciclo no grafo. Ademais, nos sitemas distribuídos, a construção deste grafo é muito dispendiosa.

Uma vez detectado o deadlock, o scheduler deve resolvê-lo. Para isso, deve-se escolher uma vítima que quebre o impasse, repetindo esta transação. E importante salientar que, por

questões de performance, é melhor repetir transações que incorrerão no menor custo. Após decidir qual é a vítima, deve-se determinar se a repetição deve ser total (reinício da execução da transação) ou se a repetição deve ser apenas o suficiente para quebrar o impasse.

Além disso, deve-se prever que a escolha da vítima não recaia sempre sobre uma mesma transação, pois incorreria no risco desta transação nunca ser executada, o que seria inadmissível. Para isso, pode-se colocar no cálculo do custo o número de repetições já efetuadas por uma transação [Bernstein 87].

2.6 Ordenação por TIMESTAMP

Uma outra técnica existente que também é utilizada no controle de concorrência é a **Ordenação por Timestamp** que consiste de um ordenamento prévio, **total e estrito**, das transações de acordo com o valor do timestamp. Isto é feito pelo Sistema Gerenciador de Banco de Dados que atribui um timestamp (senha) único e fixo no sistema para cada transação, no instante em que esta é criada.

Este número pode ser atribuido de diversas maneiras, entre as quais pode-se utilizar o próprio relógio do sistema para marcar as transações ou um contador lógico gerenciado pelo Sistema Gerenciador de Banco de Dados que é incrementado a cada transação criada [Korth 89, Lamport 78].

Esta técnica de ordenação por timestamp define uma ordem total entre as transações de forma que, se duas transações conflitam, então este conflito é resolvido pelo timestamp de cada transação.

Denotaremos o timestamp atribuído a uma transação T_i qualquer por $TS(T_i)$, que, como mencionamos anteriormente, consiste de um número único e fixo no sistema.

Portanto, se tivermos duas transações distintas, isto é, $T_i \neq T_j$, então ou $TS(T_i) < TS(T_j)$ ou $TS(T_i) > TS(T_j)$, isto é, existe uma ordem estipulada previamente pelo sistema nas execuções das transações.

Basicamente, o protocolo de ordenação por timestamp funciona da seguinte maneira : para cada objeto O do Banco de Dados, o sistema mantém duas variáveis,

W_timestamp(O) -> Variável que armazena o valor do timestamp da última transação que modificou o objeto O.

R_timestamp(O) -> Variável que armazena o valor do timestamp da última transação que leu o objeto O.

Então se uma transação T_i é escalonada e deseja ler o valor de um determinado objeto O, o sistema procede da seguinte forma :

1. Se o timestamp da transação T_i, que está querendo ler o objeto O, for menor que o valor da variável W_timestamp(O), em outras palavras, $TS(T_i) <$ W_timestamp(O), então a operação de leitura deve ser rejeitada pois esta transação T_i "chegou

atrasada". Desta forma, a transação T_i deve ser repetida.

2. Se o timestamp da transação T_i, que está querendo ler o objeto O, for maior ou igual ao valor da variável W_timestamp(O), isto é, $TS(T_i) \geq W\_timestamp(O)$, então a operação de leitura do objeto O é executada e a variável R_timestamp é atualizada para o máximo entre os valores R_timestamp(O) e $TS(T_i)$. Pensando em termos de algoritmo teríamos:

```
se TS(Ti) < W_timestamp(O)
      então Repita Ti
      senão
         início
           READ(O);
           R_timestamp(O) := MAX(R_timestamp(O),TS(Ti))
         fim
fimse
```

onde MAX(x,y) é uma função que retorna o maior valor entre x e y.

Devemos observar que como uma operação de leitura conflita apenas com uma operação de gravação, conflito read-write, fazemos a verificação apenas da variável W_timestamp(O), pois a variável R_timestamp(O) não precisa ser verificada.

Por outro lado, suponhamos agora que uma transação T_i deseje efetuar uma gravação num objeto O, isto é write(O), daí temos o seguinte procedimento :

1. Se o timestamp da transação T_i, $TS(T_i)$, for menor do que o valor da variável R_timestamp(O), isto é, $TS(T_i) <$ R_timestamp(O), então isto quer dizer que a atualização "chegou atrasada". Isto implicará na rejeição da operação de gravação write(O) e consequentemente na repetição da transação T_i.

2. Se o timestamp da transação T_i, $TS(T_i)$, for menor do que o valor da variável W_timestamp(O), isto é, $TS(T_i) <$ W_timestamp(O), então a operação de gravação, write(O), deve ser rejeitada pois está obsoleta. Entretanto, esta operação de gravação pode ser **ignorada**, segundo a **Regra de gravação de Thomas** [Thomas 79] que diz que quando tivermos $TS(T_i) <$ W_timestamp(O) e a operação é um Write(O), então ao invés de rejeitarmos a operação de gravação devemos simplesmente ignorá-la, isto dá uma otimização ao scheduler.

3. Caso contrário, isto é, $TS(T_i) \geq$ R_timestamp(O) e $TS(T_i) \geq$ W_timestamp(O) então a operação de gravação, write(O), deve ser executada e a variável W_timestamp(O) recebe o valor de $TS(T_i)$. Pensando em termos de algoritmo teríamos :

```
se TS(Ti) < R_timestamp(O)
      então REPITA Ti
      senão
          se TS(Ti) ≥ W_timestamp(O)
                então início
                        W_timestamp(O) := TS(Ti);
                        WRITE(O);
                      fim
                Senão { Regra de Thomas }
          fimse
fimse
```

O mesmo algorítmo de gravação sem considerar a regra de Thomas seria:

```
se TS(Ti) < R_timestamp(O) ou TS(Ti) < W_timestamp(O)
      então REPITA Ti
      senão
            início
                WRITE(O);
                W_timestamp(O) := TS(Ti)
             fim
fimse
```

Observemos que como a operação é de gravação, então

deve-se considerar as duas variáveis R_timestamp(O) e W_timestamp(O), posto que uma operação de gravação conflita tanto com uma operação de leitura, conflito read-write, quanto de gravação, conflito write-write.

Salientamos também que quando uma transação é repetida, recebe um novo timestamp quando do seu reinício, o que a torna improvável de ser repetida indefinidamente.

Exemplo 2.11 : Suponhamos as seguintes transações :

$T_1 = R_1(x); R_1(y); W_1(x); C_1$

$T_2 = R_2(y); R_2(x); W_2(y); W_2(x); C_2$

então dado que $TS(T_1) < TS(T_2)$, podemos ter o seguinte escalonamento S :

$S = R_1(x); R_1(y); R_2(y); R_2(x); W_2(y); W_2(x); \boldsymbol{W_1(x)}; C_1; C_2$

onde a última operação $\boldsymbol{W_1(x)}$ é ignorada. Porém o seguinte escalonamento S´ não é possível :

$S' = R_1(x); R_1(y); R_2(y); R_2(x); W_1(x); W_2(y); W_2(x); C_1; C_2,$

pois neste caso a operação $W_1(x)$ seria rejeitada e T_1 seria repetida visto que $TS(T_1) < R\_timestamp(x) = TS(T_2)$.

■

Um problema proveniente do método de ordenação por Timestamp descrito acima é que ele é susceptível a **abortos em cascata** conforme exemplo a seguir.

Exemplo 2.12 : Seja o seguinte schedule S, com $TS(T_2) > TS(T_1)$:

$$S = R_1(x); W_1(x); R_2(x); C_2$$

Como vemos, a transação T_2 lê um valor escrito por T_1 sendo que a transação T_2 é confirmada, porém a transação T_1 não o é. Isto pode causar uma situação irrecuperável caso a transação T_1 venha a abortar.

■

A solução para este problema é a **Ordenação por Timestamp Estrita**, que consiste em permitir leituras de objetos gravados por outras transações apenas se estas últimas forem confirmadas. Desse modo, se uma transação quer ler um valor que foi escrito por outra transação, a primeira deve esperar que a segunda termine através de uma confirmação ou de um aborto.

Exemplo 2.13 : Seguindo o exemplo 2.12 deveríamos ter o seguinte Schedule :

$$S = R_1(x); W_1(x); C_1; R_2(x); C_2$$

■

Vale salientar também que, se a transação T_2 do exemplo 2.12 ao invés de executar uma operação de $R_2(x)$, executasse um $W_2(x)$, tornar-se-ia uma situação de complexa recuperação, pois, caso a transação T_1 abortasse, dever-se-ia ignorar a recuperação para T_1 uma vez que o objeto x já teria sido, depois, atualizado por T_2.

Ademais, caso as duas transações abortem, então é

difícil para o mecanismo de recuperação saber qual o valor que o objeto x tinha antes de ser modificado pela primeira transação. Esta situação não causa aborto em cascata, porém requer um mecanismo de recuperação bastante complexo. Por isso, alguns autores requerem ordenação por timestemp estrita para precedências write-write. Isto quer dizer que, uma gravação de um objeto que foi gravado por outra transação só será permitido, se esta última for confirmada ou abortada.

2.7 Uma Técnica Otimista

Vimos que os protocolos mencionados anteriormente quais sejam, Bloqueio em Duas Fases e Ordenação por Timestamp, impõem um certo "overhead" para escalonar as transações, o que certamente leva a um atraso maior neste escalonamento. O Protocolo de Bloqueio em Duas Fases faz com que as transações fiquem esperando pela liberação dos objetos, enquanto que o protocolo de Ordenação por Timestamp faz a rejeição de transações em conflito que não podem ser escalonadas.

Esta seção aborda uma nova técnica conhecida por **Técnica Otimista** [Kung 81], que parte da hipótese de que o conjunto de transações é formado em sua maioria por **transações de consulta**, isto é, aquelas que apenas lêem os objetos do Banco de Dados, tornando a taxa de conflitos muito baixa.

Este novo método tenta diminuir o "overhead" causado pelos outros protocolos, através da execução de todas as

transações até o fim, sendo que as possíveis atualizações destas transações são feitas em variáveis locais à transação. A atualização do Banco de Dados se faz quando a transação passa pela fase de validação: isto significa dizer que, após a validação, as variáveis locais são copiadas para os respectivos objetos do Banco de Dados.

Neste método otimista, uma transação consiste de três fases [Kung 81]:

1. **Fase de Leitura** => é a fase em que a transação é executada por inteiro, fazendo as atualizações em cópias locais, conforme mencionamos anteriormente. Portanto, não há atualização no Banco de Dados durante esta fase.

2. **Fase de Validação** => é a fase em que se verifica se as cópias locais podem ser copiadas para o Banco de Dados sem que haja perda de consistência. Caso a transação seja validada então é executada a fase de gravação descrita a seguir, caso contrário, a transação é repetida.

3. **Fase de Gravação** => Uma vez validada uma transação durante a fase de validação, então as atualizações das cópias locais no Banco de Dados são realizadas.

Detalhamos, a seguir, a fase de validação uma vez que as outras duas fases são por si só auto-explicativas.

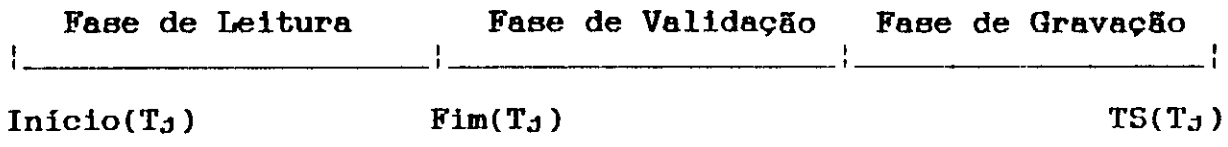
Cada transação recebe um timestamp único quando a validação ocorre, denotado por $TS(T_i)$. Utilizamos um contador

global, denotado por **ct** (contador de transação), para atribuirmos os timestamps, sendo que a cada atribuição de um valor, este contador é incrementado de uma unidade.

Durante a fase de leitura devemos guardar os conjuntos de itens lidos e gravados pela transação que estamos validando. Assim, cada transação possui um ConjRead, que é um conjunto que indica todos os objetos que foram lidos, e um ConjWrite, que indica os objetos gravados pela transação.

Além disso, devemos guardar o valor do **ct** quando a transação inicia sua fase de leitura, denotado por **Inicio(T_j)**, como também o valor do **ct** quando a transação termina sua fase de leitura, denotado por **Fim(T_j)**.

A figura abaixo mostra-nos como seria a estrutura de uma transação.

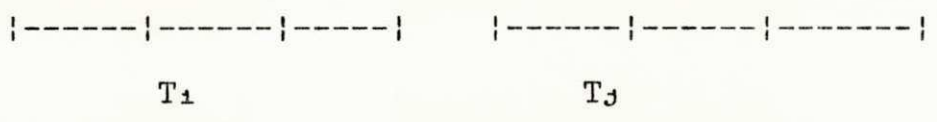


A correção deste método baseia-se no fato de que o Schedule produzido durante a fase de validação deve ser equivalente a um Schedule serial ordenado pelos timestamps $TS(T_i)$. A validação de uma transação T_j é feita com relação a todas as outras transações T_i já validadas, isto é, $TS(T_i) < TS(T_j)$. Assim, o método proibe a formação de precedências no sentido $T_j \rightarrow T_i$, só admitindo-as no sentido inverso. Isto garante a ausência de ciclos entre transações confirmadas.

Para isso, uma das seguintes condições [Kung 81] deve ser obedecida para validação de uma transação T_j com relação a toda transação T_i tal que, **$TS(T_i) < TS(T_j)$** :

C1. T_i completa sua fase de gravação antes de T_j começar sua fase de leitura, ou seja, as transações são executadas serialmente. Em outras palavras, $TS(T_i) <$ Início(T_j). Na figura a seguir, expomos a ocorrência da condição C1.

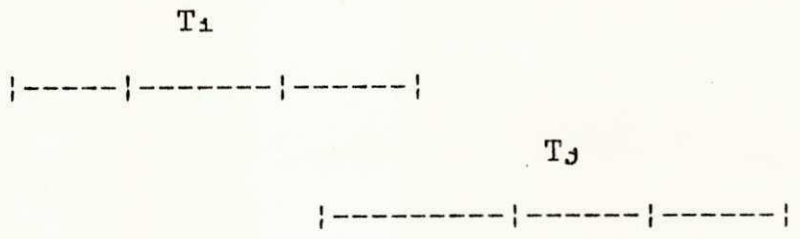
Condição C1 : $TS(T_i) <$ Início(T_j)



T_i T_j

C2. O conjunto de itens gravados por T_i não intersecciona o conjunto de dados lidos por T_j e T_i completa sua fase de gravação antes de T_j começar sua fase de gravação, ou seja, Início(T_j) $< TS(T_i) <$ Fim(T_j) e o ConjWrite(T_i) $\cap$ ConjRead(T_j) = $\emptyset$. A figura abaixo mostra a ocorrência da condição C2.

Condição C2 : Início(T_j) $< TS(T_i) <$ Fim(T_j) e ConjWrite de T_i $\cap$ ConjRead T_j = $\emptyset$



T_i

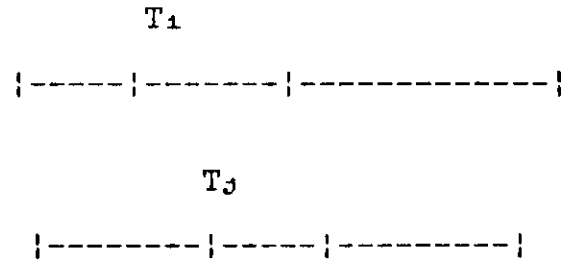
T_j

A condição C2 é para prevenir precedências da forma R_j

-> W_i. As fases de gravação são executadas serialmente, isto é, executadas em exclusão mútua. Isto também previne a ocorrência de precedências da forma W_j -> W_i.

C3. O conjunto de itens gravados por T_i não intersecciona o conjunto de dados lidos ou gravados por T_j, e T_i completa sua fase de leitura antes de T_j completar sua fase de leitura. Ou seja, $Fim(T_i) < Fim(T_j)$ e $ConjWrite(T_i) \cap (ConjWrite(T_j) \cup ConjRead(T_j)) = \emptyset$. A figura a seguir mostra a condição C3.

Condição C3 : $Fim(T_i) < Fim(T_j)$ e ConjWrite de T_i $\cap$ (ConjWrite de T_j U ConjRead de T_j) = $\emptyset$.



A condição C3 previne precedências da forma R_j -> W_i e W_j -> W_i, permitindo que haja paralelismo entre as fases de gravação.

