Controle de Concorrência e Distribuição de Dados: Teoria clássica, suas limitações e extensões modernas

João Eduardo Ferreira
jef@ime.usp.br

Marcelo Finger
mfinger@ime.usp.br

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Livro preparado especialmente para a Escola de Computação
Julho de 2000

A meus pais, Rosa e Salezy
M. F.

A minha família e em especial a minha esposa Renata
J. E. F.

Agradecimentos

Ao comitê da XII Escola de Computação, por ter nos dado a oportunidade de escrever este livro.

Aos alunos do curso de pós-graduação do DCC/IME/USP da disciplina MAC5763 — Controle de Concorrência e Distribuição de Dados, pelas inúmeras correções e sugestões feitas à versão preliminar deste texto.

Aos colegas do projeto SIDAM (Projeto Temático FAPESP 98/06138-2: Sistemas de Informação Distribuídos para Agentes Móveis) por proporcionar um rico ambiente de discussão.

Ao Departamento de Ciência da Computação do Instituto de Matemática e Estatística da USP, por prover a infra-estrutura e as condições para a realização deste trabalho (computadores, redes, impressoras e as toneladas de papel consumidas na elaboração deste texto).

Aos pesquisadores Mário Nascimento, Magareth Durham, Caetano Traina Jr., Pedro Pizzigatti Corrêa e Gisele Busichia por colaborações relevantes em diversas partes do texto.

Marcelo Finger foi parcialmente financiado pelo CNPq, bolsa de pesquisa PQ 300597/95-5. João Eduardo Ferreira foi parcialmente financiado pelo Programa BID-PNUD processo BR001.

Sumário

Capítulo 1

Introdução

1.1 Sistemas de Bancos de Dados e sua Evolução

Este livro trata da evolução de conceitos ligados a sistemas gerenciadores de bancos de dados. Tais conceitos servem de linha mestra para a evolução tecnológica da área e nos permitem entender melhor a complexidade dos sistemas computacionais que encontramos no presente e encontraremos no futuro. Quando a tecnologia evolui, ela cria um rastro formado por objetos antiquados e artefatos desatualizados. No caso dos sistemas de computação, estamos vendo com o passar dos anos a obsolescência de sitemas operacionais, progamas e versões de programas.

Mais importantes que os objetos e programas gerados pela tecnologia são as idéias e conceitos que serviram de base para a geração desta tecnologia. Enquanto objetos e versões antigas são relegados aos museus e ao esquecimento, a evolução das idéias forma um edifício conceitual de bases sólidas e de grandes dimensões. Seria impossível descrever os andares superiores sem antes explicarmos a base sobre a qual estes estão apoiados.

No nosso caso, em particular, nos ocuparemos das idéias e dos conceitos relativos à representação e acesso a dados em sistemas de de bancos de dados. O *Controle de Concorrência* trata dos métodos necessários para que mais de um agente (usuário ou programa) possa acessar um conjunto de dados partilhados entre todos eles. A *Distribuição de Dados* trata da disciplina necessária para distribuirmos e manipularmos dados armazenados em mais de um local.

Inicialmente, os sistemas de computação em geral, e os sistemas de bancos de dados em particular, não possuiam nem controle de concorrência nem permitiam distribuição de dados. Sobre este aspecto, podemos mencionar três fases da evolução dos sistemas:

1. *Sistemas centralizados mono-usuários.* Era um sistema em que apenas um programa era excutado de cada vez num computador.

   Com relação aos bancos de dados de então, sequer havia a distinção entre *bancos de dados* e *aplicações*, uma vez que os dados eram criados para uma única finalidade, sem se pensar em seu compartilhamento.

2. *Sistemas centralizados multi-usuários.* Inicialmente estes sistemas se caracterizavam pelo compartilhamento de uma mesma máquina (processador, memória, espaço em disco) por vários usuários. Com o passar do tempo, evolui-se para o compartilhamento de dados por mais de uma aplicação.

   Surgiram assim os Sitemas Gerenciadores de Bancos de Dados (SGBDs) centralizados em uma mesma máquina mas que permitiam acessos concorrentes de vários programas aos dados. Daí veio a necessidade de se ter um controle de concorrência de acessos aos dados, o que levou à teoria clássica do controle de concorrência.

3. *Sistemas distribuídos.* Os diversos computadores existentes começaram a ser conectados em rede, permitindo o acesso a dados remotos. Assim, sistemas distribuídos sempre foram multi-usuário.

   Os SGBDs também se distribuiram, sendo que um banco de dados passou a ter uma unidade lógica apesar de estar divididos em diversas máquinas. A função principal de um SGBD distribuído é a de manter a *transparência* na distribuição dos dados. Ou seja, tentaram fazer com que as aplicações acessassem os dados de maneira uniforme, independentemente da localização do dado. Baseada na noção de transparência, foi gerada a teoria clássica da distribuição de dados.

1.2 O Sistema Clássico

Com o surgimento dos sistemas multi-usuários e distribuídos, criou-se os concentos básicos de controle de concorrência e distribuição, aos quais chamamos de clássicos. Estes conceitos merecem o nome de "clássicos" pois todos as evoluções posteriores se baseiam tão fortemente neles que é impossível apresentar estes modernos sistemas sem se referir aos casos clássicos.

Com relação ao controle de concorrência dos acessos aos dados em bancos de dados, o conceito fundamental é a realização de todas as manipulações de dados através de *transações*. Uma transação é um programa com a propriedade especial de ser uma *unidade atômica de processamento de dados*. Por "unidade atômica" queremos dizer que ou todas as operações de uma transação são realizadas, ou nenhuma operação se realiza. Desta forma, uma transação ativa pode se *confirmar*, no caso em que a todas as operações são realizadas, ou pode ser *abortada*, o que na prática requer que as operações parcialmente realizadas pela transação sejam desfeitas.

Os diversos acessos concorrentes aos dados são realizados por transações distintas, e o controle de concorrência no modelo transacional clássico tem como objetivo garantir que o *escalonamento* das transações seja realizado de forma *correta*. Um escalonamento é considerado correto quando a execução de uma transação não é influenciado pela existência de outras transações sendo executadas concorrentemente.

O conceito de transação também está intimamente relacionado à *recuperação de falhas* em bancos de dados. Existem diversos tipos de falhas em um sistema, e a recuperação de falhas tem por objetivo garantir que o sistema seja colocado num estado consistente após a ocorrência de uma falha. O modelo transacional é de grande valia, pois permite identificar a condição de consistência facilmente. Num sistema transacional, a recuperação de falhas deve apenas garantir que as transações não confirmadas sejam todas desfeitas e as transações confirmadas tenham seus efeitos registrados no banco de dados.

Com o advento de sistemas distribuídos, tanto o controle de concorrência quanto a recuperação de falhas tiveram de ser extendidos para o caso distribuído. Como iremos ver, o controle de concorrência é facilmente extendido para o caso distribuído, pois o fato de uma transação ser executada em mais de uma máquina tem pouca influência sobre o controle de concorrência. Já a recuperação de falhas necessita de alguns cuidados especiais para ser realizada em ambiente distribuído, pois a consistência do sistema deve ser global e não apenas local em cada uma das máquinas.

Por outro lado, com a distribuição de dados surgem novos problemas com relação ao projeto, acesso e alocação dos dados espalhados por uma rede de computadores. Destes tópicos se ocupa a teoria clássica de distribuição de dados. O principal objetivo é o de fornecer *transparência* de acessos para as aplicações. Ou seja, uma determinada aplicação não deve ter que se preocupar com a localização de um determinado dado no sistema. A aplicação simplesmente requer o acesso ao dado e deixa a cargo do SGBD distribuído a tarefa de encontrar e operar tal dado. Esta tarefa fica ainda mais interessante quando, por motivos de eficiência e confiabilidade, há *replicação* dos dados em mais de um nó da rede.

1.3 Mobilidade de Dados e Computadores

A evolução dos sistemas computacionais não parou com os sistemas concorrentes e distribuídos. São várias as modificações que as aplicações modernas impõem à teoria clássica. Para fins expositivos, vamos concentrar estar imposições em duas categorias de repercução mais ampla nos processos de controle de concorrência e distribuição de dados:

- *Dados móveis*: os dados não estão mais fixos à nós da rede, mas podem se deslocar de uma máquina da rede para outra durante o processamento. A importância deste fato para a distribuição de dados não pode ser menosprezada. Um conjunto de novas estratégias de gerenciamento de posicionamento e acesso a dados se faz necessário.

- *Computadores móveis*: com o advento da computação móvel com conexões sem fio a rede, computadores podem estar fisicamente se deslocando durante o processamento de uma transação. Neste caso, o sistema possui uma parte móvel e uma parte fixa, todas conectadas numa mesma rede. Ao se deslocar, o computador pode mudar o ponto da rede fixa em que este está conecatado (via antena), permitindo assim a migração de transações de um nó fixo a outro. Temos então *transações móveis* como consequência da mobilidade. Para evitar esta mobilidade de transações podemos proibir que a transação seja executada nas partes móveis do sistema. Neste caso, teríamos o processamento todo sendo realizado na parte fixa do sistema, e o computador móvel será apenas um *cliente* que dispara a transação e recebe o seu resultado.

  Uma outra consequência da mobilidade é que os computadores móveis podem estar ativos mas desconectados da rede, pois o computador pode ter saído da área de abrangência das antenas do sistema móvel (por exemplo, no interior de um túnel). Este novo aspecto dos sistemas traz novos problemas que os sistemas clássicos não previam, gerando a necessidade de extendê-los.

Estes dois tipos de mobilidade geram sistemas que chamamos de *não-clássicos*. Além disso, estes dois tipos de mobilidade são independentes e podem ocorrer em separado ou conjuntamente. A tabela abaixo mostra as possíveis combinações:

Dado	*Transação*	*Cliente*
fixo	fixa	fixo
fixo	móvel	móvel
móvel	fixa	móvel/fixa
móvel	móvel	móvel

A primeira linha corresponde ao caso clássico. A segunda linha corresponde ao que se chama de computação móvel. A terceira linha corresponde à simples mobilidade do dado, não importando se o cliente que dispara a transação está fixo ou móvel. A quarta linha corresponde a interação total dos dois conceitos, onde tanto o dado se desloca pela rede quanto os computadores que estão processando as transações. Em todos estes casos estamos assumindo que o SGBD é fixo, ou seja, que está residente na parte fixa da rede.

Ao tratarmos da parte não-clássica da teoria de controle de concorrência e distribuição de dados, estaremos indo de encontro ao tratamento destes casos acima listados.

1.4 A Organização Deste Livro

Este livro se organiza em duas partes. A primeira trata dos conceitos clássicos de controle de concorrência, recuperação e distribuição de dados em bancos de dados. As segunda parte trata das mudanças e extensões aos modelos clássicos requisitados pelas modernas aplicações de bancos de dados.

Na primeira parte, temos três capítulos.

O Capítulo 2 apresenta a teoria clássica do controle de concorrência. Nele são apresentados os modelo de acessos a dados por transações, as propriedades básicas dos escalonamentos de transações concorrentes. A Teoria da Seriação trata dos critérios que os escalonamentos de transações devem cumprir para que as propriedades básicas das transações, chamadas de propriedades ACID, sejam respeitadas. Dois métodos básicos são estudados para o controle de concorrência: o protocolo de Bloqueios Bifásicos (em inglês, *two-phase locking*) e o método de ordenação por Marcas de Tempo (*timestamping*).

O tratamento, nesta parte introdutória, é rigoroso, pois dele depende o entendimento do resto dos conceitos apresentados no restante do livro. No entanto, focalizamos nossa apresentação nos conceitos e recorremos ao formalismo apenas quando este pode clarificar alguns aspectos centrais da teoria. Desta forma, recorremos à demonstração de teoremas apenas quando estas demonstrações são construtivas e permitem entender melhor a implementação dos conceitos em questão. A menos destes casos, a apresentação é tratada informalmente.

O Capítulo 3 apresenta as noções de recuperação de falhas em sistemas de bancos de dados baseados em transações. O conceito de "falha" é discutido, e apresentamos as suas implicações na arquitetura interna de um SGBD. O gerenciamento de memória e o gerenciamento de recuperação são discutidos em detalhes, e os algoritmos básicos do controle clássico são apresentados. Mostramos como este gerenciamento de recuperação impacta sobre o controle de concorrência, uma vez que a implementação básica das operações de uma transação é feita levando-se em consideração a recuperação de falhas. Por fim, mostramos como a distribuição de dados se inter-relaciona com o gerenciamento de recuperação.

Os fundamentos sobre projeto, acesso e alocação de dados em bancos de dados distribuídos está descrito no Capítulo 4.

O Capítulo 4 apresenta os fundamentos da distribuição de dados, através de uma metodologia para análises dos requisitos do projeto lógico e físico de distribuição de dados. Os conceitos de autonomia, distribuição e heterogeneidade são apresentados de modo a caracterizar os Sistemas de Informação fortemente integrados, semi-autônomos e totalmente isolados. As abordagens ascendente e descendente para a distribuição de dados são tratadas e como exemplo é apresentado a metodologia DATAID-D. As técnicas de fragmentação horizontal e vertical de dados em modelos relacionais também são apresentadas. Utilizamos respectivamente os conceitos de minitermos e matriz de afinidades para viabilizar as fragmentações horizontais e verticais. Finalizamos esse capítulo apresentando um método geral para alocação de fragmentos baseado num modelo de custos de processamento e armazenamento.

A segunda parte do livro trata das extensões e alterações na teoria clássica que se fazem necessárias para dar suporte às modernas aplicações de bancos de dados.

Os Capítulos 5 e 6 tratam de sistemas de controle de concorrência que violam algumas das propriedades ACID das transações. O Capítulo 5 discute algumas hipóteses implícitas na elaboração da teoria clássica de controle de concorrência que são violadas pelas aplicações modernas. Para tratar destas violações, apresentamos três novos modelos de controle de concorrência: o modelo de transações aninhadas, o modelo baseado em comutatividade de operações e, por fim, o modelo de transações compensatórias.

Já o Capítulo 6 se concentra na aplicação destes novos modelos em sistemas modernos. Discutimos as transações partidas e sua aplicação em sistemas de CAD. Em seguida, mostramos como estas transações partidas se aplicam a Multibancos de dados. Por fim, apresentamos o modelo Canguru de transações móveis. Tal modelo é interessante, pois é um amalgamado de todas as alterações do modelo ACID vistas no livro, ou seja, o modelo canguru engloba as transações aninhadas, compensatórias e partidas, que são realizadas num ambiente distribuído de multibancos de dados.

Os Capítulos 7 e 8 tratam de formas não convencionais de ditribuição de dados. O Capítulo 7 apresenta uma técnica de modularização do esquema conceitual global de dados baseada na caracterização de cada subsistema de informação. Tal técnica se aplica entre após a geração do esquema global de dados e antes do projeto de distribuição. Adicionou-se à metodologia de modelagem de dados duas novas fases: análise de requisitos de modularização e a modularização propriamente dita. Conceitos adicionais são apresentados como o de compartihamento unidirecional e multidirecional e as etapas necessárias para modularização. O objetivo de tal técnica é diminuir a complexidade do projeto de distribuição convencional de dados, fazendo com que a análise dos fragmentos recaia sobre os módulos de dados e não sob o esquema global de dados.

O Capítulo 8 apresenta os conceitos para distribuição de dados com dados móveis. Para abordar tal mobilidade os mecanismos de autorização convencionais não são adequados. Os conceitos de autorização fraca, forte, implícita, explícita e as regras de derivação são apresentados considerando usuários, objetos e tipos de autorizações. Esses conceitos apontam para uma nova abordagem dos mecanismos não convencionais para o controle de autorização. Outro aspecto apresentado nesse capítulo se refere ao processo de extração de dados para disponibilidade em sistemas alocados em outras máquinas da rede ou de computadores móveis. Os conceitos de separação, evolução e re-integração de dados são apresentados. Os dados extraídos são encapsulados em objetos denominados aqui de objetos fragmentos. Para controlar os tipos de operações de escrita e leitura sobre os dados encapsulados classificou-se seis tipos de vínculos possíveis entre os dados originados da base de dados e suas possíveis réplicas. Para controlar os processos de reconcialação de réplicas, as idéias dos algoritmos de multiversão de dados e os baseados em log são apresentados.

Parte I

Abordagem Clássica

Capítulo 2

Teoria Básica do Controle de Concorrência

2.1 O Modelo de Transações

O *controle de concorrência* é o mecanismo que permite diversos programas acessarem um conjunto de recursos compartilhados. O objetivo do controle de concorrência é realizar o compartilhamento de todos estes recursos de forma *transparente*, ou seja, sem que um programa precise saber da existência de outros programas tentando acessar os mesmos recursos. Desta forma, cada programa pode se comportar como se todos os recursos estivessem a sua disposição, e cabe ao controlador de concorrência intermediar o acesso aos recursos compartilhados e garantir que cada programa possa ser executado de forma *independente* de todos os outros.

Obviamente, para que os programas possam acessar concorrentemente recursos compartilhados de forma independente certas restrições a estes programas precisam ser impostas. Neste caso, dizemos que os programas devem satisfazer um *modelo de comportamento.*

O elemento principal do modelo de comportamento que estudamos neste livro é a *transação.* Transações são os programas que acessam os recursos compartilhados; uma *aplicação* pode ser composta de uma ou mais transações.

Um exemplo de uma aplicação é um programa que realiza a transferência de valores entre contas. Tal programa realiza duas transações num banco de dados compartilhado: a primeira, que verifica se o usuário possui saldo suficiente para pagar um montante; e a segunda, que realiza a transferência do montante, debitando de uma conta e creditando na outra.

O objetivo do controle de concorrência é garantir que as transações sejam executadas *atomicamente*, o que quer dizer:

- se transação terminar com sucesso, todas as alterações aos recursos compartilhados serão permanentes;
- se a transação terminar sem sucesso (dizemos que a transação foi *abortada*) ela não deve ter nenhum efeito; todas as alterações feitas pela transação abortada devem ser desfeitas.

Desta forma, uma transação é um programa que ou sucede na íntegra, ou é totalmente desfeito. Ou tudo ou nada.

No caso da transferência de valores, o sucesso significa que o montante saiu de uma conta e entrou na outra; o fracasso significa que as duas contas permanecem como estavam.

Outra restrição do nosso modelo é o fato de os únicos recursos compartilhados que estaremos estudando são *dados compartilhados*. O conjunto destes dados compartilhados é que chamamos de *banco de dados*. Além disso, estaremos assumindo que não há nenhuma forma de comunicação entre as transações sendo executadas concorrentemente no sistema a não ser através dos dados compartilhados. Ou seja, para uma transação enviar uma mensagem a outra, a única forma possível é a primeira escrever o dado no banco de dados, que será posteriormente lido pela segunda.

Um esclarecimento se faz necessário quanto à natureza da *concorrência* da forma como o nosso modelo a encara. Um sistema pode ser concorrente e ter apenas um processador funcionando em esquema de *tempo repartido*[1] entre os diversos processos nele sendo executados. Um sistema concorrente pode ter vários processadores executando em paralelo (cada um, por sua vez, pode ou não estar funcionando num esquema de tempo repartido). Um sistema concorrente pode ser formado por diversos computadores interligados em rede, com os dados distribuídos geograficamente (as transações, neste caso, seriam transações distribuídas). Todos estes casos são cobertos pelo nosso modelo de concorrência e as transações, quer locais quer distribuídas, devem satisfazer os critérios de atomicidade e independência.

O modelo de transações aqui adotado tem a característica de ser *abstrato* com relação às operações realizadas pelas transações. Vamos detalhar em que consiste esta abstração. Em primeiro lugar, precisamos esclarecer

[1] Este conceito é conhecido em inglês como *time sharing*, e consiste no compartilhamento do processador por diversos processos que se alternam ao longo do tempo na posse do processador (execução).

a distinção entre o *código* de uma transação e as *operações realmente executadas* por ela. O código da transação pode estar escrito em qualquer linguagem de programação (C, Pascal, Cobol, VB, Java, LISP, etc.) e pode conter uma série de construções destas linguagens (IF-THEN-ELSE, ciclos, variáveis globais, chamadas ao sistema operacional, etc). Dependendo dos valores de entrada lidos pelo programa, diversos caminhos distintos podem ser percorridos no código. Mas, é importante salientar que

o modelo de transações abstrai (ignora) o código da transação.

O modelo de transações se preocupa apenas com as operações realmente executadas de leitura e escrita em ítens de dados compartilhados. Todos os outros detalhes da execução do programa são ignorados. Desta forma, o modelo consegue ser genérico e independente da linguagem em que as transações foram escritas.