Exemplo 2.14 Seja o seguinte schedule S com as transações T_1 e T_2, onde $TS(T_1) < TS(T_2)$:

S = $R_1(x)$; $R_2(x)$; $pW_2(x)$; $R_2(y)$; $pW_2(y)$; $R_1(y)$; C_1; C_2

onde pW(x) significa um pré-write na cópia local do objeto x. Então a fase de validação tem sucesso e o schedule S é

serializável.

Este protocolo otimista é imune aos abortos em cascata, pois as atualizações no Banco de Dados só são efetuadas depois que as transações são confirmadas através de um Commit.

Para um maior aprofundamento nesta abordagem otimista, Boksenbaum e Ceri propuseram outros algorítmos otimistas para Bancos de Dados Distribuídos. [Boksenbaum 87, Ceri 84]

2.8 Protocolo de Teste do Grafo de Serialização - SGT

Este protocolo consiste em manter um grafo de serialização que contém as operações em conflito das transações escalonadas [Bernstein 87, Sugihara 87]. O grafo de serialização difere do grafo G descrito anteriormente na seção 2.4, pois permite transações ativas. Portanto, para distinguirmos o novo grafo do G, denotaremos o primeiro por GT.

Quando uma operação $Op_i(x)$ de T_i é recebida pelo Scheduler SGT, então o seguinte algoritmo é executado [Bernstein 87] :

1. Adicione um nó T_i ao GT se ele ainda não existe
2. Para cada operação $Op_j(x)$ conflitante com $Op_i(x)$, adicionar um arco de T_j para T_i;
3. se o grafo resultante contiver ciclos
   então
      Abortar T_i
      Deletar T_i do grafo GT
   senão
      escalonar $Op_i(x)$ imediatamente
   fimse

O scheduler pode remover uma informação sobre sua transação terminada T, se e somente se, é impossível que T esteja no futuro envolvida em algum ciclo do GT. A regra para prover remoção de nodos é a seguinte: uma informação sobre uma transação pode ser removida quando a transação terminou e pertence a uma fonte, isto é, sem nenhum arco entrando, pois para que um nodo pertença a um ciclo ele deve ter pelo menos um arco saindo e outro entrando.

Este método é mais flexível que o de ordenação por Timestamp e Bloqueio em Duas fases pois, permite qualquer intercalação de Reads e Writes que são serializáveis. Porém, existe um alto "overhead" para manter o grafo GT como também para fazer periódicas verificações de ciclos, principalmente se o sistema for distribuído. Por outro lado, este método é susceptível aos abortos em cascata, visto que ele não proíbe a formação de precedências $W_i(x) \rightarrow R_j(x)$.

2.9 Conclusão

Neste capítulo abordamos toda a teoria de controle de concorrência para os Bancos de Dados Convencionais e centralizados. Também apresentamos os principais mecanismos de controle de concorrência já desenvolvidos.

A vantagem desta abordagem convencional é que ela é muito simples de implementar e, portanto, aplica-se muito bem aos modelos de dados convencionais.

Por outro lado, esta abordagem convencional não explora a semântica dos modelos, de maneira que torna-se inviável para os novos modelos de dados, por exemplo , o modelo orientado a objeto.

Portanto, para estes novos modelos precisamos utilizar tipos abstratos de dados e explorar as semânticas das operações para aumentar cada vez mais a concorrência entre transações. No próximo capítulo, abordaremos estes novos mecanismos baseados em abstração de dados.

3. O CONTROLE DE CONCORRENCIA SOBRE OBJETOS TIPADOS

3.1 Introdução

Esta abordagem para controle de concorrência baseia-se no conceito de abstração de dados, onde objetos são instâncias de tipos abstratos de dados. Para isto, os objetos são definidos como pertencentes a um determinado tipo e as operações sobre o objeto (conhecidas como métodos) fazem parte da definição do tipo de dados.

Este conceito de Tipos Abstratos de Dados surgiu na linguagem de programação Simula [Dahl 70], e é um conceito que foi absorvido pelas linguagens de programação orientadas a objetos - Smalltalk [Goldberg 83], C++ [Stroustrup 86], etc. Por conseguinte, os Bancos de Dados Orientados a Objetos [Atkinson 89] incorporaram este conceito.

No Capitulo 2, quando definimos o nosso modelo de transações, o fizemos suportando apenas as operações de Read e Write, além do Abort e Commit. Agora, precisamos estender esta definição para suportar o paradigma da orientação a objetos, permitindo a abstração de operações.

Assim, uma transação agora é vista como uma sequência ordenada de operações tipadas indivisíveis, que por sua vez são implementadas por uma sequência ordenada de operações primitivas, Read e Write, ou operações tipadas já definidas.

Muito se tem proposto na literatura, Schwarz, Garcia-

Molina e Sha, no sentido de utilizar o conhecimento semântico para aumentar o paralelismo das transações. Schwarz propôs um método de controle de concorrência que explora a comutatividade das operações tipadas utilizando a serializabilidade como critério de correção. Garcia-Molina e Sha propuseram, em seus respectivos trabalhos, a utilização de schedules não-serializáveis mas ainda assim corretos. Estes últimos utilizam como critério de correção a coerência semântica, que diz que embora um schedule não seja serializável ele é semanticamente correto. Entretanto, neste trabalho utilizaremos como critério de correção a serializabilidade [Schwarz 84a, Garcia-Molina 83, Sha 88].

O restante deste capítulo está dividido da seguinte forma: a seção 3.2 define os diversos tipos de comutatividade entre as operações tipadas; a seção 3.3 expõe uma série de exemplos, mostrando a utilidade da comutatividade em objetos tipados complexos; a seção 3.4 mostra a especificação de um protocolo de bloqueio de tipo específico; e, por último, a seção 3.5 faz uma breve conclusão do que apresentamos neste capítulo.

3.2 Comutatividade

A comutatividade é a relação utilizada no controle de concorrência sobre objetos tipados. Esta relação explora o conhecimento semântico das operações tipadas no sentido de maximizar a performance do controle de concorrência.

Utilizamos o modelo de dados proposto por Schwarz que

suporta abstração de dados. Portanto, utilizam-se operações tipadas que são implementadas em vários níveis de abstração, definidas sobre tipos primitivos ou sobre operações tipadas já definidas. Estes níveis de abstração formam a abstração de estado, onde um estado é o valor de um objeto no Banco de Dados. Neste capítulo, estes níveis não serão utilizados para o controle de concorrência, pois uma operação tipada será vista como indivisível. [Schwarz 84a]

Um objeto é composto de uma representação (classe) e de uma interface que manipula esta representação (métodos) [Atkinson 89].

Antes de definirmos a relação de comutatividade, apresentaremos a definição de estado concreto e abstrato.

Definição 3.1 Um estado de representação de um objeto no Banco de Dados se diz **concreto**, denotado por **Ec**, se representar fisicamente o objeto no Banco de Dados, isto é, é a própria estrutura de armazenamento de um objeto.

Definição 3.2 Um estado de representação de um objeto no Banco de Dados se diz **abstrato**, denotado por **Ea**, se representar a visão dada ao usuário de um estado concreto. Este estado abstrato é obtido pela execução de operações sobre estados concretos.

Um estado abstrato pode ser representado por diversos estados concretos [Moss 86] (ver exemplo 3.2).

Utilizaremos uma função f : DomEc -> DomEa, onde DomEc

é o domínio de estados concretos e DomEa é o domínio dos estados abstratos possíveis. Onde, Se Ec ∈ DomEc e Ea ∈ DomEa, então :

Ea = f(Ec) diz que Ec representa Ea.

Ao utilizarmos o conceito de abstração, precisamos definir a serializabilidade com relação a estados abstratos.

Moss denomina de serializabilidade concreta aquela baseada na equivalência entre estados concretos e denomina de serializabilidade abstrata aquela baseada na equivalência entre estados abstratos. [Moss 86]

A partir de agora, quando nos referirmos à serializabilidade leia-se serializabilidade abstrata.

Destarte, é importante definirmos a equivalência de estados abstratos:

Definição 3.3 Sejam dois estados concretos do Banco de Dados, Ec_1 e Ec_2, e seja um conjunto de operações, ConjOp, definido sobre os estados Ec_1 e Ec_2. Considere que :

Ea1 = f(Ec1) e Ea2 = f(Ec2).

Dizemos que dois estados abstratos Ea_1 e Ea_2 são equivalentes, denotado por $Ea_1 \equiv Ea_2$, se os valores retornados pela execução de qualquer Op ∈ ConjOp sobre Ec_1 forem os mesmos valores retornados pela execução de qualquer Op ∈ ConjOp sobre Ec_2.

■

Da definição 3.3 podemos ver que a equivalência depende do conjunto de operações. O exemplo a seguir foi tirado de Martin e demonstra a utilidade da definição anterior. [Martin 87]

Exemplo 3.1 Sejam dois estados concretos, Ec_1 e Ec_2, representados pelas árvores a seguir :

Ec_1 Ec_2

E sejam os seguintes conjuntos de operações definidas sobre Ec_1 e Ec_2 :

$ConjOp_1$ = {inserir, deletar, encontrar}

$ConjOp_2$ = {inserir, deletar, encontrar, obter_raiz}

Sejam Ea_1 e Ea_2 os estados abstratos obtidos pela aplicação da função f em Ec_1 e Ec_2 respectivamente, ou seja, $Ea_1 = f(Ec_1)$ e $Ea_2 = f(Ec_2)$, respectivamente. Podemos visualizar que, aplicando a definição 3.3 com relação a $ConjOp_1$, $Ea_1 \equiv Ea_2$, porém com relação a $ConjOp_2$ $Ea_1 \not\equiv Ea_2$.

■

Exemplo 3.2 Seja um schedule com dois níveis, um nível com operações sobre registros e o outro com operações sobre páginas. Assuma no nível de registros a computação do par de operações $inserir_1(x);inserir_2(y)$ e suponha que os registros x e y são inseridos na mesma página. Como as operações de inserção comutam, podemos, ao fazer a serialização do schedule, inverter a ordem das operações de inserção. Isso pode mudar também a ordem física dos registros sobre a página passando a ser y e depois x.

Entretanto, considerando o nível dos registros através de abstração, deduz-se que esta mudança de posicionamento é irrelevante a este nível. Ou seja este posicionamento é invisível para o nível de abstração dos registros. Portanto, o que existe é uma equivalência de estados abstratos no nível de registros ao efetuarmos a comutatividade das operações de inserção.

■

A seguir, damos uma definição formal do que vem a ser a comutatividade abordada neste trabalho. Antes disso, definamos duas funções que utilizaremos na definição da comutatividade.

Definição 3.4 Seja **Op** uma operação e **E** um estado concreto em que Op está definida. Definimos:

1. **Estado(Op,E)** como sendo uma função que retorna o novo estado produzido pela execução da operação Op sobre o estado E. Em outras palavras, é o estado concreto que representa o armazenamento de um objeto no Banco de Dados.

2. **Retorno(Op,E)** como sendo uma função que contém o valor de retorno da execução da operação Op sobre o estado E. Ou seja, é o efeito da execução da operação Op em relação à transação.

■

Definição 3.5 Duas operações, Op_1 e Op_2, comutam com relação a um estado E, em que Op_1 e Op_2 estão definidas, denotado por **Op_1 Comm Op_2**, se

Estado(Op_2,Estado(Op_1,E)) = Estado(Op_1,Estado(Op_2,E)) e

Retorno(Op_1,E) = Retorno(Op_1,Estado(Op_2,E)) e

$$Retorno(Op_2,E) = Retorno(Op_2, Estado(Op_1,E)).$$

∎

A definição 3.5 expressa a comutatividade conhecida como **comutatividade baseada no estado.** Isto quer dizer que para alguns estados as operações comutam e para outros não. Esta relação é simétrica. Abordaremos depois esta comutatividade, quando expusermos os outros tipos de comutatividade existentes.

No Capítulo 2, definimos o que vem a ser um conflito entre operações. Podemos, agora, relacionar o conceito de conflito com a comutatividade vista neste capítulo. Dizemos que duas operações conflitam se elas não comutam. Denotaremos o conflito entre duas operações, Op_1 e Op_2 , por **Conflita(Op_1,Op_2).** Em termos da definição 3.5 temos que:

$$Conflita(Op_1,Op_2) = not(\forall E\ Op_1\ Com_E\ Op_2)$$

$$\Leftrightarrow Existe\ E/\ not(Op_1\ Com_E\ Op_2)$$

A relação de comutatividade é de fundamental importância no aprimoramento do controle de concorrência no sentido de que, aplicando a comutatividade, baseada na semântica das operações tipadas, podemos ter schedules corretos que eram tidos como não-serializáveis no capítulo 2.

Esta comparação pode ser feita se considerarmos que uma operação de consulta pode ser modelada como uma operação de leitura e uma operação de atualização pode ser modelada como uma operação de escrita ou uma operação de leitura seguida por uma de escrita. O exemplo a seguir mostra esta situação.

Exemplo 3.3 Seja um tipo abstrato de dados Contador que possui a operação tipada de incremento, denotada por Inc(X). Seja S um schedule contendo duas transações T_1 e T_2 que incrementam, cada uma, dois objetos X e Y , contendo a seguinte execução:

T_1 = $Inc_1(X)$; $Inc_1(Y)$; C_1

T_2 = $Inc_2(Y)$; $Inc_2(X)$; C_2

S = $Inc_1(X)$; $Inc_2(Y)$; $Inc_2(X)$; $Inc_1(Y)$; C_1; C_2

onde $Inc_1(X)$ significa que é uma operação da transação i. Então aplicando a definição 3.5 temos que :

$\forall$ E, $Inc_1(X)$ Com$_E$ $Inc_j(X)$.

Nota: Neste caso não há teto para o objeto x.

Assim, duas operações de incremento são comutativas. Portanto, o grafo de serialização não contém arcos que indicam precedências e, portanto, é acíclico. Desta forma, o schedule S é serializável e equivalente as execuções $T_1;T_2$ ou $T_2;T_1$.

Por outro lado, se modelássemos esta execução com operações de Read/Write teríamos :

T_1 = $R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1

T_2 = $R_2(Y)$; $W_2(Y)$; $R_2(X)$; $W_2(X)$; C_2

S = $R_1(X);W_1(X);R_2(Y);W_2(Y);R_2(X);W_2(X);R_1(Y);W_1(Y);$ C_1; C_2

O grafo de serialização, G, para o schedule S seria :

$$G = T_1 \underset{<---}{\overset{x}{--->}} T_2$$
$$y$$

É de se notar que o grafo de serialização contém um ciclo devido aos conflitos sobre os objetos X e Y. De forma que o schedule S seria tido, no modelo apresentado no capítulo 2, como não-serializável.

■

Beeri classifica a comutatividade em três tipos : comutatividade geral, comutatividade baseada no estado e comutatividade parametrizada, cognominada por Cart de comutatividade condicional. A seguir, descreveremos estes três tipos e daremos exemplos de cada um. [Beeri 83, Cart 89]

1. **Comutatividade Geral** => É o tipo mais restritivo, pois não utiliza o estado de representação dos objetos, nem tão pouco parâmetros. Neste tipo de comutatividade as operações devem comutar para todos os estados em que estejam definidas. Portanto, este tipo de comutatividade é o que oferece menor concorrência. Expressando esta comutatividade com a definição 3.5 temos :

$\forall E, Op_1 \text{ Com}_E Op_2.$

Exemplo 3.4 Consideremos duas transações que efetuam operações de leitura sobre um objeto x, denotadas por $R_1(x)$ e $R_2(x)$. Então, qualquer que seja a ordem de execução, estas duas operações comutam, independentemente do valor do objeto x.

■

2. **Comutatividade baseada no estado** => E a comutatividade da definição 3.5 em que se utiliza o estado de representação do objeto para definir a comutatividade. E uma relação que oferece mais concorrência do que comutatividade geral, pois não requer que as operações comutem para todos os estados, e é a comutatividade menos restritiva. Por outro lado, é de difícil implementação, pois requer o conhecimento prévio do estado do objeto. Portanto, impõe um maior "overhead" ao scheduler.

Exemplo 3.5 Consideremos um tipo Contador, com teto máximo, com operações de incremento, Inc(x). Em outras palavras, quando uma operação Inc(x) é invocada a seguinte condição é verificada:

```
Inc(x) :
  se x < Teto_Máximo
     então x := x + 1
  senão Retorne(erro)
```

Seja um schedule de duas transações com a seguinte ordem de execução : $Inc_1(x);Inc_2(x)$. Então, estas operações de incremento comutam dependendo do estado anterior a elas, pois para que cada operação de incremento seja executada, é necessário saber se o estado E, anterior àquelas, é menor ou igual ao (Teto_Máximo - 2), pois caso contrário, uma das operações será rejeitada. Vemos que esta comutatividade não pode ser geral. A seguir, veremos que ela também não pode ser parametrizada.

■

3. **Comutatividade parametrizada** => Utiliza os parâmetros de chamada e/ou de retorno das operações. Ao contrário da comutatividade baseada no estado, é de fácil implementação visto que os parâmetros a serem utilizados estão contidos na própria chamada ou retorno da operação. Portanto, incorre num "overhead" bem reduzido quando comparado à comutatividade baseada no estado. Por ser parametrizada, as operações sobre objetos estão sempre definidas para qualquer que seja o estado de representação do objeto. Expressando a comutatividade parametrizada com a definição 3.5 temos :

$\forall$ E / Cond(p,q) então $Op_1(p)$ Com_E $Op_2(q)$

onde **p** e **q** são parâmetros de chamada/retorno e **Cond(p,q)** é um predicado que avalia se a condição requerida aos parâmetros p e q é satisfeita.

Damos a seguir três exemplos em que mostramos a comutatividade dependente dos parâmetros de chamada, em seguida a comutatividade dependente dos parâmetros de retorno e, por último, a comutatividade dependente dos parâmetros de chamada e de retorno.

Exemplo 3.6 Sejam duas operações primitivas de escrita: W(X,v1) e W(X,v2) onde v1 e v2 são parâmetros de chamada que contêm os valores a serem escritos no objeto X. Então dizemos que estas duas operações comutam se v1 = v2, isto é, as operações escrevem um mesmo valor em X. Portanto, a comutatividade depende dos parâmetros de chamada. Neste caso, $\forall$ E/ W(X,v1) Com_E W(X,v2) com (v1 = v2).
■

Exemplo 3.7 Seja um conjunto de recursos R onde temos as funções:

1. Aloc(R,Ok), que aloca um recurso, caso haja um disponível e retorna com o flag Ok = verdadeiro, caso contrário, isto é, caso todos os recursos estejam ocupados, então retorna com o flag Ok = falso; e

2. Libera(R) que libera o recurso previamente bloqueado. Então podemos ver que a sequência (Aloc(R,Ok$_1$); Aloc(R,Ok$_2$)) comuta somente quando Ok$_1$ = Ok$_2$. Isto quer dizer que :

- quando Ok$_1$ = Ok$_2$ = verdadeiro, então existem ao menos dois recursos disponíveis para alocação; e

- quando Ok$_1$ = Ok$_2$ = falso, então é porque todos os recursos já estão alocados e não há recursos disponíveis naquele instante. Portanto, a comutatividade depende dos parâmetros de retorno; neste caso, $\forall$ E/ (Ok$_1$ = Ok$_2$) Op$_1$ Com$_E$ Op$_2$

onde as condições que caracterizam os estados iniciais são:

- quando todos os recursos estão alocados ou
- quando existe ao menos dois recursos disponíveis.

■

Exemplo 3.8 Sejam novamente as operações primitivas parametrizadas de leitura, R(X,v$_1$), e de escrita, W(X,v$_2$), onde v$_1$ é parâmetro de retorno e v$_2$ é parâmetro de chamada. Considere a ordem de execução R(X,v$_1$); W(X,v$_2$) então estas duas operações comutam se v$_1$ = v$_2$, isto é , o valor escrito é o mesmo que o valor lido antes da operação de escrita. Portanto, a comutatividade depende dos parâmetros de chamada e de retorno. Neste caso, $\forall$ E/ (v1 = v2) então R(X,v1) Com$_E$ W(X,v2) <=>

$\forall$ E/ E = V2 = V1 R(X,v1) Com_{v1} W(X,v2).

■

Schwarz utilizou a comutatividade parametrizada para definir as precedências entre operações tipadas complexas (diretórios, filas, pilhas, etc.). Veremos alguns exemplos na próxima seção. [Schwarz 84a, 84b]

Weihl classifica a comutatividade em forward e backward. Damos a definição das comutatividades forward e backward em função da definição 3.5. [Weihl 88, 89]

4. **Comutatividade Forward** [Weihl 89]

Definição 3.6 Sejam P e Q valores de retorno possíveis. Dizemos que duas operações, Op_1 e Op_2, comutam forward , denotado por Op_1 **ComFor**$_{P,Q}$ Op_2, se :

$\forall$ E / Retorno(Op_1,E) = P e Retorno(Op_2,E) = Q

Op_1 Com_E Op_2.

Daremos um exemplo após a definição a seguir.

■

5. **Comutatividade Backward** [Weihl 89]

Definição 3.7 Seja E um estado, sejam P e Q valores de retorno possíveis. Dizemos que duas operações, Op_1 e Op_2, comutam backward, denotado por Op_1 **ComBac**$_{P,Q}$ Op_2 se:

$\forall$ E / Retorno(Op_1,E) = P e Retorno(Op_2,E´) = Q

e Op_1 Com_E Op_2 com E´ = Estado(Op_1,E). ■

Das duas definições acima podemos concluir que :

1. A comutatividade forward é uma relação simétrica, enquanto que a comutatividade backward não o é.

2. Na comutatividade forward, as duas operações devem estar definidas sobre o estado anterior às mesmas, querendo isto dizer que a ordem de execução não importa.

3. Na comutatividade backward, a primeira operação deve estar definida sobre o estado E, anterior a mesma, mas a segunda operação deve estar definida sobre o estado inicial E como também sobre o estado E´, deixado pela execução da primeira operação. Portanto, a ordem de execução é importante.

O exemplo a seguir foi retirado de Weihl e faz um comparativo entre as comutatividades forward e backward. [Weihl 88]

Exemplo 3.9 Consideremos uma aplicação bancária na qual temos três operações sobre uma conta bancária:

- **deposito(x)**, que corresponde a um depósito de uma quantia x, com x > 0;
- **saque(x, Ok)**, que corresponde a uma retirada de uma quantia x, com x > 0; e
- **saldo(x)**, que corresponde a uma consulta do saldo, onde x é o valor de retorno.

Sendo E_a o estado anterior e E_r o estado resultante , que representam o saldo da conta, temos as seguintes definições das operações:

deposito(x) -> depósito cuja pós-condição é $E_r = E_a + x$.

saque(x,Ok) -> operação de saque , temos:

Se Ok = true então o saque teve sucesso e temos como

Pré-condição : $E_a >= x$ e

Pós-condição : $E_r = E_a - x$

Se Ok = false então o saque não foi executado, pois não tinha fundos suficientes na conta x e temos como

Pré-condição : $E_a < x$ e

Pós-condição : $E_r = E_a$

(saldo,x) -> consulta ao saldo:

Pré-condição : $E_a = x$

Pós-condição : $E_r = E_a$.

Então,

1. Considere duas operações de saque, Op_1 = (saque(x),Ok_1) e Op_2 = (saque(y),Ok_2), com $Ok_1 = Ok_2$ = true, isto é, P = Q = true e seja E um estado. Então dizemos que :

. não (Op_1 $ComFor_{P,Q}$ Op_2) <=>

não ($\forall$ E/ E $\geq$ max(x,y) e E $\leq$ x + y Op_1 Com_E Op_2)

<=> Existe E / E $\geq$ max(x,y) e E $\leq$ x + y não (Op_1 Com_E Op_2), onde max(x,y) é uma função que retorna o maior valor entre x e y.

Isto quer dizer que existe um estado E em que ambas as operações, Op_1 e Op_2, retornam Ok1 = Ok2 = true, porém este mesmo estado pode ser inferior à soma dos saldos x e y, fazendo com que a segunda operação a ser executada tenha o parâmetro de retorno

Ok = false. Portanto, Op_1 e Op_2 não comutam forward.

. Op_1 $ComBac_{P,Q}$ Op_2 <=> ∀ E/ E ≥ X + Y com Op_1 Com_E Op_2

Em outras palavras, Op_1 e Op_2 comutam backward nesta ordem de execução, pois se os parâmetros de retorno P e Q são verdadeiros, isto quer dizer que o estado anterior às operações, isto é, o saldo inicial, é superior ou igual à soma dos dois saques x e y, a primeira operação executada está definida sobre um estado E e a segunda está definida sobre E e sobre E´, que é o estado resultante deixado pela execução da primeira.

2. Seja Op_1 = saque(x),Ok_1 com Ok_1 = true, isto é uma operação de saque com sucesso, e seja Op_2 = saque(y),Ok_2 com Ok_2 = false, isto é, uma operação de saque sem sucesso. Seja E um estado que representa o saldo da conta bancária. Temos que P = true e Q = false, então

. Op_1 $ComFor_{P,Q}$ Op_2 <=> ∀ E/ x ≤ E < y Op_1 Com_E Op_2

Isto quer dizer que, como o estado inicial E é maior do que o saque de Op_1, pois Ok1 = true, e menor do que o saque de Op_2, pois Ok2 = false, então qualquer que seja a ordem de execução de Op_1 e Op_2, os parâmetros de retorno permanecerão inalterados e o efeito sobre o objeto conta bancária é o mesmo, isto é, apenas o valor x será sacado da conta.

. não(Op_1 $ComBac_{P,Q}$ Op_2), nesta ordem de execução pois

<=> não (∀ E, ∀ E´/ E´ < y e E ≥ y Op_1 Com_E Op_2)

<=> Existe E, Existe E´/ E´ < y e E ≥ y não(Op_1 Com_E Op_2) e E´ = Estado(Op_1,E).

Em outras palavras, Op_1 e Op_2 não comutam backward nesta ordem, porque se invertermos a ordem de execução, pode ser que Op_2 passe a ter valor de retorno Ok2 = true, caso o saldo inicial da conta, E, seja maior do que y. Enquanto que, forçosamente, Op_1 passa a ter valor retorno Ok1 = false, pois não haverá fundos suficientes em E´, que é o saldo deixado após a execução de Op_2, para o saque do valor x.

3. Depósitos e saques com sucesso não comutam de forma alguma com a operação de consulta de saldo, visto que as primeiras modificam o estado.

4. Sejam Op_1 = deposito(x) e Op_2 = deposito(y), isto é duas operações de depósito. Então,

. $\forall$ E Op_1 $ComFor_E$ Op_2

. $\forall$ E Op_1 $ComBac_E$ Op_2

As operações Op_1 e Op_2 não possuem parâmetros de retorno e, desde que não existe um teto máximo para um valor de uma conta, elas obedecem à comutatividade geral e, portanto, comutam tanto forward quanto backward.

∎

Para efeito de aplicabilidade Weihl diz que a comutatividade forward é utilizada em modelos com atualizações deferidas, enquanto que a comutatividade backward é utilizada nos modelos com atualizações imediatas. Não entraremos em detalhes quanto ao modelo de atualização, visto que vai além do escopo deste trabalho. Entretanto, o leitor pode consultar Weihl e Turc

para um maior aprofundamento no assunto. [Weihl 89, Turc 90]

Neste trabalho, utilizaremos diversos tipos de comutatividade e, em cada vez, explicitaremos qual o tipo utilizado.

3.3 Utilização da Comutatividade em Objetos Tipados Complexos

A utilização de tipos abstratos de dados permite que sistemas complexos sejam facilmente entendidos pela modularização e estruturação destes em componentes mais simples.

Nesta seção, exploraremos a comutatividade em objetos tipados complexos tais como : Filas e Pilhas [Schwarz 84a, 84b].

Daremos exemplos de como este modelo transacional suportando abstração de dados pode aumentar a performance do controle de concorrência.

Exemplo 3.10 Seja um tipo abstrato **Fila** FIFO, cujas operações são: InserirFila$_i$(F,x), significando que a transação T$_i$ adiciona o elemento x ao fim da fila F; e RemoverFila$_i$(F,x), que significa que a transação T$_i$ retira o elemento x do início da fila F. Caso a fila esteja vazia, esta última operação é bloqueada e permanece esperando até chegar um elemento à fila.

Como podemos notar, estas operações sobre o tipo fila F são parametrizadas, de modo que F é o objeto do tipo fila compartilhado entre várias transações e x é o parâmetro que queremos inserir/retirar da fila F.

Para garantir serializabilidade, as inserções na fila feitas por transações que ainda não foram confirmadas não devem ser observadas por outras transações (propriedade de isolação). Denotando as operações InserirFila$_i$(x) e RemoverFila$_i$(x) como Ins$_i$(F,x) e Rem$_i$(F,x) respectivamente, temos as possíveis ordens de execução para o tipo fila:

O_1: Ins$_i$(F,x) ; Ins$_j$(F,y); T_j insere um elemento y na fila F depois que T_i inseriu um elemento x.

O_2: Ins$_i$(F,x) ; Rem$_j$(F,y); T_j remove o elemento y da fila F depois que T_i inseriu o elemento x.

O_3: Ins$_i$(F,x) ; Rem$_j$(F,x); T_j remove o elemento x da fila F que foi inserido por T_i .

O_4: Rem$_i$(F,x) ; Ins$_j$(F,y); T_j insere o elemento y na fila F depois que T_i removeu o elemento x.

O_5: Rem$_i$(F,x) ; Rem$_j$(F,y); T_j remove o elemento y da fila F depois que T_i removeu o elemento x.

Analisando as possíveis ordens de execução acima, vemos que os conflitos entre as operações existem em O_1, O_3 e O_5, pois não permitem alterar a ordem de execução sem que o estado final seja alterado. Enquanto que as demais ordens : O_2 e O_4 são insignificantes, pois seus respectivos pares de operações comutam (comutatividade parametrizada).

Seja E um estado. Então, temos os seguintes conflitos :

C_1: Conflita (Ins$_i$(F,x),Ins$_j$(F,y))

C_2: Conflita ((Ins$_i$(F,x),Rem$_j$(F,x))

C_3: Conflita (($Rem_i(F,x)$,$Rem_j(F,y)$))

Fazendo uma analogia à definição 3.5 temos que:

- o conflito C_1 dá-se pelo fato de que a ordem de execução de duas operações de inserção numa fila é importante, pois, como trata-se de uma estrutura FIFO, os estados diferem ao invertermos a ordem de execução destas duas operações;

- o conflito C_2 dá-se porque se invertermos a ordem das operações teremos valores retornados diferentes e o efeito sobre o Banco de Dados também diferirá. Ademais este conflito evita o aborto em cascata visto que uma operação de inserção executada por uma transação não confirmada não pode ser removida da fila;

- o conflito C_3 existe porque, ao invertermos a ordem de execução de duas operações de remoção, os valores retornados pelas mesmas serão também invertidos.

■

Vemos que este tipo de modelagem abstrata, na qual utilizamos a semântica das operações, provê uma maior concorrência do que se utilizássemos apenas a semântica de read/write descrita anteriormente [Schwarz 84a]. Neste último caso, modelaríamos a operação de $Inserirfila_i(F,x)$ como uma operação read(F) seguida de um $write_i(F)$ e a operação $RemoverFila_i(F,x)$ como uma operação de $read_i(F)$ seguida de uma operação de $write_i(F)$.

Procedendo desta forma, estaríamos com um controle de concorrência mais restritivo, visto que os conflitos seriam ,

como já vimos, Read-Write, Write-Read e Write-Write, que restringiriam desnecessariamente o tipo abstrato fila.

Neste caso, as cinco possíveis ordens de execução listadas anteriormente, O_1, O_2, O_3, O_4 e O_5, seriam conflitantes, pois ou se recairia num conflito write-write ou num conflito read-write.

Exemplo 3.11 Seja o seguinte schedule S utilizando o tipo abstrato Fila descrito no exemplo 3.10, com a fila F inicialmente vazia :

$$S_1 = Ins_1(F,x);Ins_2(F,y);Rem_1(F,x);C_1;C_2$$

Como podemos notar, este schedule é serializável pois não existem conflitos.

Suponhamos agora o seguinte schedule de um tipo Fila contendo inicialmente os valores a e b :

$$S_2 = Ins_1(F,x);Ins_2(F,y);Rem_2(F,a);Rem_1(F,b);C_1;C_2$$

Como podemos ver existe um conflito entre as duas operações de inserção com os parâmetros x e y e existe também um conflito entre as duas operações de remoção. Existe, portanto, um ciclo $T_1 \rightarrow T_2 \rightarrow T_1$ no grafo de serialização.