A abstração do modelo vai além disso. O valor lido na operação de leitura de um dado x ($r[x]$) é ignorado. Ou seja, em vez de nos ocuparmos com algo do estilo:

$$\alpha := r[x]$$

simplesmente registramos que o item de dados x foi lido. Da mesma forma, o valor escrito no item de dados y não é levado em conta, e em vez de escrevermos

$$write(y, \langle \text{valor} \rangle)$$

anotamos apenas que algum valor foi escrito em y, $w[y]$. Desta forma conseguimos também independência de tipos de dados. Os itens de dados podem representar qualquer elemento no banco de dados: um componente de um registro, o registro todo, a tabela toda, um elemento do índice de uma tabela, etc. Nosso modelo abstrai toda essa informação, o que é responsável pela sua grande aplicabilidade.

A figura 2.1 mostra um exemplo uma transação de transferência de um valor v entre as contas x e y, e sua correspondente abstração.

Desta forma, as operações que nos interessam em uma transação i são as seguintes:

- O seu início. No entanto, em geral, não usamos um notação especial para o início de uma transação.

- As operações de leitura da transação i de um item de dado x, $r_i[x]$, e de escrita de i em um item de dados x, $w_i[x]$.

Transação	*Abstração*
BEGIN TRANSACTION	
/* *Decrementa x de v* */	
X := READ(x);	$r[x]$
X := X - v;	
WRITE(x, X)	$w[x]$
/* *Incrementa y de v* */	
Y := READ(y);	$r[y]$
Y := Y + v;	
WRITE(y, Y);	$w[y]$
COMMIT	c

Figura 2.1: Uma transação de transferência de v de x para y e sua abstração

- O término da transação i. Uma transação pode ter um dentre dois tipos de término: o término bem-sucedido, em que a transação *confirma* as alterações processadas nos itens de dados[2], representado por c_i; ou o término mal-sucedido, em que a transação é abortada e todos os seus efeitos devem ser desfeitos, representado por a_i. Assumimos que todas as transações terminam, o que pode ser garantindo abortando todas as transações que estiverem em execução por um tempo maior do que um limite previamente determinado.

  Uma transação que foi iniciada mas que ainda não se confirmou nem abortou é dita *ativa*. Os estados de uma transação estão ilustrados na Figura 2.2.

Desta forma, uma transação será formada por uma série de leituras e escritas em itens de dados (abstraídos nos símbolos, x, y, z, etc.) e, sempre, uma dentre as duas operações de término, confirmação ou aborto. Além disso, para simplificar a notação e sem nenhum prejuizo lógico, assumimos que um item de dado é lido ou escrito por uma transação apenas uma vez — do ponto de vista lógico, o que causa efeitos permanentes é apenas a primeira leitura e a ultima escrita. Assumimos também que as operações de escrita/leitura/confirmação/aborto de cada transação são executadas atomicamente, ou seja, ou são executadas na íntegra, ou não são executadas.

[2]O termo em inglês é COMMIT, que já foi traduzido para o português como *comprometimento*, *compromissamento* ou até *cometimento*. Todos estes termos nos parecem artificiais, e o termo *confirmação* traduz com maior fidelidade a semântica do término bem-sucedido de uma transação.

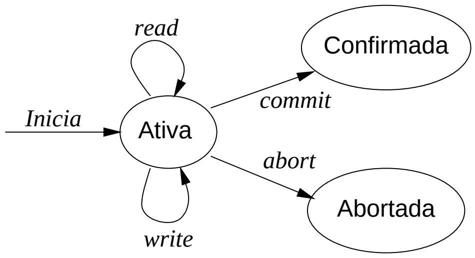


Figura 2.2: Estados de uma transação

Pode parecer que nosso modelo de transações está ignorando um aspecto muito importante do banco de dados, que são as inserções e remoções de tuplas (ou registros) nas tabelas, pois estas diferem bastante da operação de leitura e escrita de registros. No entanto, a generalidade de nosso modelo é capaz de sanar esta aparente deficiência. Num banco de dados, a inserção/remoção de um registro é sempre acompanhada de (pelo menos) uma escrita no índice da tabela correspondente. Como não restringimos o que pode ser os itens de dados do nosso modelo, nada impede de que sejam integrantes dos índices das tabelas. Desta forma, uma inserção/remoção de um item de dados está modelada como uma escrita num outro elemento do banco de dados associado a este (normalmente um ponteiro).

O principal motivo para querermos ter a execução concorrente de várias transações é o ganho em eficiência. Um acesso de escrita ou leitura a disco é várias ordens de grandeza mais lento que um acesso à memória primária. Se tivermos apenas uma transação sendo executada, o processador ficará ocioso durante o tempo de acesso a disco. Com a execução concorrente, o tempo de utilização do processador aumenta. Mas o mais significativo é que o tempo de execução concorrente de um conjunto de transações é muito menor que a soma dos tempos de execução individual de cada uma delas.

Como estamos lidando com a execução concorrente de transações, mais de uma transação poderá estar sendo executada ao mesmo tempo. Isto quer dizer que diversas operações de diversas transações podem estar sendo executadas ao mesmo tempo. A concorrência das operações de transações concorrentes será modelada como uma *ordem parcial* de operações, conforme será detalhado na Seção 2.3. Uma dada ordem (parcial) de operações das diversas transações concorrentes é chamado de *escalonamento*.

Nem todos os escalonamentos são admissíveis. Existem vários escalonamentos indesejáveis:

- escalonamentos não permitem que transações abortadas sejam desfeitas consistentemente;

- escalonamentos que são capazes de forçar que transações confirmadas sejam desfeitas.

- escalonamentos que não permitem que transações sejam executadas concorrentemente de forma independente, causando a interferência entre transações.

Desta forma, o *Escalonador* é um elemento fundamental da arquitetura de um sistema de um gerenciador de bancos de dados genérico, ilustrado da Figura 2.3. Esta arquitetura assume que o sistema está rodando num ambiente centralizado, mas pode ser estendida para um ambiente distribuído. O escalonador é responsável por evitar a ocorrência de escalonamentos indesejáveis e de garantir as propriedades básicas dos escalonamentos descritas a seguir.

Acima do Escalonador na arquitetura genérica [3] está o o Gerenciador de transações, que é o responsável pela interface do gerenciador de banco de dados com as aplicações externas ao banco de dados. O Gerenciador de Transações realiza um pré-processamento das informações recebidas das transações e as envia ao Escalonador.

Abaixo do Escalonador está o Gerenciador de Acesso a Dados (GD). Este módulo possui duas partes: o Gerenciador de Memória (GM), responsável pelos gerenciamento dos buffers que contém um espelho em memória primária de parte do conteúdo do banco de dados que está em disco; a outra parte, o Gerenciador de Recuperação (GR), é responsável pelo gerenciamento de recuperação, que durante a operação normal realiza a manutenção do log (uma espécie de diário) de transações, anotando todas as operações básicas de leitura/escrita, confirmação/aborto das transações. Este log é usado no desfazer de transações abortadas e no procedimento de reinício do sistema após falhas, restaurando o sistema a um estado consistente.

O Gerenciador de Acessos a Dados, bem como suas duas sub-partes, o Gerenciador de Cache e o Gerenciador de Recuperação, estão descritos em detalhes no Capítulo 3.

A seguir, detalhamos as propriedades fundamentais do escalonamento de transações.

[3] Esta arquitetura é apenas didática e não quer dizer que gerenciadores de bancos de dados a sigam a risca.

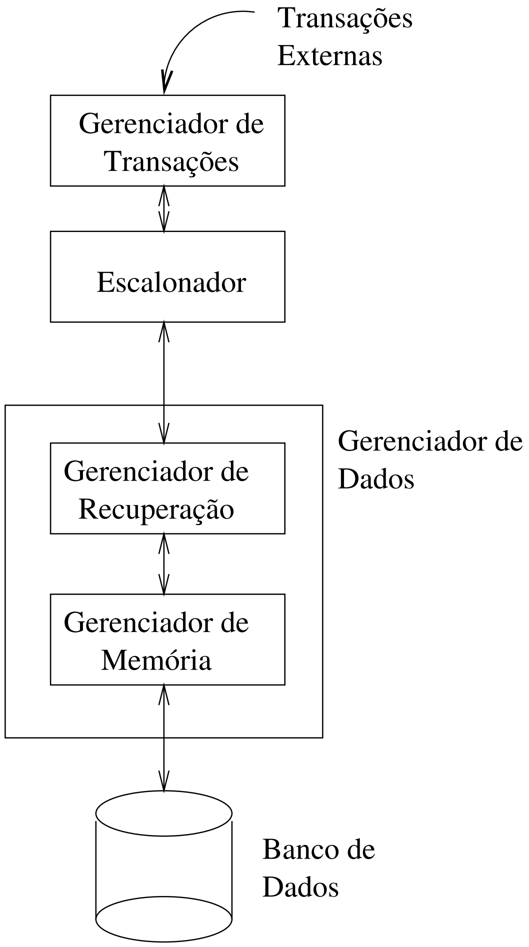


Figura 2.3: Arquitetura genérica de sistema de bancos de dados

2.2 Propriedades Básicas de Transações

Modelos de concorrência baseados em transações começaram a aprecer no início da década de 70, e pelo meio da década estes conceitos já haviam se consolidado, levando à publicação de trabalhos muito influentes [EGLT76, SLR76]. Antes disso, o problema de controle de concorrência já vinha sendo tratado no âmbito de sistemas operacionais, por exemplo em [BH73].

Na seção anterior delineamos o modelo básico do controle de concorrência baseado em transações. Nesta seção, iremos detalhar melhor as propriedades básicas das transações. Estas propriedades estão definidas em três grupos. Inicialmente, temos as propriedades fundamentais, que independem do controle de concorrência, conhecidas como propriedades *ACID*; em seguida, temos as propriedades dependentes do controle de concorrência de transações; e por fim, temos o critério de correção de escalonamentos, ou seja a condição para que um escalonamento de transações com operações concorrentes seja aceito como *correto*.

2.2.1 As Propriedades *ACID*

Descrevemos aqui as propriedades básicas que consagraram o modelo de transações, e que devido a suas iniciais ficaram conhecidas como propriedades *ACID*:

Atomicidade Uma transação é uma unidade atômica de processamento que deve ser executada integralmente, ou totalmente desfeita.

Consistência A execução de uma transação deve levar o banco de dados de um estado consistente a outro.

Por consistência entende-se que todas as restrições de integridade do banco de dados (por exemplo: unicidade de chaves, integridade referenciais, restrições de integridade lógicas, etc) estão respeitados no início da transação e no seu término.

Isolamento ou independência A execução de uma transação não pode ser afetada por outras transações sendo executadas concorrentemente. Desta forma, cada transação não deve tornar suas atualizações visíveis a outras transações até que a transação seja confirmada (enviando um COMMIT). Decorre desta propriedade o conceito de *escalonamento correto* como aqueles em que as transações são *seriáveis*, descrito na Seção 2.3.

Durabilidade ou persistência Os efeitos de uma transação confirmada não podem ser desfeitos. As alterações de uma transação confirmada só podem ser alterados por outra subseqüente transação confirmada e não podem ser perdidas nem mesmo no caso de uma eventual falha.

2.2.2 Propriedades do Controle de Concorrência

As diversas transações rodando em um sistema enviam suas operações (leitura, escrita, confirmação e aborto) em uma ordem qualquer, e o escalonador deve decidir em que ordem estas operações devem ser executadas para garantir as propriedades ACID (além de garantir o critério de correção de escalonamentos detalhado abaixo). As transações lêem dados e escrevem em dados produzidos por outras transações e isto pode causar interações indesejáveis entre transações.

Primeiramente, vamos definir precisamente de que forma esta interação ocorre.

Dizemos que uma transação T_2 *lê um dado x de uma transação* T_1 se:

- T_2 leu o valor de x depois que T_1 escreveu este valor.
- T_1 não abortou antes que T_2 leu x.
- Se alguma transação escreveu em x entre a escrita de T_1 e a leitura de T_2, esta transação foi abortada e desfeita.

De forma totalmente análoga, definimos que uma transação T_2 *escreve* em um dado x de uma transação T_1.

A primeira iteração entre transações que se deseja evitar é que *uma transação confirmada não leia um dado de uma transação abortada*, conforme ilustrado abaixo:

$$w_a[x] \;\; r_c[x] \;\; w_c[y] \;\; \mathbf{commit}_c \;\; \mathbf{abort}_a$$

Uma possível interpretação destes dados é a seguinte: imagine x como sendo o saldo de uma conta e y é o valor dos juros a serem creditados nesta conta. Suponhamos que T_a atualizou o saldo da conta e T_c computa os juros devidos. Se T_a aborta, o valor computado por T_c é inconsistente.

Obviamente, pela definição acima, no momento que T_c leu de T_a, tanto T_c quanto T_a estavam ativas. A transação abortada T_a vai ter suas atualizações desfeitas, inclusive o dado lido pela transação confirmada T_c; neste caso, T_c leu um dado inconsistente, mas como a transação já está confirmada, não pode ser abortada nem desfeita. A transação T_c está *irrecuperável*.

Um escalonamento é portanto dito *recuperável* quando uma transação só se confirma se todas as outras transações das quais ela leu dados já estiverem confirmadas.

Num escalonamento recuperável, podemos ter o fenômeno de *cascateamento de abortos*. Este fenômeno está ilustrado no seguinte escalonamento (onde podemos imaginar T_1 atualizando o saldo e T_2 computando os juros):

$$w_1[x] \;\; r_2[x] \;\; w_2[y] \;\; a_1$$

Neste caso, T_2 leu x de T_1, e em seguida T_1 foi abortada e suas atualizações foram desfeitas. Como T_2 leu um dado inconsistente, a única solução para a manutenção da consistência e recuperabilidade do sistema é abortar T_2 também:

$$w_1[x] \;\; r_2[x] \;\; w_2[y] \;\; a_1 \;\; a_2$$

Desta forma, o aborto de T_1 provocou o aborto de T_2, que descreve o fenômeno de cascateamento de abortos. O que causou o aborto de T_2 foi o fato desta transação ler um dado de uma outra transação que ainda estava ativa, o que motiva a seguinte definição. Um escalonamento é dito *livre de cascateamento de abortos* se uma transação só lê dados de transações confirmadas. Note que um escalonamento livre de cascateamento de abortos é sempre recuperável.

Finalmente, um ultimo tipo de escalonamento que queremos evitar é um que dificulta o desfazer das atualizações de uma transação abortada. A estratégia mais simples para se desfazer uma transação abortada é o *restabelecimento dos valores prévios* dos itens de dados alterados pela transação. No entanto, considere os seguintes escalonamentos:

$$\begin{array}{ll} (i) & w_0[x] \;\; c_0 \;\; w_1[x] \;\; a_1 \\ (ii) & w_0[x] \;\; c_0 \;\; w_1[x] \;\; w_2[x] \;\; a_1 \end{array}$$

Em (i), para desfazer a transação abortada T_1, basta restabelecer o *valor prévio* do dado x (que é o valor escrito por T_0). Já no caso de (ii), o restabelecimento do valor prévio da variável x estará errado, pois estará destruindo a atualização de T_2 em x e o certo seria não alterar o valor de x. Ainda em (ii), se T_2 abortar após o aborto de T_1, o restabelecimento do valor prévio irá retornar x ao valor escrito por T_1, que é um valor incorreto. O que causou o problema em (ii) foi claramente o fato de T_2 ter escrito sobre o valor escrito por T_1

Para evitar estes problemas no desfazer de uma transação abortadas, definimos como escalonamentos *estritos* aqueles nos quais as transações lêem dados e escrevem sobre dados apenas de transações confirmadas. O caso (ii) estaria portanto proibido num escalonamento estrito. Escalonamentos estritos são sempre livres de cascatementos de abortos.

As classificações aqui apresentadas são devidas a [Had83, Had88].

2.2.3 Critério de Correção de Escalonamentos

Vamos ver um exemplo no qual, mesmo com escalonamento estrito, ainda temos um comportamento indesejável de uma transação. Suponhamos que temos duas transações que realizam depósitos em uma conta. A conta possui $1000,00, a transação T_1 deposita $100,00 e a transação T_2 deposita $20,00. Note o que ocorre com a seguinte execução concorrente:

T_1	T_2
`SaldoAtual := Read( SALDO )`	`SaldoAtual := Read( SALDO )`
	`SaldoAtual := SaldoAtual + 20`
	`Write( SALDO, SaldoAtual )`
	$\texttt{COMMIT}_2$
`SaldoAtual := SaldoAtual + 100`	
`Write( SALDO, SaldoAtual )`	
$\texttt{COMMIT}_1$	

Note que este escalonamento é estrito: a transação T_1 está escrevendo sobre um valor confirmado de T_2. No entanto, o saldo ao final destas duas atualizações é de $1100,00 e não de $1120,00, como seria o desejado. Os efeitos da transação T_2 foram apagados pelo da transação T_1, e isto é um efeito altamente indesejável, violando o critério de durabilidade acima.

O escalonamento do exemplo violou o *critério de correção de escalonamentos concorrentes*, que afirma o seguinte:

> A execução concorrente de diversas transações deve ter um efeito no banco de dados *equivalente* ao de alguma execução serial destas transações.

A execução serial é aquela em que uma, e só uma transação, é executada de cada vez. Desta forma, se tivermos três transações, T_1, T_2 e T_3 sendo executadas serialmente, uma execução concorrente correta deve ter os mesmos efeitos que uma das seguintes execuções seriais:

$$\begin{array}{ccc} T_1;T_2;T_3 & T_2;T_1;T_3 & T_3;T_1;T_2 \\ T_1;T_3;T_2 & T_2;T_3;T_1 & T_3;T_2;T_1 \end{array}$$

Um escalonamento que obedece ao critério de correção acima é dito *seriável*. É importante notar que o critério de correção de um escalonamento é independente das outras propriedades. Como vimos, podemos ter escalonamentos estritos que não são seriáveis. Semelhantemente, podemos ter escalonamentos seriáveis que, no entanto, não são estritos. E o mesmo ocorre com as outras propriedades descritas na Seção 2.2.2.

A definição acima do critério de seriabilidade foi feita de forma demasiada informal. Em especial, não está claro quando um escalonamento pode ser considerado *equivalente* a outro. Tampouco está claro como se pode detectar eficientemente esta equivalência. Para tratar destes tópicos, desenvolvemos na próxima seção a *Teoria da Seriabilidade*.

2.3 Teoria da Seriabilidade

A partir do final da década de 70, o problema do controle de concorrência começou a ser tratado formalmente, com um sério embasamento teórico por trás [BSW79, SLR76, Pap79]. O conceito de seriabilidade aqui discutido foi inicialmente proposto em [GLPT75] e a nossa apresentação segue [BHG87]; outro tratamento completo sobre seriabilidade pode ser encontrado em [Pap86].

O intuito da teoria da seriabilidade é tratar de maneira rigorosa os elementos necessários para construirmos e detectarmos eficientemente escalonamentos que obedeçam ao critério de correção exposto na Seção anterior.

Para tanto, precisamos usar de métodos matemáticos rigorosos. Felizmente, estes métodos são bastante elementares e não usam mais que noções de *ordens parciais* em conjuntos.

2.3.1 Ordens Parciais

Por exemplo, considere um conjunto com seis elementos, $\Sigma = \{a, b, c, d, e, f\}$. Uma ordem parcial sobre Σ pode ter o seguinte aspecto:

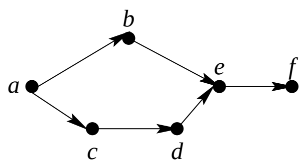


Figura 2.4: Ordem parcial

Uma ordem parcial é um conjunto de pares ordenados, respeitando algumas restrições. No caso do exemplo da figura 2.4, nós temos representado que o elemento a é menor que o elemento b, a é menor que c, mas b e c não estão relacionados pela ordem. Se representarmos a ordem por $<_\Sigma$, então podemos escrever:

$$a <_\Sigma b \quad a <_\Sigma c \quad b \not<_\Sigma c$$

O fato de $a <_\Sigma b$ para nós irá representar que o evento a ocorreu antes que o evento b. Já o fato de b e c não estarem ordenados por $<_\Sigma$ representa o fato de que b e c estão ocorrendo concorrentemente. Desta forma, temos um modelo bem simples e eficaz para representar eventos que ocorrem em série e eventos concorrentes.

Algumas outras propriedades merecem atenção. Por exemplo, na a figura 2.4, vemos que tanto $a <_\Sigma b$ quanto $b <_\Sigma c$ e, portanto, o evento a ocorre antes de c, $a <_\Sigma c$; ou seja, a ordem parcial $<_\Sigma$ é *transitiva*. Notamos também que não há nenhuma seta de um nó para si mesmo, ou seja, para nenhum elemento $x \in \Sigma$, $x <_\Sigma x$; esta propriedade é a *irreflexividade*. Estamos em condições, então, de definir formalmente as ordens parciais.