■

Exemplo 3.12 Consideremos agora um schedule com transações contendo operações pertencentes a distintos tipos de dados, por exemplo, dois tipos abstratos : **Pilha P**, com operações Push(P,x) que coloca o elemento x na pilha P, e **Fila F**, cuja

definição foi dada no exemplo 3.10. Seja o seguinte schedule :

$$S = Push_1(P,x);Push_2(P,y);Ins_2(F,w);Ins_1(F,z)$$

As duas operações $Push_1(P,x)$ e $Push_2(P,y)$ são conflitantes, pois se invertermos a ordem de execução destas, teremos estados diferentes da pilha. Como mostramos anteriormente, há um conflito entre as operações de inserção do tipo fila; então o schedule S não é serializável, pois existe um ciclo no grafo de serialização: $T_1 \xrightarrow{P} T_2 \xrightarrow{F} T_1$.

■

3.4 O Bloqueio em Duas Fases de Tipo Específico

Abordaremos nesta seção uma maneira de implementar o controle de concorrência sobre objetos tipados utilizando, para isso, um protocolo chamado Bloqueio de Tipo Específico.

O bloqueio é por si só uma técnica muito restritiva com relação à concorrência. Porém sua semântica já garante que não haja ciclos de precedências, pois toda vez que uma operação vai ter acesso a um objeto ela deve bloqueá-lo, não permitindo que outra operação conflitante tenha acesso ao objeto, até que este seja liberado.

Mencionamos no capítulo anterior o protocolo de bloqueio em duas fases que utiliza a semântica de read/write, definindo para isso dois tipos de bloqueios: ReadLock, bloqueio de leitura, e Writelock, bloqueio de gravação.

Já o protocolo de Bloqueio de Tipo Específico utiliza a informação semântica expressada nas operações tipadas.

Para implementar esta técnica, utilizamos uma tabela de compatibilidade de bloqueios para cada tipo abstrato. Esta tabela define a **comutatividade** entre as operações tipadas, isto é, a tabela informa quais operações são (ou não são) conflitantes. Ela é de tal forma que as linhas e colunas representam as operações possíveis definidas para o tipo ao qual o objeto pertence. Cada entrada da tabela diz se uma determinada operação da linha i comuta com uma outra operação que está numa coluna j.

A primeira linha da tabela representa bloqueios assegurados e a primeira coluna representa pedidos de bloqueio.

Exemplo 3.13 Ilustramos inicialmente com a semântica de Read/Write, cujos conflitos, já mencionados, são : Read-Write, Write-Read, Write-Write. Para tal abordagem, teríamos a seguinte tabela de comutatividade geral para o objeto x:

bloqueio assegurado

bloqueio pedido	X	R	W
	R	Sim	Não
	W	Não	Não

onde uma entrada "Sim" indica que as operações comutam e uma entrada "Não" indica que as operações são conflitantes.

A tabela acima expõe as compatibilidades de modo que a única entrada da tabela que contém operações comutativas é aquela

que possui duas operações de leitura, sendo as demais entradas com operações conflitantes.

■

Exemplo 3.14 Vamos ilustrar agora como este protocolo poderia ser implementado com um tipo Fila, cujas operações já foram descritas anteriormente no exemplo 3.10 : InserirFila(x) e RemoverFila(x). Haveria duas classes de bloqueio, uma para cada operação, que seriam respectivamente : LockIns e LockRem. O protocolo funcionaria da seguinte forma :

1. Para executar uma operação de InserirFila(x) deve-se obter antes um LockIns. Este bloqueio é assegurado até o fim da transação. (Protocolo 2PL Estrito)

2. Para executar uma operação RemoverFila(x) deve-se obter antes um LockRem. Este bloqueio é assegurado até o fim da transação (2PL Estrito também).

A tabela de comutatividade parametrizada para este tipo seria a seguinte :

bloqueio assegurado

bloqueio pedido	F	Ins(x)	Rem(x)
	Ins(x)	NA	NA
	Ins(y)	Não	Sim
	Rem(x)	Não	NA
	Rem(y)	Sim	Não

onde **NA** significa Não Aplicável. Uma entrada **Sim** indica que as operações são comutativas e uma entrada **Não** indica que as mesmas são conflitantes.

Consideremos que cada parâmetro x tem um identificador único, de modo que existe um só parâmetro de nome x na fila.

Portanto, de acordo com os conflitos do tipo fila citados anteriormente temos que :

- na coluna 1, linha 1, é NA, pois consideramos que um elemento possui um identificador único. Deste modo é impossível tentar inserir na fila dois elementos de mesmo identificador;

- na coluna 2, linha 1, se existe apenas um elemento x na fila, este elemento só poderá ser reinserido quando for retirado da fila. Portanto, esta entrada da tabela é NA, pois é semanticamente imposível;

- na coluna 1, linha 2, está expressado o conflito Conflita(Ins(F,x), Ins(F,y));

- na coluna 2, linha 2, está expressada a ordem Rem(F,x);Ins(F,y), que como vimos é comutativa. Portanto, ∀ E Rem(F,x) Comᴇ Ins(F,y).;

- na coluna 1, linha 3, está expressado o conflito Conflita(Ins(F,x), Rem(F,x));

- na coluna 2, linha 3, de maneira análoga à inserção, se existe apenas um elemento x na fila, este elemento só poderá ser retirado por uma única transação. Portanto, esta entrada é NA, pois é sematicamente impossível;

- na coluna 1, linha 4 está expressada a ordem Ins(F,x);Rem(F,y) que, como vimos, é comutativa. Portanto, ∀ E

$Ins(F,x)$ Com_E $Rem(F,y)$.

- finalmente, na coluna 2, linha 4, está expressado o conflito $Conflita(Rem(F,x),Rem(F,y))$.

Aplicando os bloqueios de tipo específico ao exemplo 3.11 cujo schedule é $S_1 = Ins_1(F,x);Ins_2(F,y);Rem_1(F,x)$ temos que: a transação T_1 bloqueia o objeto F utilizando um parâmetro x. Em seguida, a transação T_2 tenta bloquear o objeto F utilizando um parâmetro y, mas este bloqueio não é permitido porque, conforme a tabela de comutatividade, duas operações de inserção não comutam, portanto, a transação T_2 fica esperando pela liberação do objeto F. Em seguida, a transação T_1 executa a operação de remoção e libera o objeto F após confirmar. Após a liberação do objeto F, a transação T_2 obtém um bloqueio sobre o mesmo e executa sua operação de inserção. Por último, a transação T_2 , após confirmar, libera o objeto F.

■

3.5 Conclusão

As operações tipadas têm papel fundamental nos novos modelos de Bancos de Dados conhecidos como Não-Convencionais.

O interesse maior da utilização dos tipos abstratos de dados é justamente buscar, através da exploração da semântica das operações tipadas, mecanismos de controle de concorrência que aumentem ao máximo o paralelismo entre as transações.

Esta exploração semântica dá-se através da aplicação da relação de comutatividade que tem como objetivo reduzir o número de arcos no grafo de serialização, permitindo assim uma maior concorrência.

A desvantagem desta abordagem é que as operações tipadas são indivisíveis, isto é, executadas em exclusão mútua. Portanto, um maior paralelismo seria obtido se conseguíssemos prover concorrência entre operações tipadas. Neste sentido, surgiu a abordagem multinível que visa preencher esta lacuna deixada no controle de concorrência sobre operações tipadas, aumentando ainda mais o paralelismo.

O próximo capítulo aborda a teoria da arquitetura multinível.

4. CONTROLE DE CONCORRENCIA MULTINIVEL

4.1. Introdução

Um sistema multinível visa justamente prover um maior paralelismo ao controle de concorrência. Além disso, como todo mecanismo de controle de concorrência, um sistema multinivel tenta reduzir ao máximo o distanciamento entre o controle de concorrência e o mecanismo de recuperação. Esta preocupação de integrar controle de concorrência com recuperação vem desde os Bancos de Dados Convencionais (cf. capitulo 2), onde sempre se procurou expor uma versão estrita dos diversos mecanismos apresentados, cujo objetivo era prover o sistema de um mecanismo de recuperação eficiente.

Segundo Beeri, as ações de recuperação têm acesso a dados compartilhados como o fazem as operações normais. Portanto, elas devem preservar a serializabilidade, devendo existir uma relação cooperativa, e porque não dizer conjunta, entre controle de concorrência e recuperação. [Beeri 88]

Para tanto, num sistema multinível, quando uma transação é abortada, utilizam-se operações inversas que simplesmente anulam o efeito das operações executadas pela transação abortada [Beeri 88, Weikum 87].

Um sistema multinível é uma decomposição de transações em vários níveis. Um conceito imprescindível nessa abordagem é a **pseudo-indivisibilidade**, que quer dizer que uma operação deve comportar-se como se fosse executada sozinha, sem intercalação

com outras operações. Aliado ao conceito de pseudo-indivisibilidade, temos os conceitos de **comutatividade** e **abstração** que são largamente utilizados nessa abordagem multinível. Estes últimos foram descritas no capítulo precedente.

Segundo Moss, as transações multiníveis são uma extensão do modelo clássico de transações (cf. capítulo 2), visto que este último, como mencionamos antes, é de um só nível. Desta maneira, uma transação é dividida em subtransações e estas últimas, por sua vez, são divididas em subtransações, e assim por diante. Com isso, forma-se uma hierarquia do nível mais abstrato, o nível das transações, para o nível físico, as operações indivisíveis sobre as páginas do Banco de Dados. [Moss 82]

Veremos no decorrer deste capítulo que, com o emprego da abordagem multinível, podemos ter schedules tidos por protocolos convencionais como não-serializáveis, mas que, com a exploração da semântica e da concorrência das operações, são tidos como serializáveis nesta abordagem. De modo que há um ganho para o controle de concorrência, porque reduz-se o espectro de schedules inconsistentes.

Como exemplo deste ganho das transações multiníveis sobre as transações convencionais, 1 só nível, temos que, se uma subtransação aborta, a transação como um todo continua executando. A subtransação que é ancestral imediata da subtransação abortada deve recuperar, reexecutar ou até mesmo ativar outro procedimento que realize a mesma função da subtransação abortada. Além desta vantagem na recuperação das

transações, o interesse de estudar as transações multiníveis dá-se pela incorporação de operações tipadas pseudo-indivisíveis, explorando tanto a semântica destas operações, quanto o paralelismo interno da transação. Por isto, esta abordagem multinível é bastante indicada para sistemas distribuídos, Bancos de Dados Orientados a Objeto e aplicações não-convencionais.

Feita esta introdução, apresentaremos na seção 4.2 o modelo a ser utilizado, na seção 4.3 abordaremos a construção do grafo de precedência, na seção 4.4 enfocaremos o critério de correção da abordagem multinível, na seção 4.5 daremos alguns exemplos e, por último, na seção 4.6 faremos uma breve conclusão deste capítulo.

4.2. O Modelo de Transação Multinível

Segundo Beeri, no modelo clássico de transações, uma transação invoca operações. Por sua vez, num sistema multinível uma transação pode invocar também subtransações, possivelmente aninhadas, formando uma estrutura hierárquica similar a uma árvore conhecida na literatura [Cart 89] como árvore de decomposição. A terminologia de árvore será empregada : pai, filhos, raiz, folhas, etc. [Beeri 83]

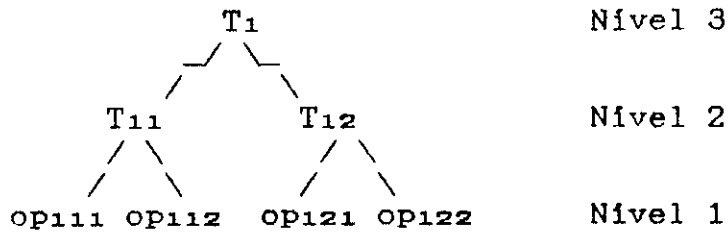
Portanto, uma transação multinível é vista como uma árvore, onde a raiz, conhecida também como nível topo, é a transação no nível mais abstrato e as folhas são as operações indivisíveis sobre as páginas do Banco de Dados. Nos níveis intermediários, entre a raiz e as folhas, estão as subtransações

que, por sua vez, são operações abstratas implementadas como se fossem transações, sendo que quanto mais nos dirigimos no sentido da raiz, maior será a abstração.

Por questão de simplificação, apresentamos a teoria da arquitetura multinível supondo que a árvore de decomposição é equilibrada, de maneira que as operações primitivas estão num mesmo nível de profundidade [Weikum 87]. A consequência desta hipótese sobre o modelo de dados é que os objetos devem ter o mesmo número de níveis de abstração.

Para rotularmos os diversos níveis de uma transação, adotaremos a seguinte convenção : o nível topo recebe o rótulo n, onde n é o número de níveis, isto é a profundidade da árvore. A medida que se vai descendo na árvore, decrementa-se uma unidade, dando assim o rótulo do respectivo nível. Desse modo, o nível folha possui rótulo 1. O exemplo a seguir mostra-nos uma árvore de decomposição.

Exemplo 4.1 Suponhamos a seguinte transação de 3 níveis:



Nesta árvore de decomposição acima podemos ver a transação T_1, as subtransações T_{11} e T_{12}, que implementam de maneira abstrata a transação T_1, e por último, as operações op_{111}, op_{112}, op_{121} e

op122 que implementam as subtransações T11 e T12 e que são as operações primitivas de read/write sobre as páginas do Banco de Dados.

■

Apresentamos a seguir o conceito de pseudo-indivisibilidade que é largamente utilizado na abordagem multinível.

Definição 4.1 Uma operação é **pseudo-indivisível** se, embora tendo suas suboperações intercaladas com outras operações, comporta-se como se fosse executada sozinha, e se suas suboperações são pseudo-indivisíveis.

Nota: Na literatura o termo pseudo-indivisibilidade é mais conhecido como atomicidade; utilizamos este novo termo para que não haja confusão com a propriedade de atomicidade descrita no capítulo 2 e que será utilizada também neste capítulo.

■

Em outras palavras, a definição 4.1 diz-nos que uma operação pseudo-indivisível **comporta-se** como se fosse indivisível, isto é, executada em exclusão mútua com as demais operações do mesmo nível que têm acesso ao mesmo objeto. Vemos que uma subtransação é uma implementação de uma operação abstrata em nosso modelo.

Por sua vez, uma operação é uma abstração de um conjunto de suboperações que realizam, num nível de menor abstração, a função da operação. Portanto, uma operação no nível

i nada mais é que o conjunto de suboperações no nível **i-1** e assim por diante.

Por exemplo, uma operação tipada do tipo Contador com uma operação de incrementação sobre um objeto x, denotado por Inc(x), é uma abstração da operação Update(x), que, por sua vez, é uma abstração das operações sobre páginas R(p) e W(p), onde a página p armazena o objeto x.

Uma operação num determinado nível **i**, Op_i, começa depois de seu pai ter iniciado sua execução no nível **i+1** e termina quando todas as suboperações invocadas por Op_i, isto é todas $Op_{(i-1)} \in$ filhos(Op_i), terem terminado.

Existe uma ordem entre duas operações Op_1 e Op_2, denotado por $Op_1 < Op_2$ (lê-se Op_1 é executado antes que Op_2), se e somente se $\forall\ Op_{1i}, Op_{2j} / Op_{1i} \in$ filhos(Op_1) e $Op_{2j} \in$ filhos(Op_2) temos que $Op_{1i} < Op_{2j}$. Denominamos esta ordem de **ordem estática**, isto é, a ordem de execução expressada pelo programador da transação.

Em outras palavras, isto quer dizer que uma operação é ordenada antes de outra, se todos os seus filhos completam sua execução antes de qualquer filho de outrem tenha iniciado sua execução [Martin 87].

Formalmente, podemos definir uma operação da seguinte forma:

Op_i = (filhos(Op_i),<) = ($Op_{(i-1)}$/ $Op_{(i-1)} \in$ filhos(Op_i),<),

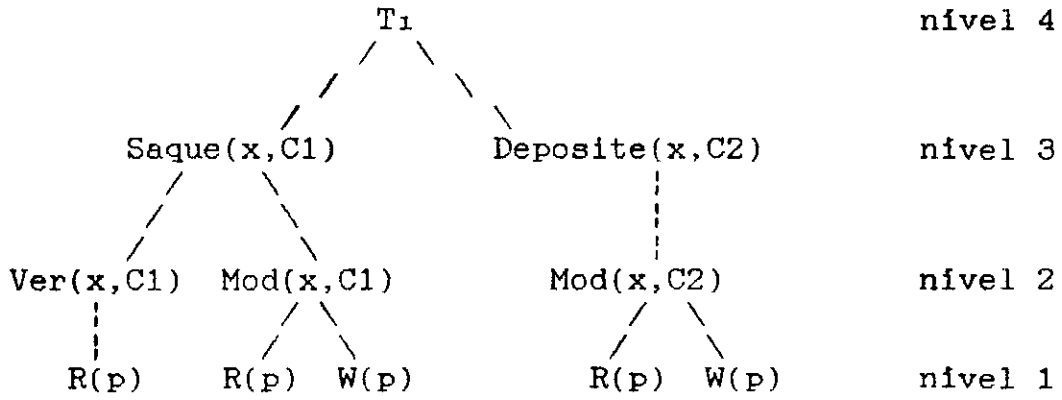
onde < é a ordem estática de execução da operação

(subtransação).

Uma outra ordem utilizada, denotada por <<, é a **ordem dinâmica**, que expressa a ordem de execução das operações folhas.

Damos em seguida um exemplo de como essa abstração funciona na abordagem multinível.

Exemplo 4.2 Suponhamos uma aplicação bancária com a seguinte transação: T_1 = Saque(x,C1); Deposite(x,C2); que significa uma transferência de fundos, cujo valor é x, da conta C1 para conta C2. Modelando esta transação numa estrutura hierárquica multinível temos:



Na árvore acima, podemos notar que a transação possui quatro níveis. A operação Saque(x,C1) é modelada como uma seleção da conta C1, Ver(x,C1), que consiste em verificar se existem fundos suficientes para se efetuar a operação de saque (supondo que saldos negativos sejam proibidos), seguido de um Mod(x,C1), que consiste em modificar a conta C1 com o novo saldo, i.é. saldo anterior menos x. De modo análogo, a operação Deposite(x,C2) é modelada com um Mod(x,C2) que atualiza a conta C2.

Descendo do nível 2 para o nível 1, vemos que uma operação Ver(x,C1) é modelada como um R(p), que é a operação de leitura da página p contendo os objetos C1 e C2, enquanto que uma operação de Mod(x,C1) é modelada como um R(p) seguido de um W(p), que é a operação de escrita na página p.

■

Esta abstração visa exclusivamente reduzir os conflitos entre as operações de alto nível, através da exploração da comutatividade das operações abstratas, provendo uma maior concorrência ao Banco de Dados.

No capítulo 2, ao definirmos nosso modelo, enfocamos as propriedades que as transações deveriam obedecer : Atomicidade (tudo ou nada), Consistência, Isolação e Durabilidade. No modelo multinível apenas as operações no nível raiz, ou seja as transações globais, devem obedecer a todas estas propriedades, visto que apenas as transações raízes podem ser observadas externamente. Desta forma, as subtransações só necessitam ser atômicas e isoladas uma vez que a Consistência e Durabilidade devem ser garantidas no nível raiz.

4.2.1 Schedules

Uma vez que definimos uma transação multinível como sendo uma árvore, é notório que um schedule multinível constitui uma floresta.

Definimos uma **camada** em um sistema multinível como sendo um par adjacente de níveis. Portanto, um schedule com n

níveis possui (n-1) camadas.

Formalmente um schedule num nível i é definido por Beeri como :

Definição 4.3 Um schedule num nível i, S_i, é da forma

S_i = (Ei, ConjOp, <, ConjRet, Ef), onde Ei é o estado inicial, ConjOp é o conjunto de operações, < é a ordem de execução das operações, ConjRet é o conjunto de valores de retorno e Ef é o estado final. [Beeri 83]

Definição 4.4 Um schedule multinível é da forma :

S = (S_1, S_2, ... , S_N), onde um S_i, i = 1, 2, ... , N é o schedule para o nível i. [Beeri 83]

4.3 Grafo de Precedência

Na abordagem multinível devemos ter grafos de precedência para cada camada do sistema. O propósito destes grafos é verificar a serializabilidade em cada camada.

Para construção dos respectivos grafos devemos ter em mente três relações de precedências [Cart 89, Martin 87] que auxiliarão na construção dos grafos. Denotaremos estas relações por $<_{iH}$, $<_i$ e $<_{ic}$. Utilizaremos as dependências estáticas (<) e dinâmicas (<<) entre as transações do nível i. A seguir explicaremos as três relações. Utilizaremos a notação T_{ji} para denotar uma transação índice j num nível i. Considere n como sendo o número de níveis.

1. A relação $<_{iH}$ expressa a herança no nível i das precedências do nível i - 1. De modo que :

Se i = 1 então $<_{iH} = <<$

Senão $T_{j1} <_{iH} T_{k1}$ sse Existe $T_{p(i-1)} \in$ filhos(T_{j1})

e Existe $T_{q(i-1)} \in$ filhos(T_{k1}) tais que

$T_{p(i-1)} <_{(i-1)C} T_{q(i-1)}$

2. A relação $<_i$ expressa a serializabilidade das transações no nível i. De modo que :

Para $1 \leq i \leq n$ temos :

$T_{j1} <_i T_{k1}$ sse ($T_{j1} <_{iH} T_{k1}$) ou ($T_{j1} < T_{k1}$)

3. A relação $<_{iC}$ permite suprimir algumas precedências através da comutatividade das operações de Ti. Portanto, temos que :

Para $1 \leq i \leq n$

$T_{j1} <_{iC} T_{k1}$ sse ($T_{j1} <_i T_{k1}$) e Conflita(T_{j1},T_{k1})

Na abordagem multinível, controem-se dois grafos para cada nível : o grafo G_i e o grafo G_{iC}.

O grafo G_i expressa as precedências estáticas existentes no nível i e as precedências herdadas do nível inferior. E da forma $G_i = (N_i, <_i)$, onde N_i é o conjunto de nodos que é formado pelas (sub)transações do nível i (T_i).

O grafo G_{iC} é o grafo construído a partir da relação $<_{iC}$ que explora a comutatividade, reduzindo os arcos. E da forma $G_{iC} = (N_i, <_{iC})$.

Definição 4.5 Um schedule multinível é correto se, para

todos os níveis, os respectivos grafos de precedência G_i, i = 1,2,...,n, são acíclicos.

4.4 Correção da Abordagem Multinível

Para provarmos que um schedule multinível é correto precisamos utilizar três conceitos básicos [Beeri 88, Martin 87]:

1. **Serializabilidade** => é detectada através da construção dos grafos de precedência G_i que utiliza a relação $<_i$;

2. **Comutatividade** => é empregada para reduzir ao máximo o número de conflitos na camada superior. E utilizada na contrução do grafo G_{ic}; e

3. **Redução** => consiste em reduzir de um nível o sistema dirigindo-se no sentido da raiz da árvore.

A seguir apresentamos um algoritmo simples que expressa este mecanismo utilizando os três conceitos apresentados acima, juntamente com as três relações apresentadas na seção 4.3.

```
Algoritmo 4.1
     Const n; /* número de níveis constante */
     i := 1; /* Começa pelo nível  folha */
     Aborto := false;
    Repita
       G1 := Construir_Grafo( ConjOp1, <1);
      Se  G1  é  acíclico  /*  Todas  transações  T1  são
                                        serializáveis */
      entao
         início
            G1c := Construir_Grafo(ConjOp1, <1c);
            i  := i + 1 /* Passa p/ nível superior através  da
                           redução */
         fim

      Senão Aborto := true /* Schedule não-serializável */

    Ate Aborto or i = n;

     Se not Aborto então Schedule serializável

Fim algoritmo
```

O algoritmo mostra-nos como seria o mecanismo de verificação da correção de um schedule multinível.

Resumindo, devemos inicialmente, começando do nível folha, através do grafo G_1, construir o grafo G_{1c}. A partir de G_{1c} construímos o grafo G_2, que contém as precedências estáticas do nível 2 e as precedências herdadas do nível 1; isto é feito para verificar se a respectiva camada é serializável.

Se o grafo G_2 for acíclico, então deve-se construir o grafo G_{2c}, que explora a semântica das operações, através da comutatividade, visando reduzir as precedências existentes.

Por último, aplica-se a propriedade de redução que significa passar para um nível superior; isto é feito através da construção do grafo G_3, que é construído a partir de G_{2c}. Daí todo o processo é repetido. Isto é, primeiro verifica-se a

serializabilidade, depois aplica-se a comutatividade e, por último, aplica-se a redução, até que se chegue ao nível raiz ou se encontre um grafo com ciclo. Neste último caso, o processo pára e o schedule não é correto. Caso se chegue ao nível raiz, então o schedule é correto.

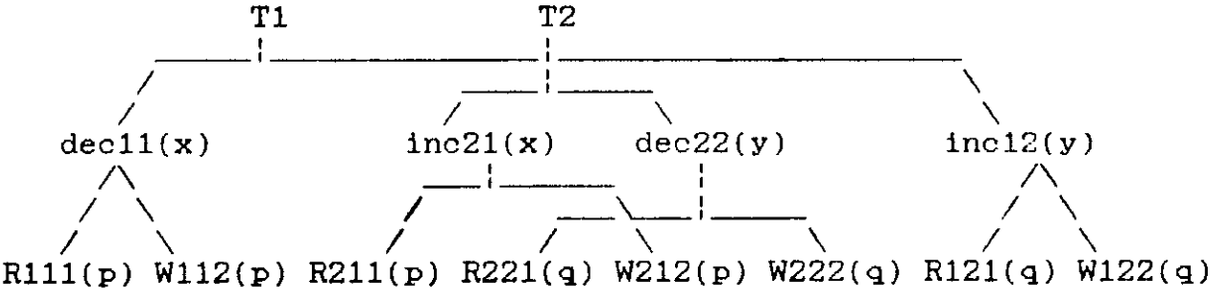
4.5 Exemplos

Nesta seção damos alguns exemplos mostrando como a abordagem multinível funciona na prática. Utilizamos a teoria apresentada para provarmos a correção de um schedule multinível. Utilizamos a topologia de grafos proposta por Cart, por entendermos que se adequa muito bem ao modelo multinível, visto que apresenta, de maneira clara, os vários níveis de abstração existentes [Cart 89]. Entretanto, outras topologias podem ser utilizadas.

Exemplo 4.3 Consideremos o exemplo do tipo abstrato de dados Contador, que possui as seguintes operações tipadas : Inc(x), que incrementa de uma unidade o valor do objeto x, e Dec(x), que decrementa de uma unidade o valor do objeto x.

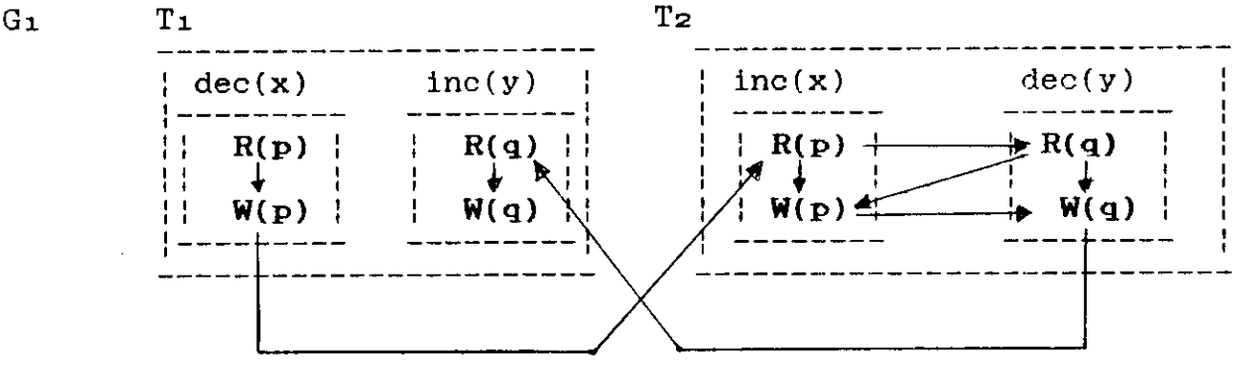
Seja uma execução concorrente de duas transações, T_1 e T_2, que utilizam operações do tipo Contador. Neste caso, temos um schedule multinível com três níveis, onde no nível 3 se encontram as transações, no nível 2 se encontram as operações tipadas Inc(x)/Dec(x) e no nível 1 se encontram as operações indivisíveis de leitura, R(p), e escrita, W(p), que atuam sobre as páginas do Banco de Dados.

Suponhamos que as operações atuam sobre os objetos x e y, que se encontram nas páginas p e q respectivamente. A árvore de execução representando um possível schedule multinível é a seguinte :



Como podemos notar, se considerássemos o schedule em um único nível (sistemas convencionais), o schedule acima não seria serializável, pois contém um ciclo : T_1 -> T_2 -> T_1 sobre as páginas p e q. Mostraremos que este schedule é serializável utilizando a abordagem multinível, através do algoritmo 4.1. Começando do nível folha, devemos construir o grafo G_1, onde os nodos estão em negrito :

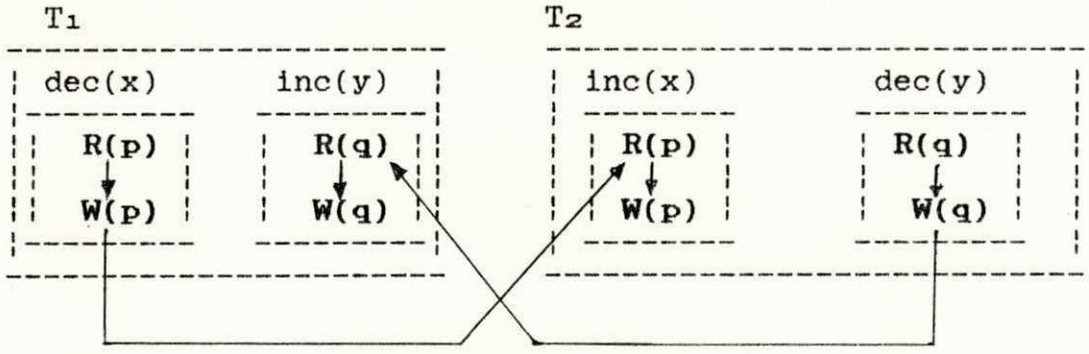
$$G_1 = (N_1, <_1) = (N_1, <<)$$



Como o grafo G_1 é sempre acíclico, então prosseguimos construíndo o grafo G_{1c} que explora a comutatividade reduzindo o

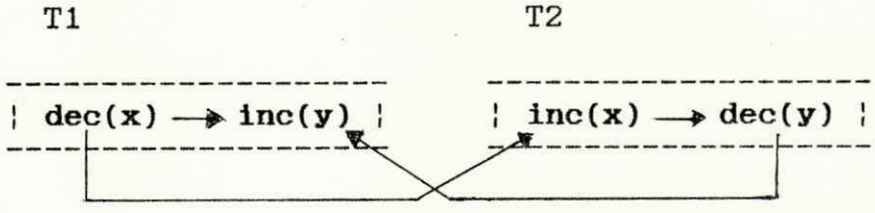
número de arcos. Para isso, podemos suprimir dois arcos na transação T_2 entre as operações inc21(x) e dec22(y), visto que são duas leituras e duas escritas em objetos distintos que portanto, comutam. Daí temos:

$G_{1c} = (N_1, <_{1c})$, onde N_1 é o conjunto de operações do nível folha. Os nodos estão representados no grafo em negrito.



A partir do grafo G_{1c}, construímos o grafo G_2 para verificarmos a serializabilidade no nível 2.

$G_2 = (N_2, <_2)$, onde N_2 é o conjunto de operações do nível 2, representada no grafo em negrito.



Como o grafo G_2 é acíclico, então, a partir dele, construímos o grafo G_{2c} que explora a comutatividade.

$G_{2c} = (N_2, <_{2c})$

Podemos ver que, como as operações de incremento e decremento são comutativas, comutatividade geral, temos que os arcos existentes entre estas operações são suprimidos.