Definição 2.3.1 Uma *ordem parcial (irreflexiva)* $<$ sobre um conjunto Σ é uma relação binária em $\Sigma \times \Sigma$ (ou seja, é um conjunto de pares ordenados), representada por $(\Sigma, <)$, satisfazendo as seguintes restrições:

- A ordem $<$ é transitiva: para todo $x, y, z \in \Sigma$ se $x < y$ e $y < z$ então $x < z$.

- A ordem $<$ é irreflexiva: para todo $x \in \Sigma$, $x \not< x$.

O exemplo da Figura 2.4 ainda nos mostra que uma ordem parcial pode ser vista como um grafo dirigido; e, mais do que isso, como a ordem parcial é irreflexiva, concluímos que o grafo dirigido deve ser *acíclico*. A representação da Figura 2.4 é um grafo dirigido acíclico.

Uma restrição de uma ordem parcial $(\Sigma_1, <_1)$ é uma outra ordem parcial $(\Sigma_2, <_2)$ tal que $\Sigma_2 \subseteq \Sigma_1$ e $<_2$ é obtido apagando-se de $<_1$ todos os pares que possuem elementos que não pertencem a Σ_2. Um *prefixo* de uma ordem parcial $(\Sigma_1, <_1)$ é uma outra ordem parcial $(\Sigma_2, <_2)$ que a restringe, de forma que se $x \in \Sigma_2$ então todo $y <_1 x$ também está em Σ_2.

Numa ordem parcial $(\Sigma, <)$, uma *origem* é um um elemento de Σ que não é precedido de nenhum outro elemento. Se Σ é um conjunto finito, então necessariamente existe pelo menos uma origem em Σ; caso contrário, todos os elementos de Σ teriam um antecessor, o que significa que ou Σ é infinito (contrariando a hipótese de ser finito) ou há um ciclo em $(\Sigma, <)$, contrariando a irreflexividade de $<$.

Uma ordem parcial finita $(\Sigma, <)$ pode ter mais de uma origem. Se retirarmos uma origem u qualquer de Σ, apagando todos os pares (u, x) em $<$, obteremos uma nova ordem parcial $(\Sigma', <')$, pois a eliminação de uma origem não pode violar nem a transitividade nem a irreflexividade da ordem. Se Σ' não for vazio, a ordem parcial $(\Sigma', <')$ possuirá pelo menos uma origem. Podemos repetir este processo, apagando origens e obtendo a seqüência $u_1, u_2, \ldots, u_n$, onde n é o número de elementos de Σ. Esta seqüência é chamada de uma *ordenação topológica* de Σ. Toda ordem parcial, portanto, possui pelo menos uma ordenação topológica. Esta propriedade será importante na determinação de uma condição de seriabilidade de transações.

2.3.2 Transações e Escalonamentos

Agora que nós já sabemos o que é uma ordem parcial, nós já podemos formalizar as noções de transações e escalonamentos.

Definição 2.3.2 Uma *transação* T_i é uma ordem parcial com ordenação $<_i$ tal que:

1. $T_i \subseteq \{r_i[x], w_i[x] | x \text{ é um dado}\} \cup \{a_i, c_i\}$.

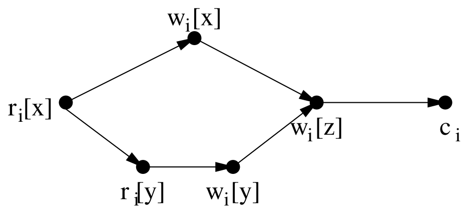
   Ou seja, T_i é um conjunto finito composto apenas de operações de leitura, operações de escrita, e operações de terminação, a_i ou c_i.

2. T_i possui uma e apenas uma operação de terminação: $c_i \in T_i \Longleftrightarrow a_i \notin T_i$.

3. A operação de terminação t_i é a última operação da transação: $p_i <_i t_i$ para toda operação p_i, $t_i = a_i$ ou $t_i = c_i$.

4. Se $r_i[x], w_i[x] \in T_i$, então $r_i[x] <_i w_i[x]$ ou $w_i[x] <_i r_i[x]$ (operações sobre o mesmo dado devem estar ordenadas).

É importante notar que a nossa definição permite que existam operações concorrentes dentro da própria transação. Se p_i e q_i são operações e nem $p_i <_i q_i$ nem $q_i <_i p_i$, isto indica que p_i e q_i são duas operações internas a T_i que são concorrentes. No exemplo abaixo de uma transação T_i,



a operação $w_i[x]$ é concorrente com $r_i[y]$ e com $w_i[y]$; as operações sobre o mesmo dado $r_i[x]$ e $w_i[x]$, e $r_i[y]$ e $w_i[y]$ estão ordenadas; e a operação c_i sucede todas as outras.

O próximo passo é a definição de escalonamentos. Um escalonamento vai ordenar (parcialmente) *todas* as operações de um conjunto de transações $\{T_1, T_2, \ldots, T_n\}$. No entanto, é importante que um escalonamento não permita que *operações conflitantes* entre as transações sejam concorrentes. Duas operações de transações distintas, p_i e q_j são chamadas de conflitantes se:

- operarem sobre o mesmo item de dado; e
- pelo menos uma delas for uma operação de escrita.

Desta forma, para um dado x existem três possíveis pares de operações conflitantes entre duas transações T_i e T_j: $\{r_i, w_j\}, \{r_j, w_i\}$ e $\{w_i, w_j\}$.

A definição formal de escalonamento é feita em duas fases: primeiro tratamos de *escalonamentos completos* para em seguida definirmos o que é um escalonamento.

Definição 2.3.3 Um *escalonamento completo* sobre o conjunto de transações $\{T_1, T_2, \ldots, T_n\}$ é uma ordem parcial $(E, <_E)$ tal que:

- $E = T_1 \cup T_2 \cup \ldots \cup T_n$;
- $<_E \supseteq <_1 \cup <_2 \cup \ldots \cup <_n$; ou seja, a ordenação do escalonamento estende a ordenação das operações internas a cada transação;
- para cada par de operações conflitantes p_i e q_j, $p_i <_E q_j$ ou $q_j <_E p_i$.

Um escalonamento preserva a ordem interna de cada transação e ordena mais algumas operações entre as transações, incluindo todas as operações conflitantes. Por fim, a definição de escalonamento:

Definição 2.3.4 Um *escalonamento (incompleto)* é um prefixo de um escalonamento completo.

Note que o conceito de escalonamento é independente do método usado para se gerar este escalonamento. Na prática, muitos escalonamentos possíveis são indesejáveis, e o controle de concorrência tem como função garantir que só os escalonamentos desejáveis sejam gerados pelo sistema.

Para ilustrar melhor as definições acima, considere as seguintes transações:

$$
\begin{array}{lclclclclcl}
T_1 & = & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 & & & & \\
T_2 & = & r_2[y] & \rightarrow & w_2[z] & \rightarrow & c_2 & & & & \\
T_3 & = & r_3[x] & \rightarrow & r_3[y] & \rightarrow & r_3[z] & \rightarrow & w_3[y] & \rightarrow & c_3
\end{array}
$$

Um escalonamento completo sobre T_1, T_2 e T_3 deve ordenar todas as operações conflitantes existents. Por exemplo, um tal escalonamento está ilustrado abaixo, onde $\rightarrow$ indica a ordenação dentro de um escalonamento:

$$
\begin{array}{cccccccccccc}
E_1 = & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 & & & & & & \\
 & & \searrow & & \searrow & & & & & & & \\
 & & & r_3[x] & \rightarrow & r_3[y] & \rightarrow & r_3[z] & \rightarrow & w_3[y] & \rightarrow & c_3 \\
 & & & & & & \searrow & & \searrow & & & \\
 & & & & & & & r_2[y] & \rightarrow & w_2[z] & \rightarrow & c_2
\end{array}
$$

Note que pelo escalonamento completo acima, não podemos afirmar qual das transações se confirmou primeiro, pois as operações de confirmação são todas concorrentes e não estão ordenadas (nem precisam estar) pela ordem do escalonamento E_1.

Por fim, um escalonamento E_1' das transações seria:

$$
\begin{array}{cccccccccc}
E_1' = & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 & & & & \\
 & & \searrow & & \searrow & & & & & \\
 & & & r_3[x] & \rightarrow & r_3[y] & \rightarrow & r_3[z] & & \\
 & & & & & & \searrow & & \searrow & \\
 & & & & & & & r_2[y] & \rightarrow & w_2[z]
\end{array}
$$

Note que o escalonamento E_1' é um prefixo do escalonamento completo E_1. Como havíamos mencionado, uma transação está *ativa* se ainda não se confirmou ou abortou, No escalonamento E_1' acima, tanto T_2 quanto T_3 estão ativas. A *projeção confirmada* de um escalonamento E, $C(E)$, é a restrição de E contendo apenas transações confirmadas; ou seja, $C(E)$ é obtido de E apagando-se todas as transações abortadas e ativas. No escalonamento E_1' acima, $C(E_1')$ contém apenas T_1.

2.3.3 Escalonamentos Seriáveis

Um escalonamento completo E é dito *serial* se para cada par de transações T_i e T_j em E, todas as operações de T_i precedem todas as operações de T_j em E, ou vice-versa.

Por exemplo, considere as duas transações T_1 e T_2, uma depositando $100,00 numa conta e outra depositando um bônus de 10% nesta mesma conta que, antes da execução das transações, continha $1000,00:

T_1	T_2
`s := Read( SALDO )`	`s := Read( SALDO )`
`s := s + 100`	`s := s + 0,1*s`
`Write( SALDO, s )`	`Write( SALDO, s )`
$\texttt{COMMIT}_1$	$\texttt{COMMIT}_2$

Existem duas execuções seriais possíveis destas duas transações. Se forem executadas na ordem $T_1; T_2$ o resultado o saldo final será de $1210,00; mas se forem executadas na ordem $T_2; T_2$ o saldo final será de $1200,00.

Escalonamentos seriais são ineficientes e indesejáveis, pois não permitem nenhuma concorrência entre operações de transações distintas, e, por conseqüência, a execução de uma transação bloqueia a execução de todas as outras.

No entanto, escalonamentos seriais são nosso parâmetro para a determinação da correção de escalonamentos. Pois um escalonamento é correto apenas se for *equivalente* a um escalonamento serial. Por exemplo, poderíamos ter os seguintes escalonamentos concorrentes da transações acima:

T_1	T_2	T_1	T_2
`s := r(SALDO)`	`s := r(SALDO)`	`s := r(SALDO)`	
`s := s + 100`	`s := s + 0,1*s`	`s := s + 100`	
`w(SALDO,s)`		`w(SALDO,s)`	
$\texttt{COMMIT}_1$			`s := r(SALDO)`
	`w(SALDO,s)`		`s := s + 0,1*s`
	$\texttt{COMMIT}_2$	$\texttt{COMMIT}_1$	
			`w(SALDO,s)`
			$\texttt{COMMIT}_2$

No escalonamento da esquerda, o saldo final será de $1100,00 pois a atualização de T_2 estará destruindo a atualização feita por T_1; tal escalonamento deve ser evitado pois viola os princípios de durabilidade e isolamento. Por outro lado, o escalonamento da direita dá como resultado um saldo de $1210,00, equivalente a uma execução serial de $T_1; T_2$; desta forma, o escalonamento da direita satisfaz nosso critério de correção.

Para podermos definir precisamente este critério, precisamos ainda tornar preciso a noção de equivalência de escalonamentos.

Definição 2.3.5 Dois escalonamento E e E' são *equivalentes*, $E \equiv E'$, se:

- tanto E quanto E' são definidos sobre o mesmo conjunto de transações e sobre as mesmas operações;
- ambos ordenam operações conflitantes de transações não abortadas (confirmadas ou ativas) igualmente. Ou seja, se p_i e q_j são operações conflitantes e $a_i \notin T_i, a_j \notin T_j$:

$$p_i <_E q_j \iff p_i <_{E'} q_j.$$

Por exemplo, considere os escalonamentos abaixo:

$$\begin{array}{lllllllllllllll}
E_2 & = & w_1[y] & \rightarrow & r_2[y] & \rightarrow & w_1[x] & \rightarrow & r_2[x] & \rightarrow & c_1 & \rightarrow & w_2[z] & \rightarrow & c_2 \\
E_3 & = & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 & \rightarrow & r_2[x] & \rightarrow & r_2[y] & \rightarrow & w_2[z] & \rightarrow & c_2 \\
E_4 & = & r_2[y] & \rightarrow & w_1[x] & \rightarrow & w_1[y] & \rightarrow & r_2[x] & \rightarrow & c_1 & \rightarrow & w_2[z] & \rightarrow & c_2
\end{array}$$

Temos que $E_2 \equiv E_3$ pois ordenam os dois conflitos $w_1[y] < r_2[y]$ e $w_1[x] < r_2[x]$ igualmente. Já E_4 não é equivalente a nenhum dos dois pois ordena $r_2[y] <_{E_4} w_1[y]$.

Note que E_3 é uma transação serial, onde todas as operações de T_1 precedem as operações de T_2. Além disso, tanto E_2 como E_3 são transações completas confirmadas. Neste caso E_2 é dito uma transação *seriável*, pois $E_2 \equiv E_3$.

Definição 2.3.6 Um escalonamento E é dito *seriável* se a projeção confirmada de E é equivalente a um escalonamento serial. Ou seja, se existe um escalonamento serial E_s tal que $C(E) \equiv E_s$.

A definição acima utiliza a projeção confirmada de um escalonamento pois, para o critério de correção de um escalonamento, apenas interessa aquelas transações que o escalonador deve garantir a todo custo a durabilidade de sua execução.

No exemplo acima, apesar de E_4 não ser equivalente a E_2, pode ser que exista um outro escalonamento serial equivalente a E_4. Será que tal escalonamento existe mesmo?

2.3.4 A Condição de Seriação

Felizmente existe uma forma bem fácil de se determinar se um dado escalonamento E é seriável ou não, que depende da construção do *grafo de seriação* para o escalonamento $GS(E)$. O grafo $GS(E)$ é um grafo dirigido construído da seguinte maneira:

- os nós de $GS(E)$ são as transações confirmadas de E;
- para cada par de operações conflitantes p_i e q_j tal que $p_i <_E q_j$, adicionamos a aresta $T_i \rightarrow T_j$ a $GS(E)$.

Pode haver mais de um par de operações conflitantes que gerem a aresta $T_i \rightarrow T_j$, mas neste caso adicionamos apenas uma aresta ao grafo $GS(E)$.

Por exemplo, se considerarmos o escalonamentos E_1, E_2 e E_4 dos exemplos apresentados até aqui, obtemos o grafos ilustrados na Figura 2.5.

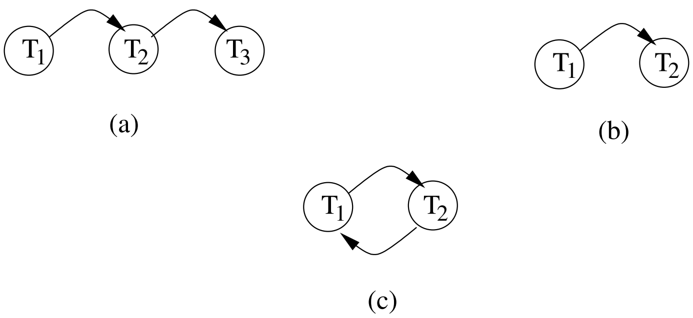


Figura 2.5: Grafos de Seriação: (a) $GS(E_1)$, (b) $GS(E_2)$, (c) $GS(E_4)$

A partir do grafo de seriação, podemos facilmente determinar se um escalonamento é seriável.

Teorema 2.3.7 (Condição de Seriação) *Um escalonamento E é seriável se e somente se $GS(E)$ é acíclico.*

Prova: Vamos supor que as transações confirmadas de E são $T_1, T_2, \ldots, T_n$.

($\Leftarrow$) Vamos supor que $GS(E)$ é acíclico, e neste caso temos que construir um escalonamento serial equivalente a E. Como $GS(E)$ é acíclico, podemos ordená-lo topologicamente (ver Seção 2.3.1). Seja E_s um escalonamento serial obtido a partir de uma ordenação topológica de $GS(E)$: $T_{i_1}; T_{i_2}; \ldots; T_{i_n}$. Vamos mostrar que $E \equiv E_s$. Ambos são definidos sobre o mesmo conjunto de transações. Considere um par de operações conflitantes p_i e q_j. Se

$p_i <_E q_j$, então a aresta $T_i \rightarrow T_j \in GS(E)$ e, portanto, qualquer ordenação topológica de $GS(E)$ vai preservar esta ordem, ou seja $p_i <_{E_s} q_j$. Se, por outro lado, $p_i <_{E_s} q_j$, então necessariamente $T_i \rightarrow T_j \in GS(E)$. Como $GS(E)$ é acíclico, não pode haver uma aresta $T_j \rightarrow T_i \in GS(E)$. Logo, $p_i <_E q_j$ e provamos que $E \equiv E_s$, ou seja, E é seriável.

($\Rightarrow$) Vamos agora supor que E é seriável. Então existe um escalonamento serial E_s, $E \equiv E_s$. Considere a aresta $T_i \rightarrow T_j \in GS(E)$. Logo deve haver um par de operações conflitantes $p_i <_E q_j$. Como E_s é serial e preserva a ordenação de conflitos, então devemos ter que T_i precede T_j em E_s sempre que $T_i \rightarrow T_j \in GS(E)$.

Vamos supor que haja um ciclo em $GS(E)$: $T_{i_1} \rightarrow T_{i_2} \rightarrow \ldots \rightarrow T_{i_k} \rightarrow T_{i_1}$. Pelo que vimos acima, em E_s devemos ter que T_{i_1} precede T_{i_2} precede $\ldots T_{i_k}$, que por sua vez também precede T_{i_1}, o que é uma contradição, pois T_{i_1} só ocorre uma vez em E_s. Logo $GS(E)$ deve ser acíclico, encerrando a prova. $\square$

Com base na condição de seriação e nos grafos de seriação construídos acima, vemos que os escalonamentos E_1 e E_2 são seriáveis, mas o escalonamento E_4 não. Ou seja, não existe nenhum escalonamento serial equivalente a E_4.

A prova do teorema acima é esclarecedora no seguinte sentido. Nela, podemos ver que se $GS(E)$ é acíclico, basta tomar uma ordenação topológica de $GS(E)$ para obtermos um escalonamento serial. Como pode haver mais de uma ordenação topológica de um mesmo grafo acíclico, podemos ter mais de um escalonamento serial equivalente a E. A definição de escalonamento seriável requer apenas que *exista* um escalonamento seriável equivalente, não importando *qual* seja este escalonamento.

A noção de seriabilidade é independente das propriedades de escalonamentos que vimos na Seção 2.2.2. Ou seja escalonamentos seriáveis podem ser estritos, ou apenas livres de cascateamento de abortos, ou apenas recuperáveis, ou até não-recuperáveis. Similarmente, podemos ter escalonamentos não seriáveis com todas estas propriedades. Escalonamentos seriais são sempre seriáveis e estritos, mas *existem (muitos) escalonamentos seriáveis e estritos que não são seriais.*

Deixamos como exercício ao leitor elaborar 8 escalonamentos diferentes, cada um com uma das propriedades acima mencionadas.

2.4 Controle de Concorrência por Bloqueios Bifásicos

Já sabemos que para um escalonamento ser aceitável ele deve ser seriável, e temos uma simples caracterização de escalonamentos seriáveis em termos de grafos de seriação acíclicos.

Resta agora apresentar um método de controle de concorrência que garanta que todos os escalonamentos serão sempre seriáveis (e, de preferência, estritos também).

O método mais popular e o mais utilizado nos gerenciadores de bancos de dados disponíveis no mercado é o *Controle de Concorrência por Bloqueios Bifásicos*[4], abreviado por BBF. Este método foi proposto inicialmente em [EGLT76], e provas de sua correção foram apresentadas em [BSW79, Pap79]. Nossa apresentação aqui segue a de [BHG87].

2.4.1 Bloqueios

Bloqueios são um mecanismo antigo de controle de acessos a recursos compartilhados, e forma utilizados na área de sistemas operacionais desde os primóridios da concorrência; ver, por exemplo [BH73].

A idéia básica dos bloqueios é muito simples. Suponhamos que exista um recurso compartilhado no sistema; por exemplo, uma impressora, uma área de memória comum, ou um dispositivo de disco. Se um processo quiser acessar este recurso comum, ele deve antes requisitar um bloqueio; se o recurso estiver livre, então o processo recebe o direito de usar o recurso e este recurso fica bloqueado para todos os outros processos do sistema, até que o processo em questão libere o bloqueio. Se um processo requisita um bloqueio de um recurso bloqueado, então o processo entra numa fila de espera; quando o recurso for liberado, um dos processos que estão na fila receberá o bloqueio. Este tipo de bloqueio é chamado de *bloqueio exclusivo*.