Por último, a partir do grafo G_{2c} construímos o grafo G_3 :

$G_3 = (N_3, <_3)$, onde N_3 são as transações globais, que formam o conjunto dos nodos de G_3. Temos que:

G_3 :

T_1 T_2

Como o grafo G_3 é acíclico e já alcançamos o nível raiz então podemos afirmar que o schedule multinível é serializável. ∎

Exemplo 4.4 [Garcia-Molina 83] Consideremos agora um sistema de reserva de passagens aéreas cujas operações são Reservar(X), que faz uma reserva de uma passagem para o vôo X, e Cancelar(X) que cancela uma reserva feita previamente para o vôo X. O sistema funciona da seguinte maneira :

Existem dois objetos para cada vôo, NX e TX, que indicam, respectivamente, o número de passagens reservadas e o tipo da aeronave. Inicialmente, o tipo da aeronave recebe o valor `pequeno´ e se o número de reservas ultrapassar a 100, então o valor de TX é mudado para `grande´, não sendo mais modificado. Um pedido de reserva pode ser negado caso não haja mais vagas na aeronave (isto será expressado pelo parâmetro de retorno que conterá o valor "false", caso contrário o parâmetro é "true" e a

reserva é efetuada). Uma operação de reserva é implementada por três operações:

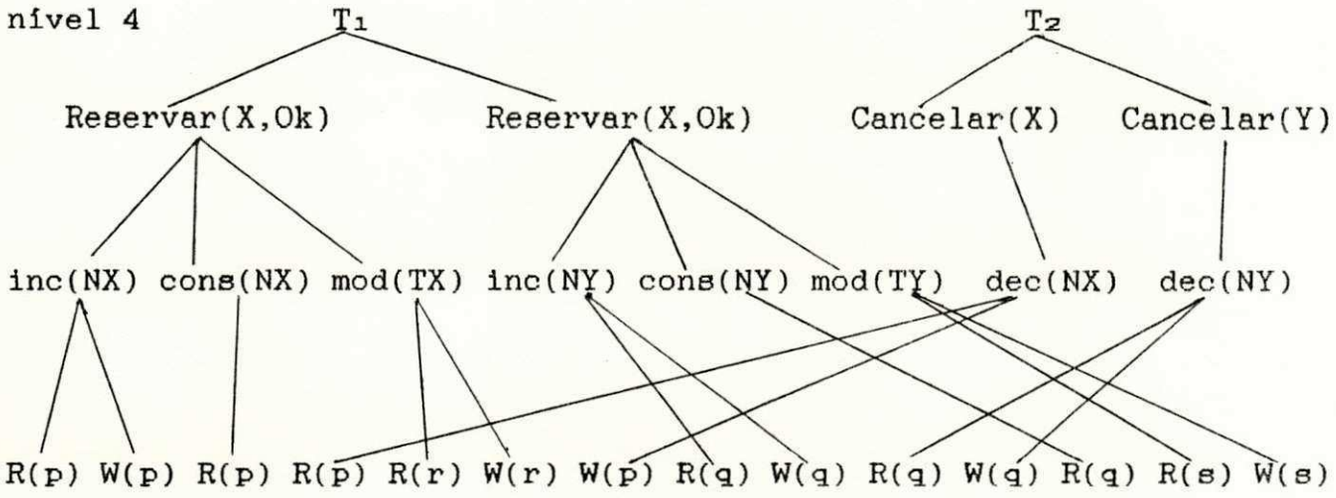
1. **inc(NX)**, que incrementa de um o objeto NX

2. **cons(NX)**, que efetua uma consulta no objeto NX para saber o número de passagens reservadas; e

3. **mod(TX)**, que modifica o tipo da aeronave de 'pequeno' para 'grande' segundo o critério mencionado anteriormente.

Por outro lado, uma operação de cancelamento de reserva é implementada por uma operação, dec(NX), que decrementa de um o valor do objeto NX.

Seja uma execução concorrente de duas transações, T_1 e T_2, cujas respectivas árvores são dadas a seguir. Os números abaixo das folhas significam a ordem dinâmica de execução das operações primitivas. A transação T_1 faz duas reservas nos vôos X e Y, enquanto que a transação T_2 faz dois cancelamentos de reservas feitas nos vôos X e Y. A página p armazena o objeto NX, a página q armazena o objeto NY, a página r armazena o objeto TX e a página s armazena o objeto TY.



A tabela de comutatividade para operações de reserva e cancelamento é a seguinte:

		operação executada	
	Vôo	Reservar(Ok1)	Cancelar
operação a ser executada	Reservar(Ok2)	Ok1 = Ok2	não
	Cancelar	Ok1 = true	sim

Legenda: Ok1 e Ok2 são parâmetros de retorno. Duas operações comutam se a condição é verdadeira.

A tabela de comutatividade das operações sobre os objetos NX e NY é dada a seguir:

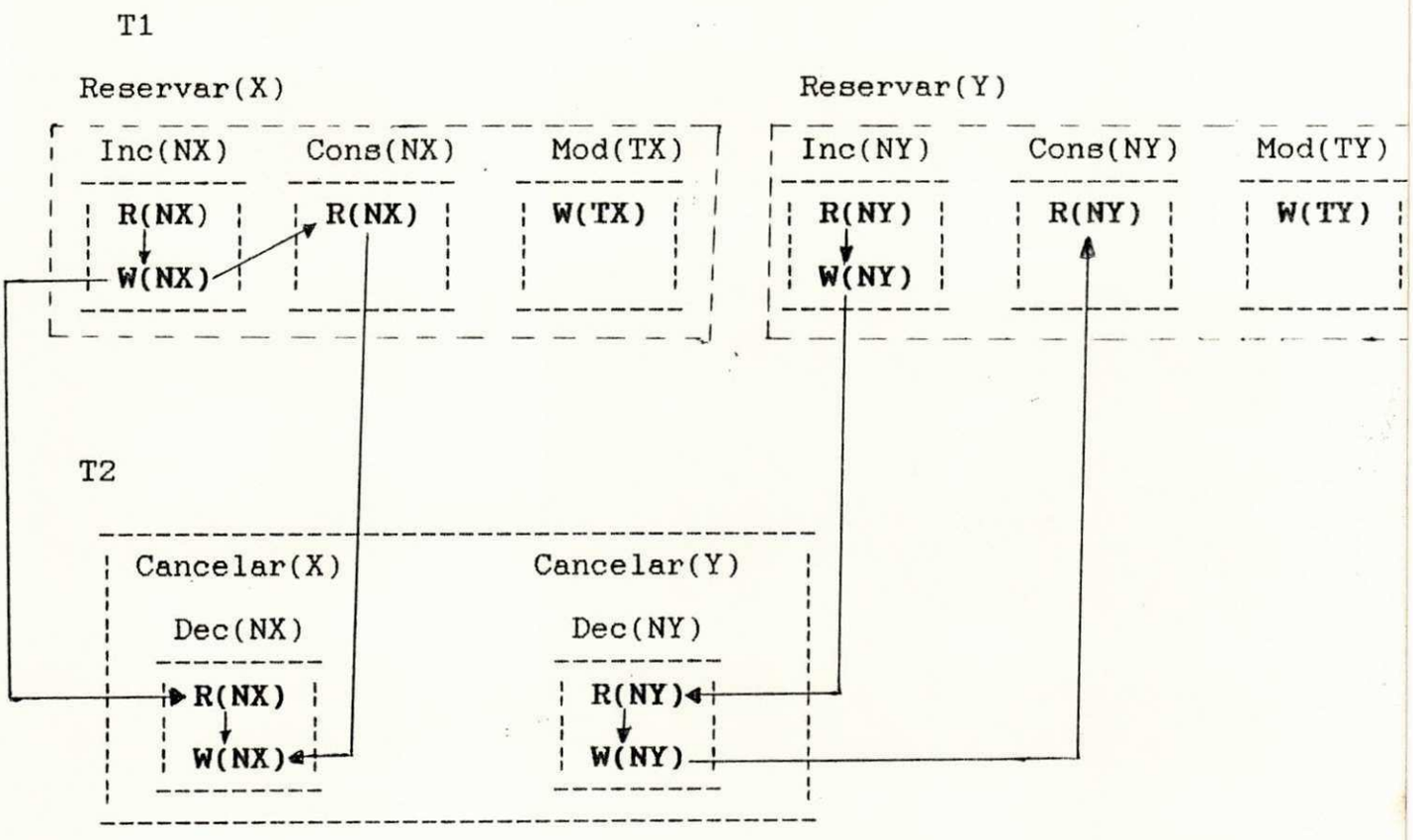
		operação executada		
	NX	inc	dec	cons
operação a ser executada	inc	NX < Max	sim	não
	dec	sim	sim	não
	cons	não	não	sim

onde Max é o teto máximo representando o número máximo de

reservas que podem ser efetuadas.

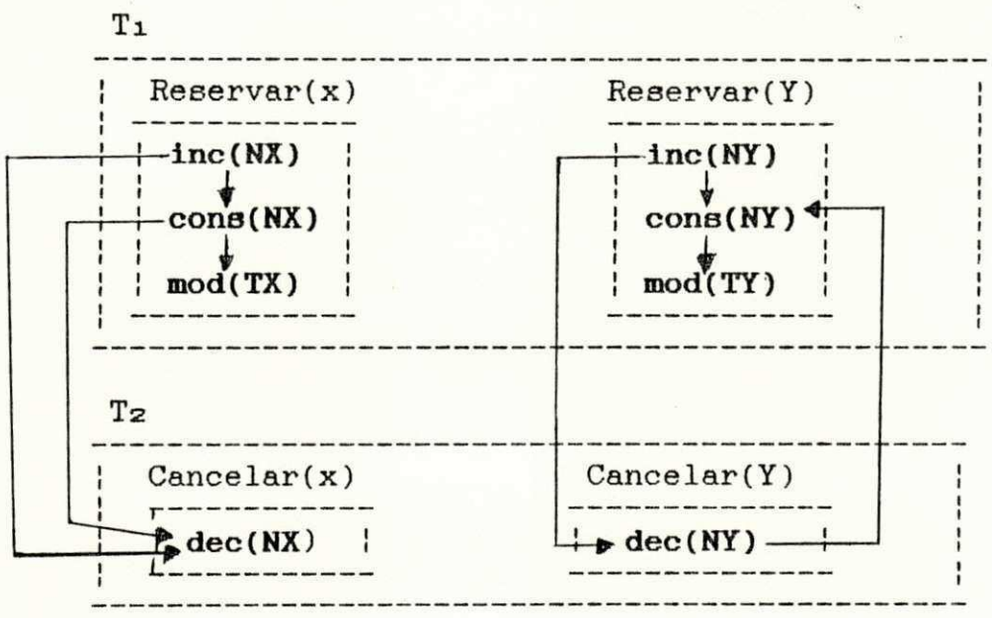
Como o grafo G_1 é sempre acíclico, começaremos construíndo o grafo G_{1c}, que é obtido a partir de G_1 suprimindo os arcos desnecessários. Os nodos estão em negrito.

G_{1c} :



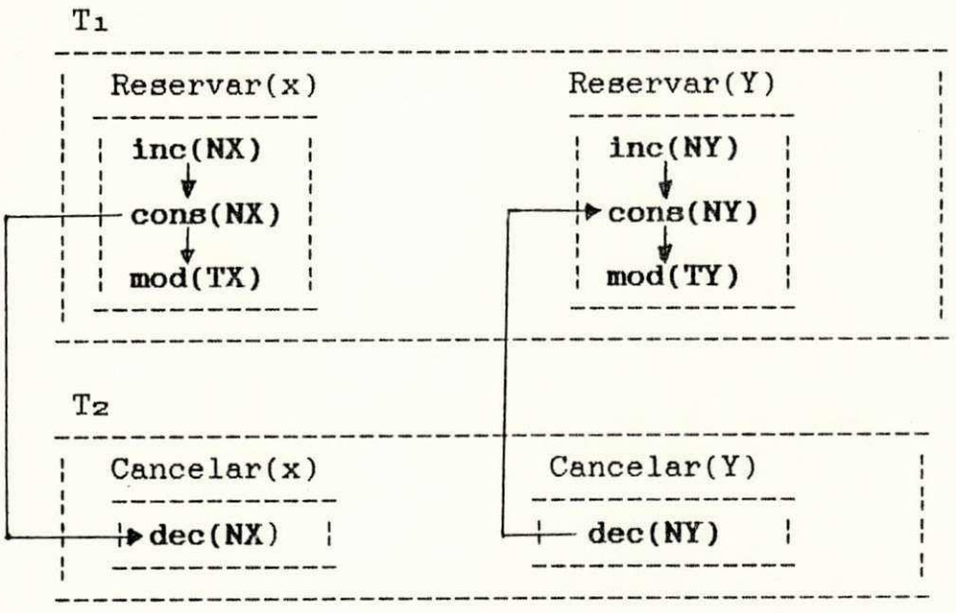
Daí, construímos o grafo G_2 a partir de G_{1c}. G_2 contém as precedências herdadas do nível 1 e as precedências estáticas do nível 2. Os nodos estão em negrito.

G_2 :

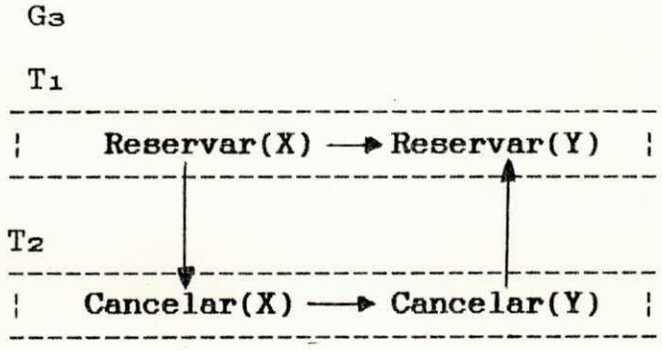


Podemos ver que o schedule do nível 2 é serializável uma vez que o grafo G_2 é acíclico. Daí devemos construir o grafo G_{2c} que explora a comutatividade das operações. Portanto, como as operações de incremento e decremento são comutativas, conforme tabela anterior, as precedências existentes entre estas operações são omitidas. Daí temos:

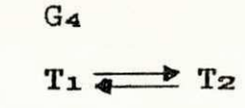
G_{2c} :



A partir do grafo G_{2c} construímos o grafo do schedule do nível 3, G_3 :



Como podemos ver, o schedule do nível 3 é correto, pois o grafo G_3 é acíclico. Como as precedências existentes não podem ser suprimidas, pois as operações não são comutativas (ver tabela de operações de reserva/cancelamento), então o grafo G_{3c} é o mesmo que o grafo G_3. Daí devemos partir para a construção do grafo G_4 que verifica se o schedule do nível topo é correto. Portanto,



O grafo G_4 contém um ciclo e, portanto, o schedule multinível não é serializável.

Nota : Garcia-Molina considera o schedule acima como correto, pois é usado o conceito de coerência semântica, que diz que as transações T_1 e T_2 são compatíveis. Deste modo, este ciclo pode ser ignorado e o schedule é considerado correto. [Garcia-Molina 83]

4.6 Conclusão

Apresentamos neste capítulo a abordagem multinível. Definimos o modelo de transações multiníveis, apresentamos o critério de correção adotado e finalizamos com alguns exemplos.

Esta abordagem multinível explora o paralelismo interno das transações, incorpora a abstração de dados, explora a semântica das operações e possui uma recuperação muito eficiente, pois uma subtransação é vista como uma unidade de recuperação e caso haja uma falha numa subtransação, a transação como um todo não irá abortar, podendo reexecutar a subtransação que falhou sem comprometer as demais subtransações em execução.

Os sistemas multiníveis são indicados para os Bancos de Dados Orientados a Objeto, sistemas distribuídos e aplicações não-convencionais.

Esta abordagem pode ser implementada pelos diversos mecanismos propostos no capítulo 2, quais sejam : Bloqueio em duas fases, Timestamp, Otimista, etc. No próximo capítulo, apresentamos um protocolo baseado no mecanismo de bloqueio em duas fases para as transações multiníveis.

5. PROTOCOLO DE BLOQUEIO EM DUAS FASES PARA TRANSAÇOES MULTINIVEIS - B2FM

5.1 Introdução

Assim como na abordagem semântica, descrita no capítulo 3, uma arquitetura multinível pode ser implementada utilizando os diversos mecanismos expostos no Capítulo 2, quais sejam : Bloqueio, Timestamp, Otimista, Detecção de Ciclos no Grafo, etc.

Além disso, Weikum diz que pode-se ter, numa implementação de um sistema multinível, diferentes mecanismos para as camadas. Desta forma, existe um scheduler para cada camada e este é implementado com o mecanismo que melhor se adapte à mesma. [Weikum 87].

Neste capítulo, iremos introduzir nosso protocolo de Bloqueio em Duas Fases para Transações Multiníveis - B2FM. Apesar de ser uma linha de pesquisa ainda pouco explorada, já existem trabalhos que utilizam alguns destes mecanismos para abordagem multinível. Podemos citar, por exemplo, Moss, que propôs um mecanismo de bloqueio em duas fases para transações multiníveis e Cart, que propôs um protocolo otimista para transações multiníveis. [Moss 85, Cart 90a].