Além do bloqueio exclusivo, existe o *bloqueio compartilhado*, onde vários processos podem obter simultaneamente um bloqueio para, por exemplo, ler uma posição de memória compartilhada. No entanto, todos os processos (inclusive os detentores do bloqueio compartilhado de leitura) estão bloqueados para escrever nesta posição de memória até que o último bloqueio compartilhado de leitura seja liberado.

No caso de bancos de dados, utiliza-se um bloqueios compartilhados para a leitura (*read-lock*) de um item de dados x, e bloqueios exclusivos para a escrita (*write-lock*) de um item de dados x; No entanto, o mero uso de

[4] Em inglês: two-phase locking, abreviado por 2PL.

bloqueios de leitura e escrita não são suficientes para garantir a seriação de um escalonamento, como demonstra o escalonamento abaixo:

T_1	T_2
`read-lock(SALDO)`	
`s := r(SALDO)`	
`read-unlock(SALDO)`	
`s := s + 100`	`read-lock(SALDO)`
	`s := r(SALDO)`
	`read-unlock(SALDO)`
`write-lock(SALDO)`	`s := s + 0,1*s`
`w(SALDO,s)`	
`COMMIT`$_1$	
`write-unlock(SALDO)`	
`write-lock(SALDO)`	
	`w(SALDO,s)`
	`COMMIT`$_2$
	`write-unlock(SALDO)`

Neste caso, mesmo com o uso de bloqueios, a operação de escrita feira por T_1 é descartada pela pela operação de escrita de T_2 e este escalonamento não é equivalente a nenhuma execução serial de T_1 e T_2. Um outro mecanismo (também chamado de protocolo) se faz necessário.

2.4.2 O Funcionamento Básico do Método BBF

No caso do controle de concorrência de transações acessando um banco de dados, existe um tipo de bloqueio do dado x para cada operação realizada sobre x. Ou seja, temos o bloqueio $rl_i[x]$, que deve ser obtido antes de cada operação de leitura $r_i[x]$, e temos o bloqueio de escrita $wl_i[x]$, que deve ser obtido antes de cada operação de escrita $w_i[x]$[5]. Similarmente, temos as operações de desbloqueio $ru_i[x]$ e $wu_i[x]$. Para uma dada transação T_i e um item de dado x, temos sempre que:

$$rl_i[x] <_i r_i[x] <_i ru_i[x] \qquad wl_i[x] <_i w_i[x] <_i wu_i[x] \tag{2.1}$$

Dois bloqueios de transações distintas conflitam se as operações correspondentes conflitam. Ou seja, se p_i e q_j são operações conflitantes, os bloqueios pl_i e ql_j são *bloqueios conflitantes*. Duas transações não podem obter bloqueios conflitantes para um mesmo dado, e desta forma:

[5] *rl*, *wl*, *ru* e *wu* são abreviações de *read lock*, *write lock*, *read unlock* e *write unlock*, respectivamente.

- Dois bloqueios de leitura no mesmo dado, $rl_i[x]$ e $rl_j[x]$ não são conflitantes. Desta forma, um dado pode ser lido por mais de uma transação concorrentemente, e por este motivo o bloqueio de leitura é chamado de *bloqueio compartilhado.*

- Se uma transação T_i obtém um bloqueio de escrita, $wl_i[x]$, então nenhuma outra transação T_j poderá obter um bloqueio de escrita ou leitura para x até que a transação T_i seja desbloqueada. Por isso, o bloqueio de escrita é chamado de *bloqueio exclusivo.*

Num dado escalonamento E, se p_i e q_j são operações conflitantes, então elas estão ordenadas: ou $p_i <_E q_j$ ou $q_j <_E p_i$. No entanto, para que a segunda operação possa ser executada, é preciso que a primeira tenha sido desbloqueada e a segunda bloqueada, nesta ordem. Ou seja, se p_i e q_j são operações conflitantes, então

$$p_i <_E q_j \Longrightarrow p_i <_E pu_i <_E ql_j <_E q_j \qquad (2.2)$$

Bloqueios de leitura podem ser *promovidos* para bloqueios de escrita da seguinte maneira. Suponha que uma transação T_i possua um bloqueio de leitura no dado x, e que T_i solicite um bloqueio de escrita para x. Se nenhuma outra transação possui um outro bloqueio de leitura para x, este bloqueio pode ser *promovido* para um bloqueio exclusivo de escrita. No entanto, se alguma outra transação estiver simultaneamente compartilhando deste bloqueio de leitura com T_i, o bloqueio de T_i não poderá ser promovido. Assim sendo, T_i será posta em espera, aguardando a liberação de todos os bloqueios de leitura de outras transações sobre x. Somente então o bloqueio de leitura de T_i poderá ser promovido a um bloqueio de escrita.

Na arquitetura genérica de um sistema de bancos de dados ilustrada na Figura 2.3, vimos que o Escalonador recebe operações de leitura, escrita, confirmação e abortos do Gerenciador de Transações e as repassa, possivelmente em ordem diferente para o Gerenciador de Dados. Com base no que vimos até aqui, temos que um Escalonador que opera pelo método BBF tem como tarefas:

1. Ao receber uma operação $p_i[x]$, o Escalonador deve verificar se o bloqueio $pl_i[x]$ não está em conflito com algum $ql_j[x]$ em atividade. Se conflitar, o Escalonador coloca a transação T_i *em espera*, até que o bloqueio necessário seja liberado; caso contrário, o Escalonador anota que $pl_i[x]$ está ativo (escrevendo a tripla $\langle x, p, T_i \rangle$ numa *tabela de bloqueios*).

2. Uma vez que o bloqueio $pl_i[x]$ está ativo, ele só poderá ser desbloqueado após o Gerenciador de Dados informar que a operação $p_i[x]$ foi completada.

3. Existe ainda uma terceira condição que deve ser mantida pelo Escalonador, conhecida como *Condição das Duas Fases*:

   > Uma vez que o Escalonador desbloqueou um dado qualquer de uma transação, esta transação não poderá mais obter *nenhum* novo bloqueio, para nenhum item de dado.

   Desta forma, uma transação fica dividida em duas fases: a fase de obtenção de bloqueios e a fase de liberação dos mesmos. Ou seja, para quaisquer item de dados x e y, e quaisquer operações p e q:

$$pu_i[x] \not<_i ql_i[y] \tag{2.3}$$

É esta terceira condição que dá nome ao método. Ela permite a liberação de *qualquer bloqueio* apenas quando nenhuma nova solicitação de bloqueio for ocorrer. Tal restrição não é gratuita. O seguinte exemplo mostra que uma pequena violação neste requisito pode levar a um escalonamento não seriável. Considere as transações:

$$\begin{array}{lcl} T_1 & : & r_1[x] \;\; r_1[y] \;\; c_1 \\ T_2 & : & w_2[x] \;\; w_2[y] \;\; c_2 \end{array}$$

e um possível escalonamento das mesmas, levando em consideração os bloqueios e os desbloqueios necessários para as operações, mas não obedecendo à Condição das Duas Fases:

$$\begin{array}{c|c} (rl_1[x]) & \\ r_1[x] & \\ (ru_1[x]) & \\ & (wl_2[x]) \\ & w_2[x] \\ & (wl_2[y]) \\ & w_2[y] \\ & c_2 \\ & (wu_2[x] \;\; wu_2[y]) \\ (rl_1[y]) & \\ r_1[y] & \end{array}$$

O grafo de seriação do escalonamento contém um ciclo entre T_1 e T_2 pois $r_1[x] < w_2[x]$ e $w_2[y] < r_1[y]$. Em T_1, tanto o bloqueio liberado $ru_1[x]$ quanto o bloqueio $rl_1[y]$ posteriormente obtido são bloqueios de leitura apenas. Se a Condição das Duas Fases fosse cumprida neste exemplo, o bloqueio $rl_1[y]$ não poderia ter sido solicitado após o desbloqueio $ru_1[x]$, e ao receber a solicitação de um bloqueio após um desbloqueio, a única ação possível para um escalonador BBF seria abortar a transação que violou a condição. Mesmo se substituíssemos estas operações por operações de escrita, permaneceria o ciclo no grafo de seriação. Isso demonstra que qualquer desobediência à Condição das Duas Fases pode levar a um escalonamento não seriável.

Por outro lado, temos o seguinte resultado, conhecido como *Teorema do Bloqueio Bifásico.*

Teorema 2.4.1 (Bloqueio Bifásico) *Escalonamentos gerados pelo método de bloqueios bifásicos são sempre seriáveis.*

Prova: Suponhamos que temos um escalonamento E obtido por um escalonador BBF. Tudo que temos que fazer é mostrar que o grafo da seriação $GS(E)$ é acíclico, pois então o Teorema 2.3.7 nos garante que E é seriável.

Suponhamos, por contradição, que $GS(E)$ possui um ciclo: $T_1, T_2, \ldots, T_k, T_1$. Como T_1 precede T_2 em $GS(E)$, existe um par de operações conflitantes $p_1^1 <_E q_2^1$. Então, por (2.2), temos que $p_1^1 <_E p^1u_1 <_E q^1l_2 <_E q_2^1$. Similarmente, o fato de T_2 preceder T_3 em $GS(E)$ nos fornece um par de operações conflitantes $p_2^2 <_E q_3^2$ tal que, por (2.2), temos que $p^2u_2 <_E q^2l_3$. Mas, de acordo com a Condição das Duas Fases, todos os bloqueios de T_{i_2} devem ter sido obtidos antes de qualquer desbloqueio e portanto $q^1l_2 <_E p^2u_2$; pela transitividade de $<_E$, obtemos $p^1u_1 <_E q^2l_3$.

Repetindo este raciocínio ao longo do ciclo, obtemos pares de conflitos $p_j^j <_E q_{j+1}^j$ tais que $p^1u_1 <_E q^jl_{j+1}$. Como a primeira e última transação no ciclo é T_1, isso implica em

$$p^1u_1 <_E q^nl_1$$

que é uma violação da Condição das Duas Fases, contradizendo a hipótese de que E foi gerado por um escalonador BBF. Desta forma, concluímos que não pode haver ciclos em $GS(E)$ e, portanto, E é seriável. □

Nesta prova, vemos que a Condição de Duas Fases é crucial para que escalonamentos BBF sejam seriáveis. E, como vimos no exemplo acima, uma transação só deve desbloquear um dado quando for absolutamente garantido que nenhum novo bloqueio será necessário. Mas como saber quando se chegou a esta situação?

2.4.3 BBF Estrito

O término de uma transação é o único ponto de seu processamento em que tem-se absoluta certeza que nenhum novo bloqueio será solicitado. Este término pode ser bem-sucedido (confirmação) ou mal-sucedido (aborto).

Um escalonamento BBF é chamado de *estrito* quando todos os bloqueios obtidos pela transação são liberados apenas após sua confirmação ou aborto.

A Figura 2.6 nos mostra o número de bloqueios obtidos por uma transação no método BBF. A linha tracejada indica a separação (arbitrária) entre a fase de aquisição de bloqueios e a fase de liberação dos mesmos.

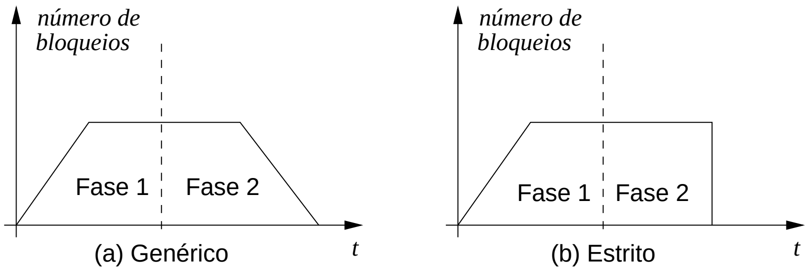


Figura 2.6: Número de bloqueios de uma transação ao longo do tempo

Na Figura 2.6(a) está ilustrado o comportamento do método BBF genérico, onde vemos que o número de bloqueios cresce monotonicamente com o tempo, até estabilizar, e em seguida decresce monotonicamente. Já no caso do método BBF estrito, a Figura 2.6(b) nos mostra que, a partir do início da transação, o número de bloqueios também cresce e atinge um patamar. A diferença está em que no caso estrito todos os bloqueio são liberados de uma única vez com o término da transação. O método estrito é, portanto, um caso particular do método genérico.

Se o Escalonador obedece ao método BBF estrito, quando uma transação T_i vai ler um dado, ela necessita obter um bloqueio de leitura para este dado. Como o escalonamento é estrito, a última transação T_j que escreveu sobre este dado necessariamente se confirmou antes de liberar seus bloqueios. Desta forma, T_i somente irá ler dados de transações confirmadas. Pelo mesmo raciocínio, vemos que T_i somente irá escrever sobre dados escritos por transações confirmadas.

Ou seja, o método BBF estrito gera escalonamentos estritos no sentido definido na Seção 2.2.2: uma transação só lê dados escritos por transações confirmadas e só escreve sobre dados escritos por transações confirmadas. Como mencionado em 2.2.2, isto traz o benefício de que os abortos podem ser processados restabelecendo-se as imagens prévias dos dados escritos pela transação abortada.

2.4.4 Travamentos

A utilização de bloqueios pode levar o sistema (ou uma parte dele) a uma situação de *travamento (deadlock)*, ou seja, a uma situação em que um conjunto de transações não pode prosseguir o processamento e ficam indefinidamente num estado de espera.[6]

O exemplo tradicional de travamento consiste na execução concorrente de duas transações T_i e T_j:

$$\begin{array}{rcl} T_i & = & w_i[x] \;\; w_i[y] \;\; c_i \\ T_j & = & w_j[y] \;\; w_j[x] \;\; c_j \end{array}$$

e um escalonamento E_{trava} que inicia executando concorrentemente $w_i[x]$ e $w_j[y]$. Portanto, T_i possui um bloqueio de escrita em x e está esperando um bloqueio de escrita em y. T_j, por sua vez, possui um bloqueio de escrita em y e está esperando por um bloqueio de escrita em x. Como uma transação está esperando pela outra, e nenhuma pode liberar seu bloqueio (pois isso violaria a Condição das Duas Fases), o escalonamento não pode progredir, ou seja, o escalonamento está travado.

O que o exemplo mostrou com duas transações pode ocorrer com mais de uma transação, onde cada transação está esperando por um recurso bloqueado pela seguinte, até que um ciclo de espera se fecha e todas as transações envolvidas ficam travadas. É bem possível que apenas uma parte das transações em execução no sistema estejam envolvidas no travamento; as demais permanecem em execução.

Um sistema travado, sem interferência externa, continuará travado para sempre. É tarefa do Escalonador BBF detectar a ocorrência de travamentos e abortar uma das transações envolvidas no travamento (a chamada *vítima*), liberando os bloqueios da vítima e, desta forma, destravando as transações envolvidas. Posteriormente, esta transação é reiniciada.

Para detectar a existência de um travamento, o Escalonador deve construir um *grafo de espera* de transações GE. Este grafo possui como nós as transações em execução. O grafo possui uma aresta dirigida $T_i \rightarrow T_j$ se a transação T_i está esperando por um bloqueio que pertence à transação T_j. Claramente, quando há um travamento no escalonamento o grafo da espera

[6]Este problema é típico de sistemas que usam bloqueios para controlar o acesso a recursos compartilhados, e não apenas do método BBF.

apresenta um ciclo envolvendo as transações travadas. A Figura 2.7 ilustra o grafo da espera associado ao exemplo acima.

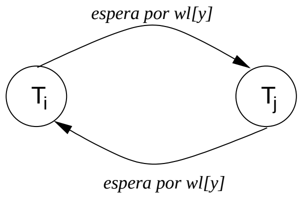


Figura 2.7: Grafo da Espera

De tempos em tempos, o Escalonador deve procurar por ciclos no grafo da espera a fim de identificar travamentos, escolher uma vítima, e abortá-la. Esta atividade não deve ser feita com freqüência alta demais, pois a busca de ciclos no grafo de espera pode sobrecarregar demais o processamento das transações. Por outro lado, se a detecção de travamentos for muito rara, é possível que as transações travadas permaneçam inativas por um tempo demasiado longo, diminuindo a eficiência do processamento de transações (medida como o tempo médio de processamento de transações no sistema).

Por outro lado, uma vez detectado um travamento, o sistema deve decidir qual das transações será abortada e reiniciada. Desta forma, existem várias estratégias para a escolha de uma vítima dentre as transações envolvidas no ciclo do grafo de espera:

- abortar a transação que estiver rodando a menos tempo;
- abortar a transação que realizou menor quantidade de modificações no banco de dados;
- abortar a transação que estiver participando de um número maior de ciclos (pois pode haver mais de um ciclo no grafo de espera).
- associar prioridades às transações e abortar a transação menos prioritária;
- uma combinação dos métodos acima.

Mas muito cuidado! Se o método de atribuição de prioridades for dinâmico, e se uma transação puder receber uma nova prioridade a cada reinício, podemos ter reinícios cíclicos. Por exemplo, suponha que temos travamento envolvendo duas transações; a primeira é abortada e reiniciada com prioridade mais alta. Em seguida, um novo travamento entre estas duas ocorre, e desta vez a segunda é abortada e reiniciada com prioridade ainda mais alta. E assim sucessivamente, as transações chegam a um travamento, são abortadas e reiniciadas com prioridades cada vez maiores. O ciclo se repetirá indefinidamente. Para evitar esta situação (chamada de *livelock*, ou de *reinícios cíclicos*), podemos proibir que uma transação seja reiniciada mais que um número fixo de vezes, ao final dos quais a transação é abortada sem reinício (ou reiniciada apenas após outra transação ter terminado).

É importante notar que a experiência mostra que, quando há um travamento, a maioria dos ciclos tem tamanho 2.

A detecção de travamentos através de ciclos em grafos foi proposta por [Hol72].

2.4.5 Aspectos de Implementação do BBF

Descreveremos aqui as estruturas de dados genéricas para a implementação do protocolo BBF. Basicamente o Escalonador possui duas estruturas de dados: a *tabela de bloqueios* e a *fila de espera*.

A tabela de bloqueios pode ser imaginada como um conjunto de triplas

$\langle$*Item de Dado, Id Transação, Tipo de Bloqueio* $\rangle$

onde assumimos que cada item de dados e cada transação possui um identificador único no sistema, e os tipos de bloqueio são: leitura (compartilhado) e escrita (exclusivo).

A fila de espera contém as transações que estão paradas à espera da liberação de um bloqueio para um determinado dado. À cabeça da fila está o item de dado e, em seguida, uma seqüência de pares $\langle$*Id Transação, Tipo*$\rangle$. Cada par representa uma transação que está esperando por um determinado tipo de bloqueio para o referido item de dados.

De acordo com a arquitetura genérica da Figura 2.3, o Escalonador pode receber quatro tipos de comandos do Gerenciador de Transações: escrita, leitura, confirmação e aborto. Ao receber um aborto ou confirmação, o Escalonador deve:

- Apagar todas as linhas da tabela de bloqueios que se refiram a transação terminada.
- Eliminar a transação terminada de todas as filas de espera.

Estas são atividades "por atacado", ou seja, várias linhas da tabela de bloqueios e filas de espera. Já o processamento das operações de leitura e escrita são mais detalhadas, e consistem em:

- Ao receber uma operação de leitura ou escrita, a tabela de bloqueios é vasculhada, para ver se a transação já possui a operação correspondente. Se possuir, a operação é passada ao Gerenciador de Dados. Ao receber a resposta do Gerenciador de Dados (dado lido, ou escrita realizada), a mesma é passada ao Gerenciado de Transações.

- Se a transação não possuir o bloqueio correspondente, o Escalonador deve procurar na tabela de bloqueios se nenhuma outra transação possui um bloqueio conflitante. Se não houver conflito, o bloqueio é inserido na tabela de bloqueios (ou promovido de bloqueio de escrita para bloqueio de leitura), e a operação é passada ao Gerenciador de Dados.

- Se houver conflito, o par ⟨*Id Transação, Tipo*⟩ é inserido na fila de espera correspondente.

A operação de desbloqueio provoca a remoção de uma linha da tabela de bloqueios. No caso de escalonadores estritos, esta remoção se faz apenas "por atacado" ao término da transação; em outros sistemas, um pedido de desbloqueio pode vir do Gerenciador de Transações. Ao apagar uma linha da tabela de bloqueios, o Escalonador deve verificar se há fila de espera para aquele item de dados. Caso haja mais de uma transação na fila, o Escalonador precisa escolher a qual transação fornecer o bloqueio, removê-la da fila, inserir o bloqueio na tabela de bloqueio e passar a operação ao Gerenciador de Dados. Novamente, existem uma série de políticas para escolher uma transação da fila de espera (ou mais de uma, no caso de bloqueios compartilhados) e, na prática, deve-se fazer um compromisso entre as diversas políticas possíveis:

- Escolher a primeira transação da fila (que é a transação que espera a mais tempo).