Entretanto, o protocolo B2FM diverge daquele proposto por Moss visto que, neste último, a sincronização é feita sobre operações de Read e Write, o que o torna pobre semanticamente. Enquanto que, no primeiro, utiliza-se o modelo estendido

apresentado no Capítulo 3, suportando abstração de dados, utilizando a semântica dos tipos abstratos e das operações tipadas, e aplicando a comutatividade. Utilizaremos bloqueios de tipo específico abordados no capítulo 3. Assim, o protocolo B2FM provê um maior paralelismo do que aquele proposto por Moss [Moss 85].

Ademais, Weikum e Badrinath propuseram mecanismos de bloqueio em duas fases para transações multiníveis, porém algumas hipóteses que estabelecemos em nosso protocolo são menos restritivas do que estes últimos. Comentaremos os ganhos e perdas de tais hipóteses com relação aos demais protocolos propostos. [Weikum 87, Badrinath 90].

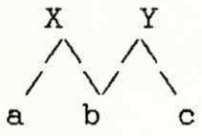
Escolhemos a técnica de Bloqueio em Duas Fases [Eswaran 76], por ser de fácil implementação, garantir uma performance razoável e por ser propício à tolerância às falhas [Turc 90].

As demais seções deste capítulo estão assim dispostas: a seção 5.2 descreve as hipóteses assumidas para o modelo, a seção 5.3 apresenta o protocolo B2FM, a seção 5.4 expõe alguns exemplos de como o protocolo funciona e a seção 5.5 apresenta uma breve conclusão deste capítulo.

5.2 Hipóteses do modelo

H1. **Os objetos são independentes** => Isto significa dizer que as operações sobre objetos distintos sempre comutam. Com isso, não se faz necessária a existência de tabelas de

comutatividade contendo operações sobre objetos diferentes, pois não existe objetos da forma:

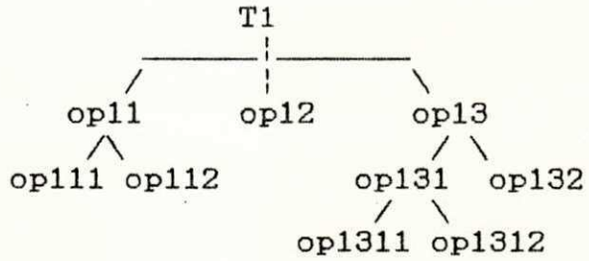


em que b é um objeto comum a X e a Y.

Cart aborda a importância da independência dos objetos, permitindo que existam objetos da forma acima desde que as operações sobre o objeto comum sejam comutativas. [Cart90b]

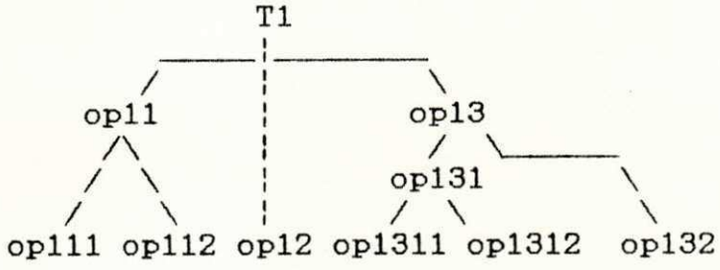
H2. Num nível i pode-se ter objetos de vários tipos, porém um determinado tipo aparece apenas num único nível. Esta hipótese difere nosso modelo daquele proposto por Rakow, onde uma (sub)transação pode invocar uma outra e ambas têm acesso a um mesmo objeto [Rakow 90]. Portanto, uma operação num nível i não pode invocar operações do mesmo nível ou num nível ancestral.

H3.**A árvore de decomposição pode ser desequilibrada** => Isto permite que tenhamos níveis de abstração incompletos, divergindo nosso modelo daqueles propostos em [Weikum 87, Moss 85, Badrinath 90], pois nestes últimos, assume-se a hipótese de que a árvore é equilibrada. Com isto, em nosso modelo, esta hipótese permite que tenhamos maior flexibilidade para suportar o modelo orientado a objetos, onde o número de níveis de abstração de diferentes objetos não é o mesmo. Portanto, em nosso modelo podemos ter uma árvore com a seguinte estrutura :



onde as operações folhas, sobre páginas, são : op111, op112, op12, op1311, op1312 e op132.

Neste caso, as operações devem ficar no nível de abstração ao qual pertença. Então a árvore acima deveria ser escrita da seguinte forma :



Podemos verificar que a operação op12 não contém todos os níveis de abstração.

A correção deste protocolo baseia-se na herança de bloqueios que descreveremos mais adiante e na técnica de bloqueios em duas fases, provendo para isso, schedules serializáveis.

A vantagem desta técnica é a aproximação com o modelo de dados sem incorrer num grande esforço do scheduler. Por outro lado, a desvantagem é que no nosso protocolo, os bloqueios de uma operação com níveis incompletos serão retidos, às vezes desnecessariamente, até o final do pai (que pode ser avô, etc. de

uma outra operação do mesmo nível que possui todos os níveis). A inserção de operações virtuais, cuja finalidade é prover um equilíbrio à árvore, resolveria este problema. Porém, esta solução é complexa, pois ao inserirmos operações virtuais, devemos informar ao scheduler a comutatividade entre estas operações virtuais e as demais operações do mesmo nível. Esta abordagem, utilizando a inserção de operações virtuais ainda não foi estudada e propomos como futuros trabalhos que poderão dar continuidade a esta dissertação.

5.3 O Protocolo de Bloqueio em Duas Fases Multinível - B2FM

Apresentaremos o protocolo através de um conjunto de regras que devem ser rigorosamente obedecidas para que tenhamos uma execução correta.

Protocolo B2FM

Regra 1 : Toda (sub)transação antes de ter acesso a um objeto deve bloqueá-lo. Este bloqueio é de tipo especifico : em cada nível os bloqueios são do tipo das operações daquele nível.

Regra 2 : Caso o objeto já esteja bloqueado, um novo pedido de bloqueio só deve ser autorizado se este for compatível com os bloqueios já assegurados ou se todos os bloqueios incompatíveis, já assegurados, estão possuídos por nodos ancestrais. A compatibilidade é averiguada na tabela de comutatividade que existe para cada objeto.

Regra 3: Caso o pedido de bloqueio não seja aceito (segundo a regra 2), ele é colocado numa fila de espera e fica aguardando que as (sub)transações que detêm bloqueios em modo incompatível, liberem-nos.

Regra 4 : Um bloqueio num nível i deve ser liberado somente ao final da (sub)transação pai, isto é, no nível i+1, juntamente com os bloqueios de todos os filhos da (sub)transação pai. Esta regra expõe a **herança de bloqueios** que existe de um determinado nível para seu ancestral; em outras palavras, quando uma subtransação é executada por completo, obtendo um pseudocommit (cujo objetivo é indicar ao pai que a subtransação não abortou, sendo, portanto, diferente do commit de uma transação global), todos os seus bloqueios são herdados pela (sub)transação pai.

Regra 5 : Após liberar ou passar para o pai quaisquer dos objetos bloqueados no nível i, a (sub)transação não pode solicitar mais bloqueios.

Regra 6 : Quando uma transação global termina sua execução, obtendo um Commit, todos os bloqueios assegurados até então devem ser liberados.

Regra 7 : Os bloqueios devem ser adquiridos no sentido top-down, isto é, primeiro o bloqueio é adquirido na transação raiz e segue-se adquirindo bloqueios de tipo específico nos descendentes, até atingir o nível folha. Os bloqueios devem ser

liberados no sentido inverso, isto é, bottom-up.

Fim B2FM

As regras 1, 2 e 3 são comuns a qualquer mecanismo de controle de concorrência baseado em bloqueios.

A regra 4 mostra a herança dos bloqueios dos filhos para o pai. Isto ocorre para que se tenha a propriedade de isolação das (sub)transações. Isto também mostra a maior eficiência deste método em relação ao protocolo de Bloqueio em Duas Fases convencional. Neste último, os bloqueios sobre páginas são retidos até o fim da transação, enquanto que no primeiro eles são liberados à medida em que as subtransações vão terminando. Estes bloqueios são conhecidos como Short Locks [Beeri 88]. A preocupação reside em reduzir a duração destes bloqueios para prover uma maior concorrência.

A regra 5 diz respeito às duas fases : fase de crescimento e fase de liberação de um bloqueio em duas fases.

A regra 6 mostra a liberação dos objetos que ficam retidos até o final da transação global.

Por último, a regra 7 mostra a maneira de aquisição/liberação de bloqueios, evitando a ocorrência de deadlocks globais.

Podemos ver que se o número de camadas é 1, isto é, um sistema convencional, este protocolo funciona como o 2PL estrito dos sistemas convencionais [cf. capítulo 2].

Como todo protocolo baseado em bloqueios, o B2FM é susceptível a deadlock. Porém, ao contrário do protocolo proposto por Moss em [Moss 85], em que pode ocorrer um deadlock entre várias camadas (conhecido como deadlock global), o B2FM é imune a este tipo de deadlock, visto que não existem objetos comuns a duas ou mais camadas (ver hipótese H2), como também devido à regra de herança de bloqueios descrita acima.

Portanto, o que é possível de acontecer no B2FM é um deadlock dentro de uma camada, que também é possível em [Moss 85] e no 2PL convencional, de forma que utilizam-se as técnicas válidas nos sistemas convencionais para detecção/resolução de deadlocks, pois o procedimento é idêntico [Bernstein 87].

A seguir, apresentamos o procedimento de pedido de bloqueio, denominado Lock e o procedimento de liberação de bloqueios, denominado Unlock.

Para cada objeto existe uma lista de bloqueios assegurados (Bloqueios) e uma lista de pedidos de bloqueios esperando (Espera).

Cada elemento da lista Bloqueios contém um campo que indica, se o bloqueio já foi herdado pelo pai (Herança = "Sim").

A função EhDescendente(X,Y) retorna true se X é descendente de Y e retorna false, caso contrário.

O procedimento RetirarEspera retira um pedido de bloqueio da lista de espera.

```
Procedimento Lock( Objeto, Tipo );

/ As variáveis globais são para cada objeto /

inicio
 se NumBloqueios > 0
   então
     inicio
      Compatível := true;
      Tabela := Tabela.Objeto; / Leitura da tabela do objeto /
      i := 1;
      enquanto ( i ≤ NumBloqueios ) e Compatível faça
       inicio
          Compatível := (Tabela [Bloqueios[i],Tipo] =  "Sim"
                          ou (Bloqueios[i].Herança = "Sim"
                    e EhDescendente(Tipo,Pai(Bloqueios[i]))));
          i := i + 1
       fim
      se Compatível
           então inicio
                    NumBloqueios := NumBloqueios + 1;
                    Bloqueios [NumBloqueios] := Tipo;
                    <<    EXECUTAR    OPERAÇÃO     >>
                 fim
      senão InserirListaEspera( Tipo )
     fim
    senão inicio
            NumBloqueios := 1;
            Bloqueios[1] := Tipo;
          fim
fim.
```

```
Procedimento Unlock( Objeto, Tipo);

início

   para i := 1 até NumFilhos(Tipo) faça
        LiberarBloqueios(Filho(Tipo)); / Retirar da lista de
                                bloqueios correspondente ao tipo /

   se (nivel  ≠ Raiz)
     então Bloqueios[Tipo].Herança := "Sim";

   para i := 1 até TamEspera faça
     início
       Compatível := true;
       j := 1;
       enquanto (j ≤ NumBloqueios) e Compatível faça
         início
           Compatível := (Tabela[Bloqueios[j],Espera[i]] = "Sim"
                          ou (Bloqueios[i].Herança = "Sim" e
                          EhDescendente(Tipo,Pai(Bloqueios[i]))));
           j := j + 1
         fim
       se Compatível
          então início
                  NumBloqueios := Numbloqueios + 1;
                  Bloqueios[NumBloqueios] := Espera[i];
                  RetirarEspera(i);
                  << EXECUTAR OPERAÇAO i >>
                fim
     fim

fim.
```

O procedimento Unlock é mais complexo do que o Lock, pois necessita de uma política justa de liberação de operações que estão esperando para que não favoreça apenas algumas operações em detrimento de outras. Por isso, é dependente da aplicação.

5.4 Exemplos

Daremos nesta seção alguns exemplos de como o protocolo B2FM funciona na prática. A notação que utilizaremos é a seguinte: utilizaremos uma numeração ao lado das operações para expressar a ordem de execução das operações. Além disso, exporemos as respectivas tabelas de comutatividade de cada exemplo, dizendo o tipo de comutatividade utilizada. O primeiro exemplo que apresentaremos contém uma árvore equilibrada enquanto que o segundo contém uma árvore irregular.

Exemplo 5.1 Consideremos um schedule com operações do tipo Contador (sem teto).

A tabela de comutatividade geral para objetos do tipo contador é :

		Bloqueio Assegurado	
	Objeto	Inc	Dec
Bloqueio Requerido	Inc	sim	sim
	Dec	sim	sim

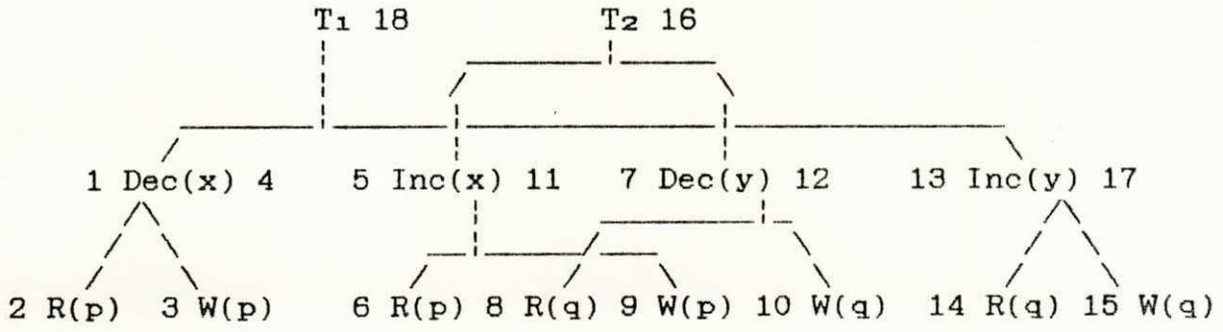
A tabela de comutatividade parametrizada para as operações sobre as páginas é :

		Bloqueio Assegurado	
	Objeto	R(v1)	W (v1)
Bloqueio Requerido	R(v2)	sim	não
	W(v2)	v1 = v2	v1 = v2

Legenda: v1 e v2 são parâmetros de retorno/chamada. Duas

operações comutam se a condição é satisfeita.

Seja a seguinte execução concorrente de T_1 e T_2



Em cada ordem de execução daremos os pedidos de bloqueio/liberação. Onde $Lock_1$($Unlock_1$) representa um pedido de bloqueio (liberação) pela transação T_1.

1: $Lock_1$(x,Dec), T_1 obtém um bloqueio de tipo Dec sobre o objeto x.

2: $Lock_1$(p,R), Dec(x) obtém um bloqueio de leitura sobre a página p.

3: $Lock_1$(p,W), Dec(x) obtém um bloqueio de gravação sobre a página p.

4: $Unlock_1$(p,R); $Unlock_1$(p,W), $Unlock_1$(x,Dec) liberação dos bloqueios de leitura e gravação e T_1 herda bloqueio Dec(x).

5: $Lock_2$(x,Inc), T_2 obtém um bloqueio de tipo Inc sobre o objeto x. (Compatível com o já assegurado)

6: $Lock_2$(p,R), Inc(x) obtém um bloqueio de leitura sobre a página p.

7: $Lock_2$(y,Dec), T_2 obtém um bloqueio de tipo Dec sobre o objeto y.

8: $Lock_2$(q,R), Dec obtém um bloqueio de leitura sobre a página q.

9: $Lock_2(p,W)$, Inc(x) obtém um bloqueio de gravação sobre a página p.

10: $Lock_2(q,W)$, Dec(y) obtém um bloqueio de gravação sobre a página q.

11: $Unlock_2(p,R)$; $Unlock_2(p,W)$; $Unlock_2(x,Inc)$, liberação dos bloqueios de leitura e gravação sobre a página p e T_2 herda o bloqueio Inc(x).

12: $Unlock_2(q,R)$; $Unlock_2(q,W)$; $Unlock_2(y,Dec)$, liberação dos bloqueios de leitura e gravação sobre a página q e T_2 herda o bloqueio Dec(y).