- Desbloquear o maior número possível de transações, no caso de bloqueios de leitura. Note que neste caso, transações à espera de bloqueios de leitura podem "passar na frente" de transações à espera de bloqueios de escrita, e isso pode levar a uma espera eterna pelo bloqueio de escrita. Para evitar tal fato, pode-se fornecer bloqueios de leitura a todas as transações na fila, até encontrar-se uma esperando por um bloqueio de escrita. Ou seja, a ordem na fila é obedecida.

- Transações de alta prioridade são inseridas no início da fila. Isto também pode levar a espera eterna de transações de baixa prioridade.

É importante notar que a manipulação eficiente destas estruturas de dados é crítica para a eficiência global do sistema, uma vez que estas manipulações feitas pelo Escalonador são muito freqüentes e podem levar à lentidão total do sistema. O código do escalonador necessita ser altamente otimizado. Na prática, cada uma das atividades aqui descritas não deve ocupar mais que uma centena de instruções de máquina para não sobrecarregar o sistema.

Uma das primeiras e mais influente descrição detalhada da implementação do protocolo BBF é encontrada em [Gra78, CM84].

2.5 Controle de Concorrência Distribuído

Até agora vimos apenas o controle de concorrência para transações centralizadas, ou seja, todos os dados estão controlados por um único escalonador. Vamos agora considerar que temos diversas máquinas distribuídas em uma rede, chamadas de nós do sistema, cada uma contendo o seu sistema de gerenciamento de dados. Em especial, cada nó possui o seu escalonador, conforme ilustrado na Figura 2.8.

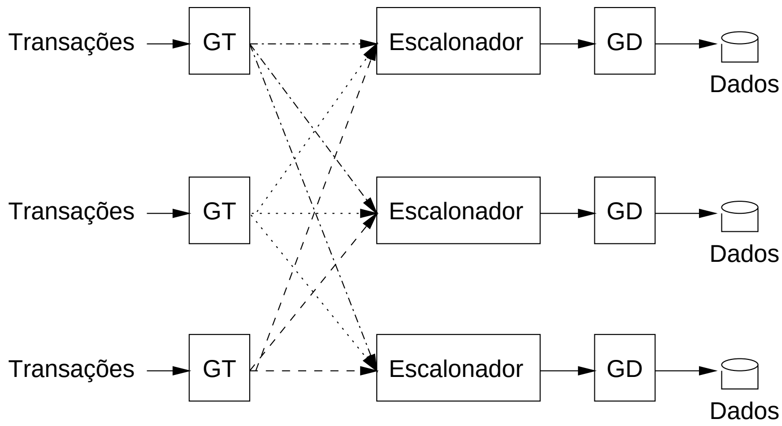


Figura 2.8: Arquitetura de um SGBD distribuído

Do ponto de vista de uma aplicação, uma transação distribuída é executada da mesma forma que uma transação local, ou seja, submetendo as operações ao um gerenciador de transações de um servidor de banco de dados. A cada operação de escrita ou leitura submetida, o gerenciador de transações deve decidir se o dado é local — e enviar a operação ao escalonador local — ou se o dado é remoto — e enviar a operação ao banco de dados remoto que pode executá-la, submetendo-a ao seu escalonador. Em cada nó que uma transação global visitar, ela será executada como uma transação local composta das operações globais que lhe forem enviadas. A *transação distribuída* é a união de diversas transações locais.

Desta forma, temos mais de um nó participando em uma transação distribuída. O conjunto dos escalonadores nos nós da rede constituem um *escalonador distribuído*. São dois os assuntos que vamos abordar:

- Como adaptar o protocolo BBF para o caso distribuído.
- Como garantir que, uma transação sendo executada em diversos nós do sistema tem uma *terminação atômica*, ou seja, como garantir que a transação ou se confirma em todos os nós ou aborta em todos eles. Para isso, vamos investigar os *protocolos de confirmação atômica*.

2.5.1 BBF Distribuído

Vamos assumir que cada escalonador num sistema de bancos de dados distribuído obedece homogeneamente ao protocolo BBF. Desta forma, cada escalonador mantém sua própria tabela de bloqueios, e uma transação distribuída pode ter bloqueios em diversos nós do sistema, cada um dos quais localmente obedecendo ao BBF.

A observância do BBF local, no entanto, não garante a observância do BBF globalmente. Podemos ter uma transação liberando bloqueios num nó N_1 da rede enquanto num nó N_2 outros bloqueios ainda estão sendo obtidos. Isto pode levar a escalonamentos não seriais de transações distribuídas, pelo mesmo motivo que uma violação da condição das duas fases localmente pode levar a escalonamentos não seriáveis. Necessitamos, então, de uma *Condição Distribuída das Duas Fases*:

> Uma vez que o Escalonador desbloqueou um dado qualquer de uma transação *em algum nó da rede*, esta transação distribuída não poderá mais obter nenhum novo bloqueio *em nenhum outro nó da rede.*

Para implementarmos o BBF distribuído sem um grande aumento de mensagens na rede (de um nó para todos os outros participantes em uma transação distribuída, avisando que não tem mais bloqueios para solicitar), vamos assumir que todos os escalonadores locais implementam o BBF estrito. Desta forma, para garantir a observância da Condição Distribuída das

Duas Fases, basta *garantir que os nós participantes da transação só receberão a solicitação de confirmação após todos os nós terem executado todas as suas operações*; como garantir tal comportamento será visto em 2.5.2.

Para verificar a correção do BBF distribuído, basta notar que um escalonamento produzido por um sistema que respeita BBF localmente e a Condição Distribuída das Duas Fases é indistinguível de um escalonamento BBF local, e portanto só pode gerar escalonamentos seriáveis. Além disso, se cada BBF local é estrito, então o BBF distribuído também o é. Para uma análise detalhada do controle de concorrência distribuído, ver [Pap86, BHG87].

Travamentos Distribuídos

Além de travamentos locais, podemos agora ter também os travamentos globais. O exemplo clássico é o seguinte. Suponha que o dado x reside no nó N_1 e y reside no nó N_2. Suponha que temos as transações:

$$T_i = r_i[x]; w_i[y]; c_i \qquad T_j = r_j[y]; w_j[x]; c_j$$

Vamos supor que a leitura inicial de ambas transações realizam a operação de leitura concorrentemente e solicitam a operação de escrita. Mas T_i possui um bloqueio de leitura para x em N_1 e T_j possui um bloqueio de leitura para y em N_2. Portanto, nenhuma das solicitações de escrita será atendida até que uma das duas transações aborte e libere seus bloqueios. No entanto, tanto no escalonador de N_1 quanto no de N_2, os grafos de espera são acíclicos (ver Seção 2.4.4). Em N_1 temos $T_i \to T_j$ e em N_2 temos $T_j \to T_i$.

É evidente que, para se detectar um travamento distribuído, necessitamos de um *grafo de espera global*. Existem dois métodos básicos para a detecção de travamentos distribuídos: centralizados e distribuídos.

Detecção centralizada. Uma única máquina na rede é designada para a construção do grafo da espera distribuído. De tempos em tempos, cada um dos escalonadores enviam seus grafos locais de espera para o detector global. Este, então, deve localizar os travamentos distribuídos, escolher uma vítima e reiniciá-la. Para a escolha da vítima, o detector necessita de informações locais sobre as transações (por exemplo, início de execução, prioridade, número de operações já realizadas, etc), o que aumenta consideravelmente o trafego de dados na rede, uma vez que estas informações são repetidamente transmitidas.

Detecção distribuída. Neste caso, cada um dos escalonadores possui um grafo parcial de espera global e pode detectar alguns ciclos e abortar uma vítima. Uma das formas de se realizar esta detecção distribuída é organizar os escalonadores de forma circular. De tempos em tempos, cada escalonador envia o seu grafo de espera global para o escalonador seguinte na ordem; este grafo da espera contém informações locais e informações obtidas de seu antecessor. Como travamentos não desaparecem sozinhos, se houver um travamento global ele será detectado ao fim de uma volta completa no circuito de escalonadores. Cada nó apenas precisa se preocupar em desfazer os ciclos que envolvem transações daquele nó. Para aumentar a rapidez na detecção distribuída, um nó pode mandar seu grafo para mais de um vizinho.

A detecção distribuída gera uma distribuição de mensagens mais homogênea na rede, em vez de sobrecarregar um único nó. Além disso, como os ciclos de travamento possuem, na maioria dos casos, comprimento 2, ciclos no grafo global da espera serão rapidamente detectados. A desvantagem do método distribuído é que todos os escalonadores precisam ser alterados para colaborar na detecção de ciclos.

No entanto, tanto o método centralizado quanto o distribuído sofrem de um sério problema: *travamentos fantasmas*. Como há atrasos tanto na comunicação quanto no intervalo de tempo em que os escalonadores locais enviam informações, pode ocorrer o caso em que uma transação já tenha sido terminada mas permaneça no grafo de detecção.

Esta terminação só pode ser por aborto, uma vez que uma transação travada não pode prosseguir para terminar com sucesso. Pode ocorrer, no entanto, que uma transação travada seja abortada (por exemplo, por estar envolvida em outro travamento local), mas o grafo de espera global estar desatualizado. Neste caso, teremos um *travamento fantasma*. O escalonador que detectar tal travamento tentará escolher uma vítima e abortá-la. Se a vítima for a transação já terminada, nada ocorre. Mas se a vítima for uma outra transação, este terá sido um aborto desnecessário.

Uma forma de diminuir a ocorrência de abortos fantasmas (mas sem ser capaz de evitá-los na totalidade) é aumentar a freqüência com que as informações locais sobre o grafo de espera é passada para a rede; isto, obviamente, aumenta o tráfego na rede, o que é indesejável em excesso. Portanto, a freqüência com que os escalonadores enviam informações locais deve ser calibrada para nem causar um congestionamento na rede, nem criar um número exagerado de travamentos fantasmas.

2.5.2 O Protocolo de Confirmação Atômica

Um dos problemas mais sérios em um banco de dados distribuídos é garantir que uma transação global atinja a mesma terminação em todos os nós participantes. Ou seja, se uma transação distribuída está sendo executada nos nós $N_1, \ldots, N_k$, deseja-se garantir que em todos os k nós da rede, as componentes locais todas abortam ou todas se confirmam.

Um protocolo que garanta tal comportamento é chamado de *protocolo de terminação atômica*. Existem dois protocolos básicos para garantir a terminação atômica: o *protocolo de confirmação bifásica* e o *protocolo de terminação trifásica*.

Para a compreensão do controle de concorrência em sistemas distribuídos, basta sabermos que um tal protocolo existe, e que este garante que, apesar das falhas, a terminação de uma transação será consistente em todos os nós da rede envolvidos.

O funcionamento destes protocolos está intimamente ligado à recuperação de falhas. Portanto, a discussão sobre os mesmos será adiada até a seção 3.6, onde apresentaremos o protocolo de confirmação bifásica.

2.6 Controle de Concorrência por Ordenação de Marcas de Tempo

Nesta seção, vamos examinar um mecanismo de controle de concorrência que *não* se utiliza de bloqueios para garantir a seriação das transações. Este protocolo é devido a [SLR76].

O *controle de concorrência por marcas de tempo* (em inglês: *timestamps*) atribui uma marca de tempo a cada transação T_i, $mt(T_i)$. As marcas de tempo são valores totalmente ordenados atribuídas em ordem crescente às transações. Ou seja, se $T_i \neq T_j$, então ou $mt(T_i) < mt(T_j)$ ou $mt(T_j) < mt(T_i)$.

Normalmente, é o gerenciador de transações que atribui as marcas de tempo. Em um sistema centralizado, isto pode ser facilmente gerado usando-se um contador; este contador pode ser o próprio relógio do sistema, o que garante a ordenação das transações. Em sistemas distribuídos, a ordem total pode ser obtida da seguinte maneira: cada gerenciador de transações recebe um número único e mantém um contador; a marca de tempo é composta por duas partes: primeiro o contador, seguido pelo número do gerenciador de transações. Desta forma, duas marcas de tempo, geradas por gerenciadores diferentes sempre serão diferentes, e serão sempre totalmente ordenados.

A principal característica da ordenação por marcas de tempo é a seguinte:

> *O controle de concorrência por marcas de tempo escalona as transações de acordo com a ordem de suas marcas de tempo.*

É por isso que este método é muitas vezes chamado de controle de concorrência por *pré-ordenação*. Desta forma, o controle de concorrência por marcas de tempo (MT) funciona de forma oposta ao BBF: o BBF serializa as transações ordenandas pelo instante de confirmação, enquanto que o MT serializa as transações ordenandas pelo instante de seu início.

Esta propriedade básica é obtida garantindo-se a seguinte regra, conhecida como regra MT:

> Se $p_i[x]$ e $q_j[x]$ são duas operações conflitantes, então $p_i[x]$ será executanda antes de $q_j[x]$ se e somente se $mt(T_i) < mt(T_j)$.

Vemos que a regra MT, por si só, garante a seriação das transações pois, no grafo de seriação, sempre que houver uma aresta de T_i para T_j a regra MT acima garante que $mt(T_i) < mt(T_j)$. Como as marcas de tempo são totalmente ordenadas, isso implica que nunca poderá haver um ciclo no grafo da seriação, e o escalonamento será sempre seriável.

A seguir, veremos formas de garantir que a regra MT seja respeitada.

2.6.1 Funcionamento Básico

O controle de concorrência por MTs possui o seguinte funcionamento básico. A idéia principal é a seguinte. Dizemos que uma operação $p_i[x]$ *chegou tarde demais* se o escalonador já executou uma operação conflitante $q_j[x]$ com $mt(T_i) < mt(T_j)$. Neste caso, se $p_i[x]$ for executada, estaremos violando a regra MT. Só nos resta como opção abortarmos T_i e a reiniciarmos com uma nova marca de tempo $mt'(T_i) > mt(T_j)$.

Para obtermos tal funcionamento, cada item de dados x no banco de dados recebe dois valores, que são alterados com o tempo:

- $r\text{-}mt(x)$: valor da marca de tempo da última transação a ler x.
- $w\text{-}mt(x)$: valor da marca de tempo da última transação a escrever sobre x.

O funcionamento básico do protocolo MT é o seguinte:

- Para executar uma operação de leitura $r_i[x]$, verificar o valor de $w\text{-}mt(x)$.

  - Se $w\text{-}mt(x) < mt(T_i)$, execute a leitura e faça $r\text{-}mt(x) := mt(T_i)$.
  - Caso contrário $w\text{-}mt(x) > mt(T_i)$ e T_i deve ser abortada e reiniciada com uma nova marca de tempo $mt'(T_i) > w\text{-}mt(x)$.

- Para executar uma operação de escrita $w_i[x]$, verificar o valor de $r\text{-}mt(x)$ e $w\text{-}mt(x)$.
  - Se $mt(T_i)$ é maior que ambos, então execute a operação de escrita e faça $w\text{-}mt(x) := mt(T_i)$.
  - Caso contrário, rejeite a operação de escrita, aborte T_i e a reinicie com uma nova marca de tempo maior que $r\text{-}mt(x)$ e $w\text{-}mt(x)$.

Note que este método nunca chega a gerar um travamento. Considere um exemplo de um escalonamento que geraria um travamento sob o protocolo BBF:

$$T_i = r_i[x]; w_i[y]; c_i$$
$$T_j = r_j[y]; w_j[x]; c_j$$
$$E = r_j[y]; r_i[x]; ?$$

Sob o protocolo MT este escalonamento não leva a um deadlock. Vamos examinar o que acontece no caso que $mt(T_i) < mt(T_j)$ e a próxima operação a ser executada é $w_i[y]$. No ponto em que o escalonamento parcial acima está parado, temos o seguinte:

$$r\text{-}mt(x) = mt(T_i)$$
$$r\text{-}mt(y) = mt(T_j)$$

Quando T_i submete $w_i[y]$, verifica-se que $r\text{-}mt(y) > mt(T_i)$ e a operação de escrita é rejeitada, levando ao aborto de T_i; T_i é reiniciada com $mt'(T_i) > mt(T_j)$. A próxima operação submetida é $w_j[x]$ que pode ser executada pois $r\text{-}mt(x) < mt(T_j)$, alterando-se o valor de $w\text{-}mt(x)$ para $mt(T_j)$. A operação de $r_i[x]$ será bem sucedida, pois $mt'(T_i) > mt(T_j)$. O escalonamento final será:

$$E = r_j[y]; r_i[x]; a_i; w_j[x]; r_i[x]; c_j; w_i[y]; c_i$$

Note que o protocolo MT não está livre de reinícios cíclicos (*livelock*), pois se a operação $r_i[x]$ fosse executada antes de $w_j[x]$, isto levaria a rejeição de $w_j[x]$ e ao aborto de T_j. Desta forma, para evitar reinícios cíclicos, é boa prática esperar para reiniciar a transação abortada pelo menos até o fim da transação que causou o seu aborto.

Note também que o escalonamento acima não é estrito, pois a transação T_i está lendo de uma transação não confirmada. O protocolo MT básico permite também que uma transação escreva sobre outra não confirmada. Desta forma, muito embora os escalonamentos gerados sejam sempre seriáveis, eles nem sempre são estritos ou recuperáveis.

2.6.2 MT Estrito

Para garantirmos que escalonamentos por MT são sempre estritos, não podemos permitir que haja leitura ou escrita de não confirmados. Para isso, precisamos alterar o funcionamento básico do protocolo MT.

Criamos duas filas de operações, *op-em-transito* e *op-em-espera*. Então alteramos o protocolo básico da seguinte maneira. Uma vez que uma operação $p_i[x]$ foi autorizada para execução, ela não é executada imediatamente mas:

- Se não há uma operação conflitante $q_j[x]$ em *op-em-transito*, então a operação $p_i[x]$ é executada e inserida em *op-em-transito*.

- Se, por outro lado, há uma operação conflitante $q_j[x]$ em *op-em-transito*, então $p_i[x]$ não é executada imediatamente e a operação $p_i[x]$ é inserida em *op-em-espera*.

- Quando a transação T_j termina (com ou sem sucesso), todas as suas operações são removidas da fila *op-em-transito*; se T_j foi abortada, então suas operações também são removidas de *op-em-espera*. Em seguida, por ordem de marca de tempo, as operações de cada transação em *op-em-espera* são examinadas e, se não houver mais conflito, elas serão executadas e inseridas em *op-em-transito*.

Note que as filas *op-em-transito* e *op-em-espera* funcionam como bloqueios, mas não há perigo de travamentos uma vez que a regra MT é obedecida. Tais filas garantem que os escalonamentos serão estritos. Para melhorar a eficiência, as filas *op-em-transito* e *op-em-espera* podem ser parametrizadas por item de banco de dados e tipo operação (r ou w).

Com este arranjo, o escalonamento do exemplo anterior ficaria:

$$E = r_j[y]; r_i[x]; a_i; w_j[x]; c_j; r_i[x]; w_i[y]; c_i$$

uma vez que a operação $r_i[x]$ agora fica na fila *op-em-espera* até que T_j termine.

Por fim, é importante notar que o MT estrito ainda pode sofrer do problema de reinícios cíclicos. No exemplo acima, nada impede de $r_i[x]$ ser executada logo após a_i, o que levaria ao aborto de T_j e a uma potencial situação de reinícios cíclicos.

2.6.3 MT Distribuído

O protocolo MT é bastante fácil de ser distribuído. Basta que cada nó do sistema possua seu próprio escalonador MT, e que cada nó possua um identificador único. Desta forma, o esquema proposto acima de produzir uma marca de tempo com dois componentes pode facilmente ser implementado.

A decisão sobre permitir, atrasar ou rejeitar uma operação sobre um item de dados x é tomada localmente, e cada escalonador mantém as informações sobre seus dados locais apenas, independentemente de onde foi iniciada a transação que realiza as operações. O funcionamento do MT permanece basicamente o mesmo. Além disso, o MT distribuído possui a vantagem de não necessitar a comunicação entre os nós para coordenar a detecção de travamentos distribuídos. Ou seja, não há nenhuma necessidade de comunicação entre os escalonadores MT.

Por fim, um escalonador por MT pode participar de um BD distribuído com outros escalonadores BBF, bastando que os escalonadores BBF também atribuam marcas de tempo a suas transações. Quanto ao procedimento para a confirmação distribuída, ela permanece a mesma independentemente do escalonador, pois é um problema de recuperação e será descrito no próximo capítulo.

2.6.4 Implementação do MT

Além do gerenciamento de marcas de tempo por cada escalonador e os reinícios cíclicos, ambos discutidos acima, um outro problema do escalonador MT é o armazenamento dos valores $r\text{-}mt(x)$ e $w\text{-}mt(x)$ para cada item de dados no banco de dados.

Uma possível solução para tal problema seria manter-se duas tabelas, uma para *r-mt* e outra para *w-mt*, com apenas alguns valores de x. Estas tabelas possuem duas colunas, uma para x e outra para o valor de $r\text{-}mt(x)$ ou $w\text{-}mt(x)$.

Além disso, guarda-se o valor da marca de tempo da transação mais antiga no sistema, *min-mt*. Desta forma, entradas nas tabelas que são menores que *min-mt* representam dados alterados há muito tempo e que nenhuma transação no sistema pode chegar "tarde demais" para alterar tais dados.

Desta forma, estas entradas nas tabelas podem ser removidas, mantendo pequeno o tamanho das tabelas.

Se um dado foi apagado das tabelas $r\text{-}mt$ e $w\text{-}mt$ isto significa que a marca de tempo de qualquer transação executando no sistema será maior que $r\text{-}mt(x)$ ou $w\text{-}mt(x)$, e portanto que a operação deve ser aceita pelo sistema.

Uma descrição pormenorizada de diversas implementações do protocolo MT pode ser encontrada em [Fer96].

Capítulo 3

Noções de Recuperação

3.1 Falhas

Uma das maiores vantagens de se utilizar um sistema de transações atômicas é a possibilidade de processarmos transações de uma maneira *tolerante a falhas*. Existem vários tipos de falhas, e nós vamos classificá-las em quatro tipos. Cada um desses tipos de falhas possui um tratamento distinto.

Falhas de transação. Falhas de transação causam o aborto da transação. Este tipo de falha pode ter diversas causas. Por exemplo, uma transação falha (e é abortada) se tentar realizar uma operação ilegal, tais como divisão por 0, violação de memória, acesso a recursos não autorizados. É o tipo de falha causado por erros de programação, ou por erros de operação. Também é uma falha de transação os abortos causados por *time out* ou *estouro de temporização*: uma transação excedeu o tempo máximo de processamento que lhe foi permitido; tal falha garante que toda transação sempre termina.

O procedimento de recuperação de erro de uma falha de transação é o próprio mecanismo de aborto, ou seja, desfazer todas as atualizações feitas pela transação; este mecanismo será descrito na Seção 3.3. Podemos ainda incluir como falha de transação um aborto causado ao se dissolver um travamento. Neste caso, o reinício da transação abortada é feito automaticamente pelo sistema.

Falha de Sistema. A falha de sistema é caracterizada pela perda dos conteúdos de memória volátil do sistema. Tipicamente, é o tipo de falha que ocorre por uma interrupção no fornecimento de energia elétrica. Devido a possibilidades de falhas de sistema, o adicionar $[T_i, x, v]$ ao log; gerenciador do banco de dados deve manter uma cópia *estável* do seu conteúdo em memória secundária, tipicamente um disco.

A recuperação de falhas de sistema é o principal tema deste capítulo.

Falha de Meio de Armazenagem. Este tipo de falha é caracterizado pela perda dos conteúdos do banco de dados estável. Tipicamente, é causada por um *crash* de disco, onde os conteúdos do banco de dados estável são destruídos. Sempre que ocorre uma tal falha assume-se que uma falha de sistema também ocorreu, ou seja, não se pode nem confiar nos conteúdos de memória volátil.

Desta forma, caso não haja uma cópia do banco de dados estável em algum outro meio de armazenagem (fita, ou disco backup), não há como se recuperar deste tipo de erro. No caso em que há uma cópia, o procedimento de recuperação é o restabelecimento da cópia do banco de dados estável, seguido do procedimento de reinício idêntico ao da falha de sistema.

Falhas Distribuídas. Em sistemas distribuídos, dois novos tipos de falha são possíveis: as falhas de nó e as falhas de comunicação. Uma falha de nó é causada quando um dos nós participantes numa trasação distribuída sofre uma falha de sistema. Neste caso, os dados situado naquele nó ficam temporariamente inacessíveis e as transações distribuídas que necessitam destes dados não podem continuar. As falhas de comunicação são caracterizadas pelo isolamento de um conjunto de nós do sistema devido a interrupção nos serviços de comunicação.

Em ambos os casos, temos o problema de garantir a terminação consistente de transações globais em todos o nós participantes da transação global, mesmo na presença de falhas. Mecanismos para solucionar este problemas serão descritos na Seção 3.6.

Nas próximas seções vamos nos concentrar em estudar a recuperação de falhas de sistema e a perda da memória volátil. Estamos assumindo aqui que tais falhas sempre são detectadas (via códigos de redundância, bit de paridade, etc). Portanto, vamos primeiro estudar o Gerenciador de Dados, responsável pelo gerenciamento dos dados em memória primária e seu descarregamento em memória secundária. Em seguida, vamos estudar os procedimentos que devem ser tomados nas operações regulares do banco de dados para que este possa se recuperar na eventualidade de uma falha.

A recuperação da falhas e o gerenciamento da estrutura de *log*[1] (ou diário) necessário para a recuperação de falhas serão abordados.

Por fim, na Seção 3.6 descreveremos um procedimento que garante a confirmação atômica, tolerante a falhas, em sistemas distribuídos.

3.2 O Gerenciador de Memória

Na arquitetura de referência de um gerenciador de bancos de dados da Figura 2.3 temos que, logo abaixo do Escalonador está o Gerenciador de Acessos a Dados (GD). Este gerenciador está dividido em duas partes: o Gerenciador de Memória (GM) e o Gerenciador de Recuperações (GR). Nesta seção, vamos detalhar o funcionamento do GM.

O interior do GD está detalhado na Figura 3.1. O GD recebe do escalonador os comandos das operações básicas de escrita, leitura, aborto e confirmação. Além disso, ele também recebe o comando de *reinício*, que é emitido na recuperação de alguma falha de sistema e vai diretamente ao GR. O GR envia ao GM duas operações: *Busca* de dado e *Descarga* de memória.

O banco de dados propriamente dito fica residente em memória secundária, normalmente um disco magnético, estável, cujos conteúdos não se perdem durante uma falha de sistema. No entanto, o tempo de acesso à memória secundária ainda é muito maior que o tempo de acesso à memória primária, e portanto, por razões de eficiência, tenta-se manter uma parte do banco de dados em memória primária. Esta parte da memória primária usada para espelhar parte do banco de dados é muitas vezes chamada de *cache* do banco de dados.

A memória secundária contém dois tipos de informação. Uma delas é o banco de dados estável, a outra é o *log* [2], uma estrutura de dados que contém informação sobre *todas* as operações básicas realizadas pelas transações executadas no sistema. O *log* e sua manipulação será discutido na apresentação do Gerenciador de Recuperação (GR) na Seção 3.3.

O cache [3] do banco de dados é um vetor de posições; cada posição

[1] O termo *log* é tão usual que fica difícil até encontrar um termo substituto em português.

[2] Uma possível tradução da palavra inglesa *log* seria *diário*, ou seja, um repositório de anotações sobre atividades, no caso, do banco de dados.No entanto o termo *log* está tão difundido que preferimos mantê-lo sem tradução.

[3] O cache do banco de dados não deve ser confundido com o cache de hardware, que é um dispositivo de memória ultra-rápida usada para agilizar a comunicação entre um processador e a memória primária do sistema. No entanto, em ambos casos o conceito de cache é o mesmo: aumentar a eficiência no acesso a dados.

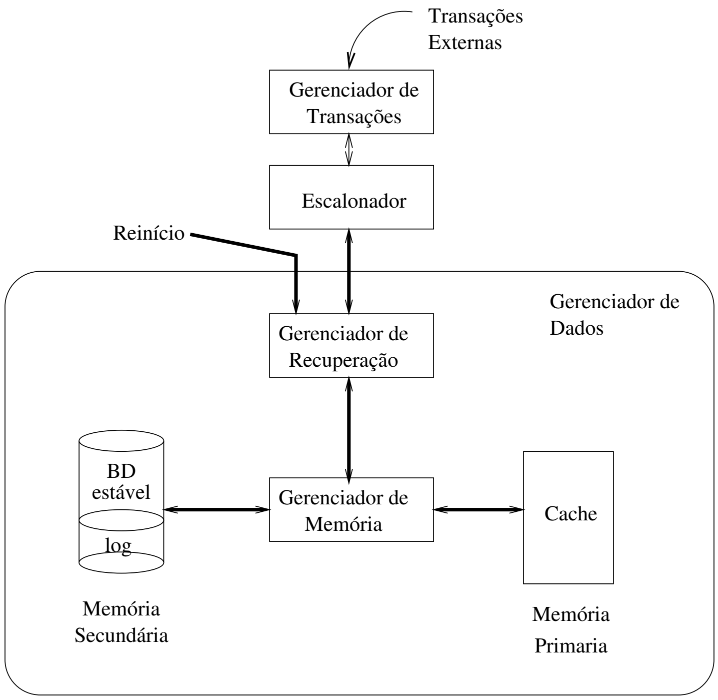


Figura 3.1: O Gerenciador de Dados

contém, pelo menos, um *bit de alteração* e um valor de item de dado. Além disso, deve haver um *diretório de cache* que indica qual item de dados está presente no cache e em que posição; outras informações possíveis no diretório de cache são: transação que solicitou o posicionamento do dado no cache, tempo da solicitação, etc. Abaixo ilustramos um conjunto mínimo de atributos das estruturas do cache.

Cache

Posição	*Bit de Alteração*	*Valor*
0	1	"José da Silva"
1	0	2
2	1	16/8/1999
⋮		

Diretório de Cache

Item de Dado	Posição
x	2
y	0
z	1
⋮	

Quando uma operação de leitura é submetida ao Gerenciador de Acessos a Dados, verifica-se se o dado está presente no cache. Se o dado não está presente na memória primária, conforme indica o diretório de cache, envia-se ao Gerenciador de Memória uma operação de *Busca* do dado no banco de dados estável. Com o dado no cache o seu valor é retornado ao Escalonador (e daí ao Gerenciador de Transação e, finalmente, à transação que requisitou o dado).

Quando um dado é escrito no banco de dados, também verifica-se se o dado está presente no cache. Caso não esteja, procura-se uma posição do cache para este dado de acordo com uma *estratégia de reposição* do cache; uma vez decidida qual a posição que o dado deve ocupar, uma operação de *Descarga* da posição é emitida. Em seguida, o dado é escrito na posição de cache liberada e o *bit de alteração* é posto em 1. O dado só vai ser escrito no banco de dados estável quando esta posição for descarregada.

Portanto, o Gerenciador de Memória recebe dois comandos do Gerenciador de Recuperação:

- `Busca(Item_de_dado)`: Verifica se o item de dado está no diretório de dados; se estiver, retorna o valor contido na posição do cache correspondente.

  Se o dado não estiver no cache, procura uma posição vazia do cache. Se não houver uma posição vazia, escolhe alguma posição ocupada e a descarrega. Uma vez que uma posição liberada foi encontrada, o item de dado é lido do banco de dados estável para a posição do cache liberada, e o diretório do cache é atualizado.

- `Descarrega(Posição)`: Verifica se a posição possui o bit de alteração em 1. Se o bit de alteração estiver em 0, não há nada a fazer. Se o bit estiver em 1, então descobre no diretório de cache qual o item de dado correspondente à posição e escreve o valor da posição no item de dados do banco de dados estável.

  Durante a descarga, o Gerenciador de Memória precisa garantir que o dado não seja nem alterado nem uma nova operação de descarga seja iniciada.

  A operação de descarga pode ser requisitada pela necessidade de liberar uma posição, e também pode ser requisitada no processo de confirmação de uma transação, quando é possível que o gerenciador de recuperação exija que todos os dados escritos pela transação sejam descarregados no banco de dados estável.

Como o tamanho do *cache* em memória é limitado, há que se implementar estratégias para substituição e alocação dos dados em memória. Há basicamente três estratégias: a) retirar os dados que foram menos utilizados até um determinado momento, conhecido como *Least Recently Used - LRU*; b) considerar que os primeiros dados a serem alocados são os primeiros a serem retirados (*First-In First-Out, FIFO*); c) incluir um parâmetro de frequência de utilização do dado combinado com a estratégia do item a, de modo que o dado a ser retirado terá como avaliação, além do parâmetro do período de tempo de utilização, também a quantidade de utilização.

3.3 O Gerenciador de Recuperação

A Figura 3.1 ilustra a arquitetura do gerenciador de banco de dados centralizado incluindo vários elementos, desde a solicitação da transação até a atualização efetiva no banco de dados. Como mencionado na seção 3.2 o GC provê funções de carregamento e estratégia para substituição dos dados em disco para a memória e vice-versa. Assim o Gerenciador de Recuperação (GR) é o módulo de software entre o GC e as transações, requisitadas pelo escalonador e o evento de recuperação do banco de dados. Para desempenhar tal funcionalidade é necessário definir uma interface com os seguintes procedimentos.

1. `GR-leitura`(T_i, x): leitura do valor de x para a transação T_i;
2. `GR-escrita`(T_i, x, v): escrita do valor de v em x para a transação T_i;
3. `GR-confirma`(T_i): confirmação da transação T_i;
4. `GR-aborta`(T_i): cancelamento da transação T_i;
5. `GR-reinicia`: inicia o processo de recuperação no evento de uma falha.

O GR executa as operações acima de forma atômica, de acordo com a sequência repassada pelo escalonador, de modo que nunca há conflito entre as operações. O algoritmo de recuperação pode se tornar muito complexo quando assumimos que as operações não são atômicas. Isto implica numa abordagem de recuperação em cascata, que está fora do escopo deste capítulo.

3.3.1 Recuperação com *log*

Na implementação da recuperação baseada em *log* pressupomos que existe uma única localização do dado em memória secundária. A estrutura de um arquivo de *log* é dependente da implementação do respectivo algoritmo de recuperação. Entretanto elementos básicos podem ser descritos como os dados dos registros e as funções que traduzem as operações das transações.

São os seguintes os dados do registro de *log*:

T_i:	identificação da transação;
x_j:	localização em memória secundária do item de dados x;
v_a:	valor do item de dados x anterior a operação;
v_p:	valor do item de dados x posterior a operação.

E as seguintes funções do registro de *log*:

(T_i, inicia) :	inicia transação T_i;
(T_i, x_j, v_a, v_p) :	escreve ou lê a posição x_j envolvendo valores v_a e v_p;
(T_i, confirma) :	confirma a transação T_i;
(T_i, aborta) :	aborta a transação T_i.

Para ilustrar como uma transação pode ser traduzida num arquivo de log, vamos considerar as seguintes transações com o respectivo arquivo de *log*. Pode-se observar que as operações com os dados feitas em memória primária não tem correspondência no arquivo de *log*. Apenas são representadas as operações que interagem com a memória secundária.

transações	arquivo de *log*
T_0: início	(T_0 inicia)
leia (A)	(T_0, A_j, 30, 30)
A := A - 20	
escreva (A)	(T_0, A_j, 30, 10)
leia (B)	(T_0, B_j, 20, 20)
B:= B + 30	
escreva (B)	(T_0, B_j, 20, 50)
T_0: fim	(T_0, confirma)
T_1: início	(T_1 inicia)
leia (C)	(T_1, C_j, 0, 0)
C:= C + 10	
escreva (C)	(T_1, C_j, 0, 10)
T_1: fim	(T_1 confirma)

Técnicas de Recuperação de Informação são utilizadas para restaurar um banco de dados levando-o a um estado consistente, após a ocorrência de uma determinada falha. Para isso, as informações das transações ocorridas sobre os dados precisam ser de alguma forma monitoradas. Tais informações são tipicamente armazenadas nos arquivos de *log*.

O processo de recuperação está vinculado as funções do sistema operacional, de modo específico, ao mecanismo de *caching* e paginação de disco em memória principal. As páginas de dados em disco estão associadas a um ou mais item de dado numa estrutura de dados na memória principal como visto na seção 3.2.

Uma típica estratégia de recuperação de dados envolve duas situações:

1. Uma falha *catastrófica*, na qual só é possível recuperar o banco de dados através de uma cópia dos dados feita anteriormente, perdendo as transações ocorridas entre o intervalo de tempo da falha e a última cópia realizada.

2. Uma falha *não catastrófica* na qual o banco de dados ficou num estado inconsistente, mas é possível recuperar os dados a partir da eliminação do conjunto de operações que gerou tal inconsistência. Neste caso as transações são desfeitas até encontrarmos um estado consistente do banco de dados. Não é necessário copiar integralmente o arquivo anterior do banco de dados. Basta consultar as informações existentes nos sistemas de *log* e iniciar o processo de recuperação.

Na situação não catastrófica há duas abordagens: *atualizações postergadas* e *atualizações imediatas*. Nas *atualizações postergadas* as modificações dos dados oriundas das transações só são repassadas para a base de dados após o ponto de confirmação da transação. Todas as modificações ocorridas antes da confirmação da transação, são registradas num espaço local da transação e no arquivo de *log*. Se a transação falha antes do ponto de confirmação, nada foi repassado para a base de dados. Desse modo nenhuma operação precisa ser realizada, ou não é preciso disparar a operação *UNDO*[4]. Entretanto é necessário refazer as transações que tenham o registro de confirmação no arquivo de *log*, pois há a possibilidade das modificações não terem sido repassadas para o banco de dados. Nesse caso essas transações devem ser refeitas, disparando a operação *REDO*[5]. Tal algoritmo é também conhecido por *No-UNDO/REDO*.

Na técnica de *atualizações imediatas* as operações das transações são sempre repassadas para o banco de dados, mesmo antes do ponto de confir-

[4]O termo *UNDO* é utilizado em inglês para a operação que desfaz as transações inconsistentes.

[5]O termo *REDO* é o termo utilizado em inglês para a operação que refaz as transações confirmadas.

mação da transação. Neste caso, as informações das transações também são armazenadas no arquivo de *log*. Na ocorrência de uma falha, o processo de recuperação deverá desfazer as operações das transações que não atingiram o ponto de confirmação e ainda refazer aquelas operações das transações confirmadas. Tal técnica de recuperação é também conhecida como algoritmo de *desfazer/refazer* que será visto em detalhes na Seção 3.4. Uma variação de tal algoritmo pode ser considerada, se antes de repassarmos as atualizações para o banco de dados, impormos a precedência da escrita no arquivo de *log*, dispensando assim a necessidade da utilização da operação de refazer, ou como conhecido algoritmo de *desfazer/não-refazer*.

3.3.2 Recuperação com páginas sombras

Na recuperação com tabela de endereços de páginas duplicadas [6] pressupomos a existência da cópia de um determinado dado, em páginas distintas do banco de dados. Apenas uma versão é utilizada como válida durante o processamento de uma transação. Para a implementação desta técnica é necessário que o banco de dados esteja construído em tamanhos fixos de páginas [7] em disco. Cada entrada na tabela de páginas aponta para as páginas em disco. Existem dois tipos de tabelas de páginas: corrente e a sombra conforme ilustrado na Figura 3.2. Inicialmente as tabelas apontam para o mesmo endereço de página. Na medida em que as transações solicitam a escrita de dados, o novo dado é alocado numa nova página do banco de dados e seu endereço anotado na tabela corrente de modo que existam dois valores: o novo e o anterior referenciados respectivamente pelas tabelas de páginas corrente e sombra. A tabela corrente é utilizada durante o processamento da transação. A tabela sombra nunca é modificada durante a execução da transação.

Para recuperação na ocorrência de falha deve-se descartar as páginas do banco de dados referenciadas pela tabela corrente. O estado consistente do banco de dados está armazenado na tabela sombra. Portanto esses valores são considerados e utilizados para re-execução da mesma ou de outras transações. Se uma transação chegar ao ponto de confirmação então o inverso é realizado, ou seja, descarta-se as páginas do banco de dados referenciados pela tabela sombra. Como esta técnica não envolve os mecanismos de refazer ou desfazer transações o algoritmo de recuperação é conhecido como *não-desfazer/não-refazer*.

---

[6] em inglês, *shadowing*.

[7] também conhecido como blocos de dados.