13: $Lock_1(y,Inc)$, T_1 obtém um bloqueio de tipo Inc sobre o objeto y.

14: $Lock_1(q,R)$, Inc(y) obtém um bloqueio de leitura sobre a página q.

15: $Lock_1(q,W)$, Inc(y) obtém um bloqueio de gravação sobre a página q.

16: $Unlock_2(x,Inc)$; $Unlock_2(y,Dec)$, T_2 libera os bloqueios de de tipo Inc e Dec sobre os objetos x e y, respectivamente, que tinha herdado anteriormente.

17: $Unlock_1(q,R)$; $Unlock_1(q,W)$; $Unlock_1(y,Inc)$, liberação dos bloqueios de leitura e gravação sobre a página q e T_1 herda bloqueio Inc(y).

18: $Unlock_1(x,Dec)$; $Unlock_1(y,Inc)$, T_1 libera os bloqueios de tipo Dec e Inc sobre os objetos x e y, respectivamente, que tinha herdado anteriormente.

Podemos notar que como as operações de incremento e decremento são comutativas então uma (sub)transação não espera

por nenhuma outra.

∎

Exemplo 5.2 Consideremos um tipo conjunto C, com operações de inserção de um elemento x, denotado por Ins(C,x), de remoção de um elemento x, Rem(C,x) e uma operação de consulta a um elemento x, Cons(C,x). Seja Num o número de elementos do conjunto. As tabelas abaixo mostram a comutatividade entre os objetos do tipo conjunto.

A tabela de comutatividade parametrizada para operações do tipo conjunto é a seguinte :

		Bloqueio Assegurado		
	Objeto	Ins(x)	Del(x)	Cons(x)
Bloqueio Pedido	Ins(y)	sim	x <> y	x <> y
	Del(y)	x <> y	sim	x <> y
	Cons(y)	x <> y	x <> y	sim

Legenda : Parâmetros x e y. Duas operações comutam se a condição for verdadeira ou se contiver uma entrada sim.

Tabela de comutatividade geral para o objeto do tipo contador:

		Bloqueio Assegurado	
	Objeto	Inc	Dec
Bloqueio Pedido	Inc	sim	sim
	Dec	sim	sim

A tabela de comutatividade parametrizada para as operações sobre as páginas é :

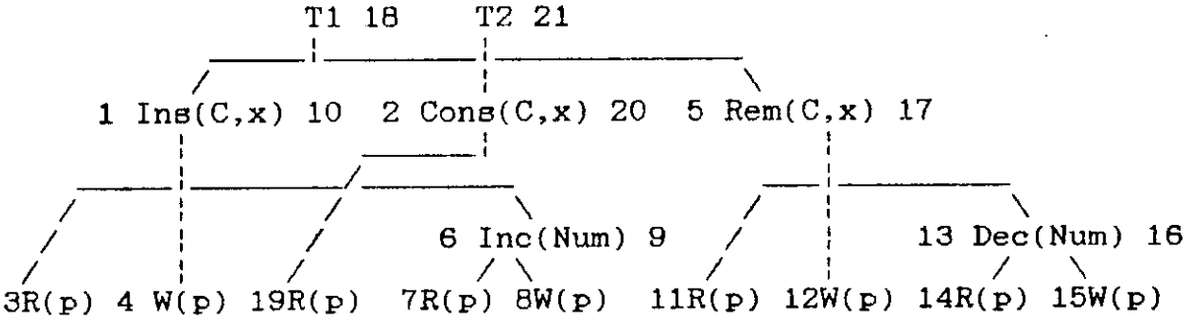
		Bloqueio Assegurado	
	Objeto	R(v1)	W (v1)
Bloqueio Pedido	R(v2)	sim	não
	W(v2)	v1 = v2	v1 = v2

Legenda: v1 e v2 são parâmetros de retorno/chamada. Duas operações comutam se a condição é satisfeita.

Sejam T_1, que insere um objeto x no conjunto C e depois retira este objeto de C; e T_2, que verifica se o objeto x está no conjunto C, duas transações.

As estruturas das operações estão descritas na árvore de execução a seguir.

Seja a seguinte execução concorrente de T_1 e T_2, podendo ser obtida pelo B2FM. A numeração ao lado das operações representa a ordem de excução e o objeto x se encontra na página **p**.



Para cada número acima, que representa a ordem de execução, daremos os pedidos de bloqueio/liberação; quando não for óbvio, explicaremos o que significa cada bloqueio.

1: Lock(C,Ins)

2: Lock(C,Cons) -> A operação é bloqueada, pois o tipo de bloqueio Cons não comuta com o bloqueio já assegurado Ins.

3: Lock(p,R)

4: Lock(p,W)

5: Lock(C,Rem) -> A operação é bloqueada, pois não comuta com o bloqueio já assegurado por Ins.

6: Lock(Num,Inc)

7: Lock(p,R) -> Este bloqueio é assegurado porque os bloqueios feitos em p, por R(p) e W(p), foram herdados pela operação Ins que é ancestral desta operação requerente.

8: Lock(p,W) -> este bloqueio é assegurado pela mesma razão exposta no bloqueio 7.

9: Unlock(Num,Inc) -> liberação dos bloqueios herdados : R e W e passagem do bloqueio Inc para o pai.

10: Unlock(C,Ins) -> liberação dos bloqueios herdados : R, W e Inc; e T_1 herda o bloqueio Ins.

11: Lock(p,R) -> Este bloqueio é assegurado porque a operação

Rem já foi desbloqueada, pois o bloqueio Ins já foi liberado para o pai T1 e a página p está livre.

12: Lock(p,W) -> Mesma razão anterior.

13: Lock(Num,Dec)

14: Lock(p,R) -> Este bloqueio é assegurado porque a operação Rem herdou os bloqueios sobre a página p, R(p) e W(p), e é ancestral da operação requerente.

15: Lock(p,W) -> Mesma razão anterior.

16: Unlock(Num,Dec) -> liberação dos bloqueios herdados R e W; e passagem do bloqueio Dec para o pai.

17: Unlock(C,Rem), liberação dos bloqueios herdados: R , W e Dec; e passagem do bloqueio Rem para o pai T_2.

18: Unlock(C,Ins); Unlock(C,Rem), liberação dos bloqueios de Ins e Rem pela transação T_1.

19: Lock(p,R) -> O bloqueio é assegurado pois a operação Cons que estava bloqueada no passo 2 foi liberada pelo passo 18.

20: Unlock(C,Cons) -> liberação do bloqueio herdado R e T_2 herda o bloqueio do tipo Cons.

21: Unlock(C,Cons)

Podemos notar que o exemplo acima utiliza as hipóteses mencionadas anteriormente e que o schedule é a execução serial T1;T2.
∎

5.5 Conclusão

Neste capítulo apresentamos, o protocolo de Bloqueio em Duas Fases Multinível - B2FM, que difere daqueles apresentados por Moss, Weikum e Badrinath e apresenta um acréscimo considerável no paralelismo entre as transações, quando comparado ao 2PL convencional. [Moss 85, Weikum 87, Badrinath 90]

O protocolo baseado em bloqueios tem sido largamente difundido pela sua facilidade de implementação e pela razoável performance em relação aos outros mecanismos. Entretanto, trata-se de um protocolo pessimista, susceptível a deadlocks e existem muitos schedules corretos que não podem ser obtidos pelo protocolo B2FM. Mas este é o preço que tem que ser pago para poder usá-lo e, portanto, a aplicação é quem vai dizer se é viável ou não o utilizar. Até então, os protocolos baseados em bloqueios são os mais utilizados nos sistemas de Bancos de Dados Convencionais e Não-Convencionais.

Capítulo 6: CONCLUSÃO

Apresentamos nesta dissertação um estudo sobre o controle de concorrência em Sistemas Transacionais, fazendo um comparativo entre a abordagem Convencional com a Não-Convencional.

Quando se trata de aplicações convencionais, os métodos clássicos, apresentados no Capítulo 2, atendem, de forma muito eficiente e segura, às suas necessidades. Porém, quando partimos para as aplicações não-convencionais, a utilização de novas técnicas, entre as quais a abordagem multinível, faz-se necessária.

Mostramos ao longo deste trabalho que, a utilização da semântica das operações, através da abstração de dados e da comutatividade, resulta num ganho enorme no controle da concorrência [Badrinath 90].

Ao utilizarmos a abordagem multinível, estamos incorporando o conceito dos tipos abstratos de dados, como também estamos permitindo que haja paralelismo entre as operações tipadas, tidas na abordagem semântica como indivisíveis.

Ademais, a utilização de uma arquitetura multinível vem de encontro às exigências das novas aplicações, explora o paralelismo interno das transações e provê um mecanismo de recuperação eficiente e integrado ao controle de concorrência. Assim, o estudo da abordagem multinivel visa melhorar cada vez mais a performance e a confiabilidade dos sistemas transacionais.

O protocolo de bloqueio em duas fases é apenas uma das maneiras de implementar uma arquitetura multinível. O estudo das hipóteses fizeram com que nosso protocolo fosse menos restritivo do que os protocolos já propostos anteriormente. Entretanto, existe ainda muito trabalho a ser explorado nesta área. Citaremos, a seguir, algumas sugestões para futuros trabalhos.

1. O estudo dos impactos da tolerância às falhas neste protocolo de controle de concorrência.

2. A introdução de operações virtuais em árvores desequilibradas, para que haja um aumento na performance do protocolo, por exemplo, reduzindo a duração de bloqueios.

3. Extensão do modelo, permitindo que objetos possam ser referenciados em mais de um nível [Rakow 90], o que aproximaria ainda mais o modelo de transações do poder de expressão semântica provido pela abordagem orientada a objetos.

4. Estudo da viabilidade da utilização de objetos dependentes (cf. Hipótese 1).

5. Estudo de novos critérios de correção que não a serializabilidade, tais como coerência semântica [Garcia-Molina 83], modularidade [Sha 88] e incorporação destas idéias à abordagem multinível.

6. Utilização de técnicas mistas de controle de concorrência na abordagem multinível, de modo que cada camada tenha o scheduler que melhor se adapte à mesma.

7. REFERENCIAS BIBLIOGRAFICAS

[Atkinson 89] ATKINSON, M. et alii. The Object-Oriented Database System Manifesto. Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japão, dez/1989.

[Badrinath 90] BADRINATH, B. R. e RAMAMRITHAM, K. Performance Evaluation of Semantics-Based Multilevel Concurrency Control Protocols. Proceedings of the 1990 SIGMOD Conference.

[Beeri 83] BEERI C. et alii. Concurrency Control Theory for Nested Transactions. Proc. 2nd ACM Symposium on Principles of Distributed Computing, Montreal, Canadá, p. 45, ago/1983.

[Beeri 88] BEERI, C., SCHECK, H. e WEIKUM, G. Multi-Level Transaction Management, Theoretical Art or Practical Need? Proceedings of the International Conference on Extending Database Technology, Venice, Itália, mar/1988.

[Bernstein 81] BERNSTEIN, P. e GOODMAN, N. Concurrency Control in Distributed Database Systems. ACM Computing Surveys. 13(2):185-220, jun/1981.

[Bernstein 87] BERNSTEIN P. A. et alii. Concurrency Control and Recovery in Databases Systems. Massachusetts, Addison-Wesley Publishing, Reading, 1987.

[Boksenbaum 87] BOKSENBAUM, C. et alii., Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems. IEEE Transactions on Software Engineering. 13(4):409-419, abr/1987.

[Casanova 85] CASANOVA, M. A. e MOURA, A. V. Principios de Sistemas de Gerência de Bancos de Dados Distribuidos. Rio de Janeiro, Editora Campus, 1985.

[Cart 89] CART, M., FERRIE, J. e RICHY, H. Contrôle de l'execution de transations concurrentes. Technique et Science Informatiques. 8(3):225-240, 1989.

[Cart 90a] Cart, M. et alii. An Optimistic Multi-level Control for Nested Typed Objects. Proceedings of the Twenty-Third Annual Hawaii International on System Sciences, vol. 2, 1990. Edited by Bruce D. Shriver.

[Cart 90b] CART, M. e FERRIE J. Integrating Concurrency Control into an Object-Oriented Database. Proc. 2nd International Conference on Extending Data Base Technology (EDBT 90), Venice, Itália, p. 363-377, mar/1990.

[Ceri 84] CERI, S. e PELAGATTI, G. Distributed Databases : Principles and Systems. McGraw-Hill 1984.

[Dahl 70] DAHL, O-J., MYRHANG, B. e NYGAARD, H. Simula Common Base Language. Norwegian Computing Center S-22, Oslo, Norway, 1970.

[Eswaran 76] ESWARAN, K. P. et alii. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM. 19(11), nov/1976.

[Garcia-Molina 83] GARCIA-MOLINA H. Using Semantic Knowledge for Transaction Processing in a Distributed Database. ACM TODS, 8(2):186-213, jun/1983.

[Goldberg 83] GOLDBERG, A. e ROBSON, D. Smalltalk-80 : The Language and Its Implementation. Massachusetts, Addison - Wesley, 1983.

[Hadzilacos 88] HADZILACOS V. A Theory of Reliability in Database Systems. JACM, 35(1):121-145, jan/1988.

[Haerder 83] HAERDER, T. e REUTER, A. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys. 15(4):287-317, dez/1983.

[Korth 89] KORTH, H. e SILBERSCHATZ, A. Sistema de Bancos de Dados. McGraw Hill, 1989.

[Kung 81] KUNG, H. T. e ROBINSON, J. T. On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems. 6(2):213-226, jul/1981.

[Lamport 78] LAMPORT, L. Time, CLocks and the Ordering of events in a Distributed System. Communications of the ACM. 21(7):558-565, jul/1978.

[Martin 87] MARTIN, B. Modeling Concurrent Activities with Nested Objects. Proc. 7th International Conference on Distributed Computing Systems, Berlin, Alemanha, Set/1987.

[Moss 82] MOSS, J. E. Nested Transactions and Reliable Distributed Computing. Proceedings of the 2nd IEEE Symposium on Reliability in Database Software and Databases Systems, Pittsburg, EUA, jul/1982.

[Moss 85] MOSS, J. E. Nested Transactions - An Approach to Reliable Distributed Computing. Cambridge, Massachusetts, MIT Press, 1985.

[Moss 86] MOSS J. E. et alli. Abstraction in Concurrency Control and Recovery Management. Techical Report COINS 86-20, University of Massachusetts at Amherst, Amherst, mai/1986.

[Rakow 90] RAKOW T. et alii. Serializability in Object-Oriented Database Systems. Proc. of IEEE Sixth International Conference on Data Engineering, Los Angeles, EUA, p. 112-120, Fev/1990.

[Schwarz 84a] SCHWARZ, P. Transactions on Typed Objects. Tese de Doutorado, Carnegie-Mellon University, Pittsburg, EUA, dez/1984.

[Schwarz 84b] SCHWARZ, P. M. e SPECTOR, A. Z. Synchronizing Shared Abstract Types. ACM Transactions on Computer Systems. 2(3):223-250, ago/1984.

[Sha 88] SHA L. et alii. Modular Concurrency Control and Failure Recovery. IEEE Transactions on Computers. 37(2):146-158, fev/1988.

[Stroustrup 86] STROUSTRUP, B. The C++ Programming Language. Addison-Wesley, 1986.

[Sugihara 87] SUGIHARA, K. Concurrency Control Based on Distributed Cycle Detection. Proceedings of the 3rd International Conference on Data Engineering, Los Angeles, fev/1987.

[Thomas 79] THOMAS, R. H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Database. ACM Transactions on Database Systems. 4(2):180-209, jun/1979.

[Turc 90] TURC, S. Comparasion of Immediate-Update and Workspace Transactions : Serializability and Failure Tolerance. Proceedings of the International Conference on DBs Parallel Architectures and their Applications, PARBASE 90, Miami Beach, EUA, mar/1990.

[Ullman 83] ULLMAN, J. D. Principles of Database Systems. Segunda Edição, Pitman Publishing Limited, Rockville, 1983.

[Weihl 88] WEIHL, W. E.. Commutativity-Based Concurrency Control for Abstract Data Types. IEEE Transactions on Computers. 37(12):1488, Dez/1988.

[Weihl 89] WEIHL, W. E. The Impact of Recovery on Concurrency Control (Extended Abstract). Proceedings of the 1989 ACM SIGMOD Conference.

[Weikum 87] WEIKUM, G. Enhancing Concurrency in Layered Systems. Proceedings of the 2nd International Workshop on High Performance Transaction Systems, 1987.