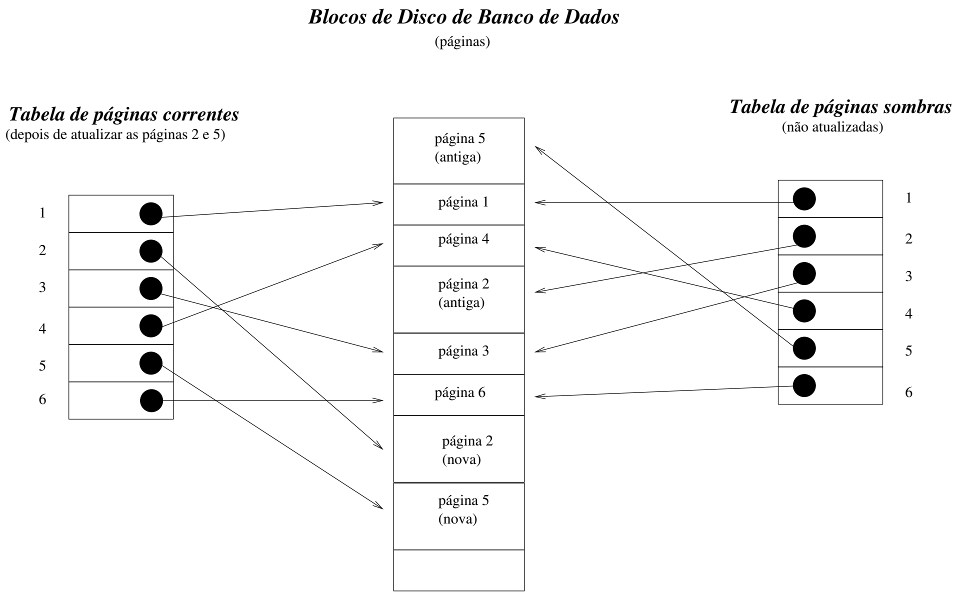


Figura 3.2: Recuperação com páginas sombras

A grande vantagem da técnica em questão, se prende ao fato de que para o processo de recuperação não seja necessário nem refazer ou desfazer operações das transações. Por outro lado, as transações confirmadas geram trocas sucessivas de localização e alocação das páginas em disco. Se a página for de tamanho significativo isto causará um grande trabalho adicional para o serviço de alocação de blocos do GD. Outra complicação é como fazer de maneira eficiente a coleta de páginas que se tornaram lixo, no momento em que a transação foi confirmada. Esse processo não pode ser muito demorado pois os dados devem estar disponíveis para as transações seguintes.

3.4 Detalhes do Algoritmo *desfazer/refazer* de Recuperação

Para descrever o algoritmo de *desfazer/refazer* para recuperação das transações utilizaremos cinco procedimentos como mencionado em [BHG87]: `GR-leitura`, `GR-escrita`, `GR-confirma`, `GR-aborta` e `GR-reinicia`. Para melhor compreensão dos procedimentos a serem descritos, tenha como referência a Figura 3.1.

`GR-leitura`(T_i, x)

1. se x não estiver no cache, localizá-lo e carregá-lo para o cache;
2. adicionar $[T_i, x,]$ ao log como leitura;
3. devolver o valor de x para o escalonador.

`GR-escrita`(T_i, x, v)

1. adicionar T_i à lista ativa, caso não esteja nesta lista;
2. se x não estiver no cache, localizá-lo e carregá-lo para o cache;
3. adicionar $[T_i, x, v]$ ao log como escrita;
4. escrever v no espaço/posiçao do cache ocupado por x;
5. informar o fim do processamento de `GR-escrita`(T_i, x, v) para o escalonador.

`GR-confirma`(T_i)

1. adicionar T_i na lista de confirmação;
2. adicionar $[T_i, confirma]$ ao log;
3. informar a confirmação de T_i para o escalonador;
4. eliminar T_i da lista de transações ativas.

`GR-aborta`(T_i)

1. para cada item x atualizado por T_i
   - se x não estiver no cache, alocar espaço/posição para x;
   - copiar o valor de x anterior à transação T_i no espaço alocado para x no cache;
2. adicionar T_i na lista de transações abortadas;
3. adicionar $[T_i, aborta]$ ao log;
4. informar a transação abortada T_i para o escalonador;
5. eliminar T_i da lista de transações ativas.

`GR-reinicia`

1. descartar todos os espaços/posições do cache;
2. fazer as variáveis receber vazio: `redone`:=$\emptyset$ e `undone`:=$\emptyset$;
3. considerando o início do arquivo de log, para cada entrada $[T_i, x, v]$ no arquivo de log, se x não pertence a `redone` $\cup$ `undone`, então:

- se x não está no cache, alocar espaço/posição para x;
- se T_i está na lista de transações confirmadas, copiar v no espaço/posição de x no cache e ainda fazer `redone`:= `redone` $\cup\{x\}$;
- se T_i está na lista de transações abortadas ou na lista de transações ativas mas não na lista de transações confirmadas então copiar o valor de x anterior a transação T_i no espaço/posição de x no cache e ainda fazer `undone`:= `undone`$\cup$ $\{x\}$;

4. para cada T_i, no arquivo de lista de transações confirmadas, se T_i estiver na lista de transações ativas, remover a transação T_i da lista de transações ativas;
5. informar o fim do procedimento GR-reinicia para o escalonador.

3.5 Impacto dos Algoritmos de Recuperação no Controle de Concorrência e na Distribuição de Dados

Neste ponto, convêm lembrarmos o porquê de de estarmos investigando a recuperação de dados na descrição do controle de concorrência. Não basta apenas dizer que a recuperação de falhas é um tópico importante, ou que este tópico está intimamente relacionado com os dois tópicos principais deste livro, o controle de concorrência e a distribuição de dados. É preciso detalhar este relaconamento íntimo e é a isso que se dedica esta seção.

Recuperação e Controle de Dados

O relacionamento entre a recuperação de falhas e o controle de dados é bidirecional. O ponto em que este interrelacionamento se dá, na nossa arquitetura de referência da Figura 3.1 se dá na interface entre o Escalonador e o Gerenciador de Recuperação. Esta interdependência detalhamos a seguir.

Inicialmente, o controle de dados tem uma função determinante na recuperação de falhas. Como vimos pelos algoritmos acima, todo o esquema de recuperação de falhas apresentado se baseia no modelo transacional. Em particular, a propriedade de atomicidade é o pilar básico dos procedimentos de recuperação: transações confirmadas devem ter seus resultados mantidos, transações abortadas ou ativas no momento de uma falha de sistema devem ser abortadas pelo sistema. Esta diretriz básica é a alma dos mecanismos acima descritos; e podemos até dizer que o resto são detalhes de implementação.

Um destes "detalhes de implementação" é a garantia que o controle de concorrência fornece ao Gerenciador de Recuperação de que os escalonamentos são *estritos*; esta propriedade é usada para facilitar a implementação de abortos restabelecendo as imagens prévias. Sem esta garantia do escalonador, o mecanismo de implementação de abortos ficaria consideravelmente mais complicado, pois em vez de percorrermos o log no sentido fim para o começo para encontrarmos o último valor do item de dados a ser refeito, precisaríamos percorrer o log inteiro, no sentido do começo ao fim, para descobrirmos qual a última transação confirmada ou ativa que escreveu no dado, pois a imagem prévia não mais pode ser usada. Isto constituiria num encarecimento considerável no processo de desfazer.

Como o desfazimento de abortos é uma atividade bastante corriqueira num banco de dados, a garantia de escalonamentos estritos pelo escalonador tem um efeito de melhorar o desempenho global do sistema, evitando o dispêndio de um tempo elevado no processamento de abortos.

Por outro lado, no sentido da dependência entre a recuperação no controle de transações, vemos que o gerenciador de recuperação, que implementa os abortos e confirmações, é o responsável pela garantia da atomicidade das transações, sendo desta forma essencial para a manutenção do modelo de transações.

Salientamos estes pontos aqui pois, no Capítulo 5 veremos modelos de transação não-ACID. Em casos em que a propriedade de atomicidade é violada, teremos necessariamente que adaptar os métodos de recuperação de falhas, a implementação de confirmações e de abortos.

Recuperação e Distribuição de Dados

Como vimos no caso acima, uma das principais funções do gerenciamento de recuperação é a implementação das confirmações. No caso de bancos de dados distribuídos, a implementação de tal confirmação é fundamental para a garantia de que uma transação global sempre atinja um estado consistente.

No caso local, há muito pouco o que coordenar para se atingir a confirmação, bastando atualizar algumas tabelas e o log local. No entanto, no caso distribuído, para ser possível atingir a confirmação de uma transação distribuída (à qual normalmente nos referimos como *confirmação distribuída*) é necessario uma camada de coordenação entre os participantes da transação.

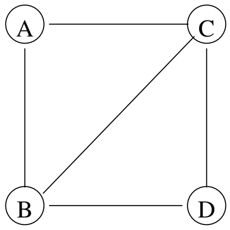
Este fato tem repercussões importantes na distribuição de dados. Por exemplo, se esta distribuição é feita por sistemas heterogêneos (ver Capítulo 4) a coordenação de confirmações distribuída apresenta uma dificuldade a mais na integração de sistemas autônomos, pois existe um antagonismo implícito entre autonomia e coordenação.

Em sistemas distribuídos homogêneos, todos os bancos de dados participantes na distribuição respeitam um protocolo de confirmação distribuída. O protocolo mais usual para isto é o protocolo de Confirmação Bifásica (no inglês: *Two-phase Commit*). Este protocolo não exige homogeneidade nos dados, e portanto pode ser implementado também por sistemas heterogêneos.

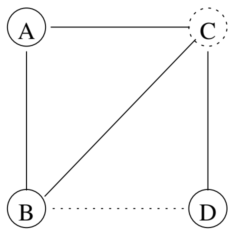
Passamos então à sua descrição.

3.6 Recuperação Distribuída e Confirmação Bifásica (*Two-Phase Commit*)

Conforme mencionado na Seção 3.1, existem dois tipos de falhas distribuídas: as falhas em nós da rede, e falhas de comunicação. Na Figura 3.3(a) vemos uma rede de computadores sem falhas e, em oposição, a Figura 3.3(b) ilustra a mesma rede com uma falha no nó C e uma falha na comunicação entre os nós B e D. Esta combinação causa um *particionamento* da rede, pois a subrede formada pelos nós A e B não pode se comunicar com a subrede formada pelo nó D, embora os nós A, B e D estejam ativos.



(a) rede sem falhas



(b) rede particionada

Figura 3.3: Falhas em redes de computadores

O particionamento de uma rede pode ser causado também apenas por falhas dos nós (por exemplo, falhando B e C, A e D ficam incomunicáveis) ou só falhas de comunicação (por exemplo, falhando entre D e B, e entre D e C, o nó D fica isolado).

O que diferencia as falhas distribuídas das falhas locais é que o particionamento da rede pode levar ao bloqueio de um nó no reinício após uma falha.

Quando um nó falha, ao ser reiniciado ele deve determinar o estado confirmado/abortado de todas as transações que estavam ativas quando a falha ocorreu. Para confirmar uma transação distribuída é necessário que todos os nós participantes concordem com a confirmação. Devido a esta necessidade de coordenação para confirmar uma transação distribuída, sempre haverá um instante em que um nó não sabe o status de uma transação global, como veremos abaixo. Dizemos que o nó está num estado indefinido.

Uma falha de um nó indefinido, seguida de um particionamento da rede, faz com que o nó não saiba se uma transação está confirmada ou abortada. Desta forma, para garantir a consistência e atomicidade das transações do sistema, este nó não pode continuar o seu reinício enquanto a indefinição não for resolvida. Nesta situação, dizemos que o nó está bloqueado.

Um resultado famoso da teoria de distribuição de dados nos diz que sempre que existem falhas na comunicação existirá a possibilidade de um nó ficar bloqueado [Mil89]. Desta forma, qualquer protocolo de confirmação distribuida deverá lidar com tal situação.

No que se segue, apresentaremos primeiro uma definição precisa do que se considera uma *Confirmação Atômica* de uma transação distribuída. Em seguida, apresentaremos o *Protocolo de Confirmação Bifásica*, que implementa confirmação atômica.

3.6.1 Confirmação Atômica de Transações Distribuídas

Vamos definir agora, precisamente, o que entendemos por *confirmação atômica de transações distribuídas*. Como o próprio nome indica, este é um mecanismo distribuído em que os vários participantes devem atingir a *mesma decisão*, quer seja a confirmação quer seja aborto. Esta definição é usada como critério de correção do protocolo de confirmação bifásica definido na Seção 3.6.2.

O mecanismo básico de decisão pode ser visto na seguinte arquitetura. Todos os participantes da transação devem chegar a uma decisão de confirmação ou aborto. Um destes participantes é o *coordenador* os outros são os *coordenados*. Assume-se que cada nó da rede possui um *log de transações distribuídas*, mantido em memória secundária de forma a sobreviver a falhas de sistema.

Cada nó participante deverá votar em uma de apenas duas opções sobre a decisão de confirmar uma determinada transação distribuída: *Sim* ou *Não* e atingir uma de duas decisões: *Confirmação* ou *Aborto*. O mecanismo de confirmação atômica deve obedecer às seguintes cinco regras:

1. Todos os participantes que atingem um decisão devem atingir à *mesma* decisão.

2. Nenhum participante pode reverter sua decisão uma vez que ela tenha sido tomada.

3. A desisão de *Confirmação* só será atingida se todos os participantes votarem *Sim*.

4. Na ausência de falhas, e se todos os participantes votarem *Sim*, então a decisão será pela *Confirmação*.

5. Se todas as falhas que ocorrerem durante o processo de confirmação atômica forem reparadas, e nenhuma nova falha ocorrer, então eventualmente uma decisão será alcançada por todos os participantes.

É importante salientar a iteração entre o procedimento de confirmação atômica e a recuperação de falhas. Primeiramente, notamos que um nó está num estado indefinido entre o momento em que ele vota *Sim* e o momento em que ele recebe a decisão sobre confirmar ou abortar. Se o nó ficar isolado na rede quando está no estado indefinido, este nó estará bloqueado. Ele somente poderá chegar a uma decisão quando a falha de comunicação estiver reparada.

Por outro lado, se o nó falhar quando estiver no estado indefinido, ele não saberá qual a decisão que foi tomada e não poderá chegar a uma decisão sozinho, pois ele também estará bloqueado. Neste caso o nó precisará se comunicar com outros nós participantes para descobrir qual a decisão tomada. A possibilidade de chegar a uma decisão sem se comunicar com outros nós é chamada de *recuperação independente*. Demonstra-se que um conjunto de nós que respeitem às regras da confirmação atômica não podem garantir que a recuperação independente de um nó na rede, pois a obediência às regras acima sempre gera um período de incerteza, que sempre poderá levar a um bloqueio do nó.

São duas as implementações principais da confirmação atômica: o protocolo de *confirmação bifásica* [Gra78, LS76] e um refinamento seu, o protocolo de *confirmação trifásica* [Ske82]. O primeiro deste, *confirmação bifásica*, é mais básico e é muito mais frequentemente encontrado em implementações na prática, e portanto será discutido em detalhe a seguir.

3.6.2 O Protocolo de Confirmação Bifásica

O protocolo de *confirmação bifásica* (2PC)[8] foi inicialmente publicado num artigo clássico de Gray sobre as características que um sistema operacional precisa suportar para servir de base a um sistema gerenciador de Bancos de Dados [Gra78]. As idéias básicas do protocolo 2PC já haviam circulado na forma de um relatório técnico em [LS76]. Até hoje, este protocolo permanece sendo o mecanismo mais utilizado na implementação da confirmação atômica em sistemas distribuídos.

Este protocolo requer que haja um *coordenador* do procedimento de confirmação, e pode haver diversos outros nós participantes. O procedimento básico, assumindo-se que a transação decidiu solicitar sua confirmação[9], possui duas fases, como o nome indica. A primeira é a fase de *Votação* e a segunda é a fase de *Decisão*. Uma vez que um participante já sabe qual decisão foi atingida, ele pode encerrar a sua participação no protocolo.

Os passos do protocolo são os seguintes:

1. O coordenador solicita a todos os participantes que votem a favor ou contra a solicitação de confirmação.

2. Cada participante responde ao coordenador com um voto de *Sim* ou de *Não*. Se um participante vota *Não*, ele decidiu abortar e já pode até desfazer a parte local da transação; o coordenador se encarregará de abortar a transação nos outros nós, e o nó que votou *Não* encerra a sua participação no protocolo.

3. Se um dos votos for *Não*, o coordenador envia mensagens de Aborto a todos os participantes (menos a quem já votou *Não*). Se todos os nós votarem *Sim*, o coordenador decide por confirmar a transação e envia mensagens de Confirmação a todos os participantes. Em ambos os casos, o coordenador encerra sua participação no protocolo ao enviar as mensagens.

4. Cada participante que votou *Sim* fica a espera de uma mensagem de Aborto ou de Confirmação vinda do coordenador. Quando a mensagem chega, ele decide de acordo com seu conteúdo e encerra a sua participação no protocolo.

[8] Do inglês, Two-Phase Commit protocol.

[9] Normalmente, a solicitação é feita através de um comando de COMMIT existente em várias linguagens de transação, ou simplesmente alcançando o término normal do programa da transação.

A fase de votação são os passos 1 e 2. A fase de decisão são os passos 3 e 4. Na ausência de falhas, claramente todos os participantes chegam à mesma decisão. Se todos votarem *Sim*, o coordenador decidirá pela confirmação e todos participantes receberão uma mensagem de confirmação. Se um nó votar não, o coordenador decidirá por abortar, e todos os participantes serão abortados.

Portanto, os quatro primeiros itens do mecanismo de confirmação atômica estão respeitados pelo 2PC. O item mais difícil de respeitar é o último, que está relacionado com a recuperação de falhas.

Falhas de comunicação ou de nó pode deixar participantes e coordenador eternamente esperando. Para evitar isto, cada participante possui um *temporizador*[10].Quando há um estouro na temporização, os participantes reagem da seguinte maneira:

- Um nó que está esperando pela solicitação de voto pode decidir unilateralmente por abortar. Quando a solicitação chegar, se chegar, ele poderá votar *Não* e a decisão geral será pelo aborto.

- O coordenador, após enviar a solicitação de votos aos participantes, ainda não atingiu nenhuma decisão. Neste momento, nenhum participante pode ter decidido pela confirmação, que só ocorrerá quando o coordenador enviar a mensagem de confirmação. Portanto, se algum nó demorar a responder e houver um estouro na temporização do coordenador, ele poderá decidir pelo Aborto e enviar uma mensagem de aborto a todos os participantes.

- Cada participante que votou *Sim* está esperando por uma mensagem de confirmação ou aborto do coordenador. Neste momento, este participante não pode decidir unilateralmente nem por abortar nem por confirmar. O nó está bloqueado.

Quando um nó está bloqueado, pode ser que o coordenador já chegou a uma decisão, mas uma falha de comunicação impediu que esta chegasse ao participante. O participante pode tentar recuperar a comunicação com o coordenador para descobrir a decisão. Isto pode levar a um longo bloqueio do participante, o que é indesejável.

No entanto, os participantes podem se comunicar entre si, através de um *protocolo de terminação cooperativa*. Neste caso, um participante envia a outro participante um pedido de status da decisão para uma determinada transação (lembre-se transações globais possuem um identificador global

[10]No inglês, *timeout*.

único). Ao receber um pedido de status, um participante pode reagir de três formas. Se ele já sabe qual é a decisão, ele responde com a mensagem de decisão; se ele ainda nem votou (e portanto não está bloqueado) ele pode decidir unilateralmente pelo aborto, e responder com uma mensagem de aborto; por fim, se o nó também está bloqueado ele não pode ajudar outro participante, e informa que está bloqueado.

Se um nó falhou no estado bloqueado, na sua recuperação ele tentará acionar o protocolo de terminação cooperativa. Se todos os nós e canais de comunicação que falharam forem recuperados, eventualmente todos os participantes chegarão a uma única decisão.

Se um nó falhar em algum estado que não o de bloqueio, ele sempre poderá decidir unilateralmente pelo aborto durante a sua recuperação.

E como um nó que falhou sabe em que estado estava no instante da falha? Para isso é necessário a existência de um *log para Transações Distribuídas* (logTD) em cada nó participante da transação distribuída. Cada participante tem acesso a apenas o seu logTD, e este logTD é mantido e atualizado durante o processamento da transação.

Primeiramente, se um participante decide abortar uma transação global antes de iniciar-se o procedimento 2PC, ele escreve tal fato no logTD, indicando o ID global único da transação (TIDg) abortada. Desta forma, se um pedido de voto chegar para a confirmação desta transação, ele poderá votar *Não*. Se algum outro participante lhe enviar um pedido de status através do protocolo de terminação cooperativa, ele poderá lhe informar a decisão por aborto.

Quando um coordenador envia um pedido de votos a todos os participantes, ele escreve em seu logTD uma mensagem INÍCIO-2PC(TIDG), gravando no logTD inclusive os identificadores dos nós participantes.

Se um participante vota *Sim* (VOTA-SIM(TIDG)), este voto é gravado em seu logTD *antes* de enviar o voto ao coordenador. Se a ordem fosse inversa, o seguinte cenário seria possível: o participante envia um voto *Sim* ao coordenador, e falha antes de escrevê-lo no logTD. Desta forma, ao se recuperar, o nó poderia decidir unilateralmente pelo aborto, enquanto existe a possibilidade da decisão global ser pela confirmação. Para evitar esta possibilidade, o participante só envia o seu voto após escrevê-lo em seu logTD.

Igualmente, o coordenador escreve (CONFIRMA(TIDG)), a decisão de confirmação, *antes* de enviar a mensagem de confirmação aos participantes. O motivo para tal ordem é o mesmo do caso anterior, pois o coordenador pode falhar e, na sua recuperação o dado que ele utilizará para saber o status da transação é o dado anotado no logTD. Em hipótese alguma pode se admitir que o coordenador envie uma msg de confirmação aos participantes antes de registrá-la em seu logTD, para então falhar e na recuperação decidir pelo aborto.

Por outro lado, se o coordenador decidir pelo aborto, ele tem a liberdade de escrever sua decisão (ABORTO(TIDG)) antes ou depois de enviar as mensagens.

Por fim, um participante que votou *Sim* e recebe uma mensagem de confirmação ou aborto do coordenador, simplesmente escreve esta decisão em seu logTD; no caso de aborto, obviamente, a transação deve ser desfeita. A informação gravada no logTD será ainda usada caso um outro participante lhe envie um pedido de status através do protocolo de terminação cooperativa.

Como vimos, um nó participante pode ficar bloqueado sempre que falhar no estado de incerteza. Existem protocolos que restringem os casos em que o bloqueio ocorre ao caso de uma falha total de comunicações, ou seja, quando um nó falha e se recupera com as comunicações ativas, ele não ficará bloqueado. O protocolo de confirmação trifásica (3PC) é um tal protocolo; para uma explicação de seu funcionamento, recomendamos consultar [BHG87]. No entanto, paga-se um custo por esta maior resistência a situações de bloqueio, pois a confirmação é mais lenta, requer uma fase a mais e um número de mensagens também maior. Na prática, o protocolo 2PC continua sendo o preferido da maioria das implementações.

Capítulo 4

Fundamentos de Distribuição de Dados

4.1 Introdução

Com a possibilidade de interligação de múltiplos equipamentos em rede, existem diversas alternativas para a distribuição dos sistemas de armazenamento e processamento de informação. Neste capítulo veremos os conceitos e as técnicas para os projetos genéricos de distribuição de dados e de modo específico as técnicas de fragmentação e alocação para o modelo relacional de dados. Novas alternativas de distribuição em modelos distintos do relacional serão vistos no Capítulo 8.

Do Ponto de vista do emprego de SGBD's as opções de distribuição para dar suporte a armazenagem dos dados de uma aplicação, podem ser, de maneira simplificada, classificadas em:

Sistemas Distribuídos: neste caso, o sistema operacional provê recursos para que as aplicações em execução, em diferentes máquinas, possam trocar informação. A informação está centralizada e os processos estão distribuídos. Assim, uma aplicação pode acessar dados cuja proteção de acesso é garantida pelo sistema operacional em tendo como unidade de referência os arquivos. Múltiplas aplicações podem acessar o mesmo dado, desde que cada uma delas respeite um protocolo de acesso definido pelo projetista do sistema. Não existe um controle de acesso em registros. Exemplos típicos são sistemas de controle de postos de vendas: diversos terminais (caixas registradoras) são gerenciadas por um aplicativo, que compartilha dados com programas geradores de relatórios. A característica básica é a existência de apenas um programa que modifica os dados num determinado instante.

Sistemas Cliente-Servidor: nesse caso, o SGBD provê recursos para que as diversas aplicações possam compartilhar os dados por ele gerenciados. A informação está centralizada, mas diversos aplicativos podem solicitar a manipulação de dados ao SGBD. Do ponto de vista do sistema operacional, o SGBD é uma aplicação com direito de acesso exclusivo aos dados, a qual pode manipular esses dados atendendo às solicitações dos demais aplicativos. O controle de acesso aos dados pode ser tão refinado de acordo com as exigências das aplicações. Porém, se existirem diversos servidores no sistema, os aplicativos devem saber onde a informação pode ser encontrada. Exemplos típicos são sistemas de informação gerenciais e sistemas baseados em transações. A característica básica é a possibilidade de diversos programas poderem acessar os dados tanto para consulta quanto para modificação, simultaneamente, em servidores explícitos.

Sistemas de Bases de Dados Distribuídas: nesse caso, a informação está distribuída em diversos servidores. Cada servidor atua como no sistema cliente-servidor, porém as consultas oriundas dos aplicativos são feitas para qualquer servidor indistintamente. Caso a informação solicitada seja mantida por outro servidor ou servidores, o sistema encarrega-se de obter a informação necessária, de maneira transparente para o aplicativo, que passa a atuar consultando a rede, independentemente de conhecer seus servidores. Exemplos típicos são bases de dados corporativas, em que o volume de informação é muito grande e deve ser distribuído por diversos servidores. Porém não é dependente de aspectos lógicos da carga de acesso aos dados, ou bases de dados fracamente acopladas, em que uma informação solicitada vai sendo coletada numa propagação da consulta numa cadeia de servidores. A característica básica é a existência de diversos programas aplicativos consultando a rede para acessar os dados necessários, sem o conhecimento explícito de quais servidores dispõem desses dados.

O grande desenvolvimento de SGBD's relacionais tem permitido a implementação e disponibilidade comercial de SGBD's com arquitetura cliente-servidor e SGBD's distribuídos. O mesmo não se tem verificado com os SGBD's orientados a objetos (ou gerenciadores de objetos). No caso do desenvolvimento de gerenciadores de objetos distribuídos, o próprio paradigma de orientação a objetos conduz naturalmente a uma distribuição tanto do processamento quanto do armazenamento dos objetos, o que leva a uma importância redobrada dos aspectos de projeto de um sistema, para que a distribuição inerente a uma aplicação possa ser adequadamente modelada e suportada pelo sistema uma vez informatizado.

4.2 Bases de Dados Distribuídas

Özsu e Valduriez apresentam em [OV99] uma taxonomia que classifica as possíveis alternativas para a distribuição de bases de dados ao longo de três dimensões:

Autonomia: refere-se à distribuição do controle e indica o grau em que SGBD's individuais podem operar independentemente, de acordo com a consideração de alguns fatores como a necessidade de interoperabilidade dos sistemas componentes e a execução de transações de forma independente por parte desses sistemas;

Distribuição: refere-se ao local de armazenamento físico dos dados, que podem estar armazenados em um único local ou de forma distribuída sobre vários locais;

Heterogeneidade refere-se à utilização de SGBD's distintos ou não para gerenciamento dos dados, ou seja, SGBD's heterogêneos ou homogêneos, respectivamente.

A necessidade de ter-se um controle de transação global em bases de dados distribuídas, faz com que seja necessário o entendimento da "essência de autonomia", pois quanto mais independentes forem os sistemas componentes, menor é a necessidade de um controle de transação global. A maioria dos sistemas assume a existência de nenhuma autonomia ou total autonomia. A taxonomia proposta por Özsu e Valduriez apresenta uma visão diferenciada do conceito de autonomia, considerando a existência de três tipos de autonomia: forte integração, semi-autonomia e total autonomia (ou total isolamento), que correspondem, respectivamente, a três tipos de sistemas:

- *Sistemas fortemente integrados:* uma visão única de toda a base de dados é disponibilizada aos usuários que querem compartilhar a informação que pode estar fisicamente armazenada em múltiplas bases de dados;

- *Sistemas semi-autônomos:* consistem de SGBD's que podem operar independentemente, mas que participam de um conjunto de sistemas para permitir o compartilhamento de seus dados;

- *Sistemas totalmente isolados:* os componentes individuais são SGBD's stand-alone que desconhecem a existência de outros SGBD's, bem como a maneira de se comunicar com eles.

Entretanto, entre nenhuma autonomia e total autonomia provavelmente existem muitos níveis intermediários, devendo-se delineá-los e definí-los precisamente, identificando-se o grau de compartilhamento de dados existente em cada um deles e discutindo-se modelos de transação apropriados para cada um dos níveis de autonomia delineados.

4.2.1 Projeto de Distribuição de Base de Dados

O desenvolvimento sistemas de bases de dados distribuídas surgiu da necessidade de se compartilhar dados, usados em locais distintos. Portanto, pressupõe-se a existência de um conjunto de unidades computacionais interligadas (nós), cada uma contendo um SGBD, idênticos ou não, capazes de processar as transações locais (operações sobre a base de dados local) e as transações globais que solicitam o acesso a dados em outros nós.

Por distribuição de dados, em bases de dados relacionais, entende-se: a *replicação* de duas ou mais cópias do mesmo item de dado, alocados em nós distintos; ou o *particionamento* de relações divididas em conjuntos disjuntos e alocados em nós específicos, sem duplicação da base de dados.

O projeto de distribuição de dados é de fundamental importância para o bom desempenho de um sistema de bases de dados distribuídas. Existem duas abordagens de projeto de distribuição de dados para bases de dados relacionais: *descendente* e *ascendente*[1], sendo que na literatura existem muitos trabalhos realizados dentro de cada uma delas.

A abordagem descendente

A abordagem de projeto de distribuição *descendente* assume que o projeto da base de dados distribuída parte da estaca zero. Deste modo, nesta abordagem, o projetista deve realizar as quatro fases de projeto já existentes no método de projeto de bases de dados centralizadas (análise de requisitos, projeto conceitual, lógico e físico), além de uma fase adicional específica de projeto de bases de dados distribuídas (projeto de distribuição).

Análise de requisitos: consiste em entender os requisitos especificados informalmente pelo usuário, produzindo uma definição não ambígua e classificando os elementos a serem considerados no projeto da base de dados;

Projeto Conceitual: consiste em produzir, ignorando detalhes de implementação (em particular, distribuição de dados), o esquema conceitual global da base de dados e das aplicações que irão interagir com essa base;

Projeto Lógico: consiste em transformar o esquema conceitual global da base de dados em um esquema específico para um dado tipo de SGBD;

Projeto Físico: é realizado de acordo com as características e os recursos do SGBD, produzindo a definição das estruturas físicas de acesso à base de

[1] em inglês os termos são respectivamente top-down e bottom-up

dados;

Projeto de Distribuição: consiste em dividir o esquema global de base de dados em subesquemas, possivelmente sobrepostos, onde cada um representa um subconjunto de informações associadas a cada localização da base de dados distribuída. A princípio, esta fase pode ser aplicada a qualquer um dos esquemas globais (conceitual, lógico ou físico). A escolha de quando realizar esta fase está sujeita ao seguinte argumento: "Detalhes sobre implementação devem ser decididos apenas após ser feita a distribuição da base de dados, permitindo realizar o projeto físico de cada base de dados local independentemente. Projeto físico independente é obrigatório se os SGBD's de cada local forem heterogêneos. Por outro lado, uma descrição precisa de dados e operações ajudam a estimar a performance de distribuições". Este argumento sugere que o projeto de distribuição deve ser realizado no início da fase de projeto lógico, quando dados e operações são precisamente descritos e os primeiros problemas de implementação são considerados.

A distribuição da base de dados requer determinar a *fragmentação* e a *alocação* de dados. A fragmentação é o processo de subdividir um objeto global (entidade ou relação) em vários pedaços, chamados de fragmentos. A alocação é o processo de mapear cada fragmento a um ou mais locais. Cada fragmento deve ser definido como uma coleção homogênea de informações do ponto de vista de transações de acesso, isto é, tal que todas as instâncias dos fragmentos sejam uniformemente acessadas pelas transações. Dois tipos de fragmentos são possíveis:

- *Fragmentos horizontais:* são formados por subconjuntos de instâncias de um objeto global. Cada fragmento está associado a um predicado (chamado qualificação) que indica a propriedade específica das instâncias daquele fragmento;

- *Fragmentos verticais:* são obtidos através da projeção de objetos globais em subconjuntos de seus atributos. Cada fragmento deve incluir o atributo chave do objeto global ou, pelo menos, um identificador de tupla interno.

O objetivo da fragmentação horizontal é produzir fragmentos com o potencial máximo de localidade em relação as operações. De modo que cada fragmento esteja localizado onde é mais freqüentemente usado. Deve ser considerada a possibilidade de não se particionar um objeto global, pois a sobrecarga devido a fragmentação horizontal pode não ser compensada pela vantagem do processamento local.

O objetivo da fragmentação vertical é agrupar atributos que são freqüentemente usados de forma conjunta. Uma fragmentação vertical ideal existe quando cada fragmento é usado por apenas uma aplicação. De forma geral, deve-se balancear os benefícios (devido a possibilidade de se posicionar cada fragmento perto da aplicação que o usa mais freqüentemente) e as desvantagens (devido a um fragmento ser acessado por mais de uma aplicação).

No processo de fragmentação, o projetista aplica repetidamente a fragmentação horizontal e vertical para cada objeto até que os fragmentos apropriados sejam obtidos.

A *alocação* de fragmentos pode ser: *não-redundante*, onde cada fragmento é mapeado para exatamente um local; ou *redundante*, onde cada fragmento é mapeado para um ou mais locais.

Na alocação redundante o projetista deve decidir o grau de replicação de cada fragmento. O benefício da replicação cresce com a razão entre leitura e escrita, pois manter a consistência da base de dados requer a atualização de todas as cópias. Entretanto, o sistema pode permitir inconsistências temporárias e, neste caso, a replicação torna-se conveniente. O benefício da replicação cai com o aumento do custo de armazenamento, pois cópias requerem mais espaço. Entretanto, a replicação possibilita a recuperação de falhas, já que a perda de várias cópias independentes ao mesmo tempo é improvável e as aplicações podem acessar cópias alternativas quando alguma falha afeta as cópias usualmente acessadas.

A abordagem *descendente* para o projeto de distribuição consiste em resolver, para cada objeto global, os problemas de fragmentação horizontal e vertical e alocação redundante e não-redundante, os quais estão claramente relacionados entre si.

De acordo com a taxonomia de Özsu e Valduriez, apresentada anteriormente, a abordagem *descendente* se aplica ao projeto de distribuição de sistemas fortemente integrados, em ambiente de gerenciamento de dados homogêneo ou heterogêneo.

A abordagem ascendente

A abordagem de projeto de distribuição *ascendente* assume que as bases de dados de cada local já existem, devendo-se integrá-las produzindo uma especificação global. Deste modo, nesta abordagem, o ponto de partida para o projeto são os esquemas representando a porção de dados armazenada em cada local, e o projeto de distribuição consiste em identificar os dados comuns a cada esquema, bem como suas diferenças.

Os problemas do projeto de distribuição ascendente estão na construção do esquema global de dados, com os seguintes passos para construí-lo:

1. *Reconhecer similaridades:* consiste em reconhecer o "casamento" (match) entre entidades e seus atributos, identificando porções de esquemas que se sobrepõem devido a similaridades no nome ou estrutura ou, então, através da análise dos dados de bases de dados ou arquivos pré-existentes;

2. *Reconhecer conflitos:* consiste em identificar representações ou definições de domínios diferentes para dados similares em esquemas distintos. Pode-se resolver conflitos incorporando-se as diferenças no esquema global ou fazendo-se ajustes nos esquemas locais. Os conflitos que não podem ser solucionados na fase de projeto requerem o uso de políticas de respostas de consultas para dados inconsistentes. Os conflitos incluem:

   - *conflitos de nome:* existem dois tipos, os sinônimos, quando objetos de dados que representam os mesmos objetos do mundo real tem nomes diferentes em esquemas distintos, e homônimos, quando objetos de dados com o mesmo nome em esquemas distintos representam diferentes objetos do mundo real. Esses conflitos podem ser solucionados armazenando-se uma tabela de correspondência de nomes no esquema global;

   - *diferenças de domínio:* ocorre quando entidades em esquemas distintos se sobrepõem parcialmente, ou seja, existem atributos que são comuns a todas essas entidades e outros que são específicos de algumas delas. A solução, neste caso, é a utilização de uma hierarquia de generalização no esquema global, onde os atributos comuns ficam associados ao supertipo e para cada entidade que contenha atributos específicos é gerado um subtipo;

   - *diferenças de escala:* podem ocorrer em diferentes visões do mesmo valor numérico. Neste caso, o dado deve ser recuperado usando-se, se possível, a escala mais precisa, sendo convertido para a escala desejada através de fórmulas de conversão;

   - *diferenças estruturais:* ocorrem devido a diferentes escolhas de projeto, por exemplo, um mesmo objeto do mundo real pode ter sido modelado como um atributo em um esquema e como uma entidade em outro. Neste caso, o ideal seria modificar um ou ambos os esquemas. Entretanto, quando isso não for possível, deve-se recorrer a procedimentos de modificação de consultas complexos.

O gerenciamento de fórmulas de conversão, procedimentos de modificação de consultas ou políticas para respostas de consultas para dados inconsistentes, requerem uma extensão das capacidades tradicionais dos SGBD's. Caso o SGBD a ser utilizado não tenha essas capacidades, deve-se implementá-las nos aplicativos.

3. *Tratar dados inconsistentes durante operação:* operacionalmente multibases tem erros devido, por exemplo, a falhas de sincronização de atualizações e recuperação inadequada de erros de sistema. Deste modo, o projetista da base de dados deve decidir as políticas a serem utilizadas para tratar inconsistências que surgem durante a operação da base de dados global. As políticas incluem:

   - apresentar qualquer um dos valores inconsistentes sem notificar o usuário: é a solução mais direta e ao mesmo tempo a mais perigosa;
   - apresentar todos os valores inconsistentes e mostrar ao usuário as fontes da informação: neste caso o usuário deve estar apto a avaliar as causas de inconsistências;
   - avaliar alguma função agregada (média, mínimo, máximo) e apresentar o resultado da função ao usuário;
   - apresentar o valor mais recente: requer a marcação de tempo das operações de atualização;
   - apresentar o valor do sistema mais seguro: assume que o projetista está apto a avaliar a confiabilidade de cada local da base de dados distribuída.

De acordo com a taxonomia de Özsu e Valduriez, apresentada anteriormente, a abordagem ascendente se aplica ao projeto de distribuição de sistemas com múltiplas bases de dados com um certo grau de autonomia, pois envolve a integração de bases de dados existentes. É o caso de um sistema, no qual as bases de dados componentes participam de um conjunto de bases de dados homogêneas ou não para permitir o compartilhamento de seus dados. Essas bases são conhecidas pelo termo bases de dados federadas. Assim, essas bases de dados podem ser gerenciadas por SGBD's que podem operar independentemente, havendo também a necessidade de um certo grau de compartilhamento de dados apoiados por SGBD's *federados.*

Interação entre as abordagens *ascendente* **e** *descendente*

Na abordagem descendente o projetista ignora qualquer detalhe físico, incluindo distribuição, durante o projeto conceitual. Entretanto, existem situações nas quais isto não é apropriado. Por exemplo, uma empresa que é organizada por funções, cada uma sendo diretamente mapeada a uma base de dados local. Neste caso, não é natural ignorar informações sobre distribuição. Uma possível abordagem é proceder parcialmente descendente e parcialmente ascendente, realizando o projeto conceitual para cada função independentemente, em seguida integrando os esquemas conceituais em um esquema global e, finalmente, redistribuindo a informação (para introduzir replicação e "mover" alguns objetos de um esquema local para outro).

A abordagem descendente é conveniente em situações onde nas fases iniciais do projeto não se tem uma correspondência direta entre entidades e sua distribuição. A abordagem ascendente é conveniente quando nas fases iniciais do projeto já se tem uma correspondência imediata entre tabelas e cada local da base de dados distribuída.

4.2.2 Bases de Dados Heterogêneas

Num ambiente de distribuição de bases de dados, a heterogeneidade pode ocorrer de várias formas: diferenças de hardware, diferenças no protocolos de rede, vários gerenciadores de dados e vários modelos lógicos de dados. A concepção *ascendente* considera a existência de múltiplas bases de dados com esquemas de dados independentes e autônomos. Assim é muito comum a utilização da metodologia *ascendente* para a solução de problemas de distribuição de bases de dados em ambientes heterogêneos.

Do ponto de vista de modelos lógicos de dados, a heterogeneidade é solucionada através da criação de um esquema global de dados, capaz de representar os esquemas parciais das bases de dados heterogêneas. Entretanto, um importante obstáculo a ultrapassar são os valores semânticos dos elementos utilizados em cada esquema de dados. Um simples conceito pode ter diferentes significados em dois esquemas de bases de dados heterogêneas, como por exemplo o elemento endereço: pode ser definido como atributo, numa base de dados cadastral de uma empresa; ou como entidade numa base de dados do correio. Estes problemas são típicos de ambigüidade semântica e incompatibilidade de especificação de projetos ou ambigüidade estrutural. Tais ambigüidades podem ser classificadas em: escala (mesmo atributo é armazenado em diferentes unidades de medida), nível de abstração (diferentes níveis de detalhes), nome (atributos diferentes com mesmo nome ou atributos iguais com nomes diferentes), tipo, tamanho, etc. Uma vez identificada a heterogeneidade, o modelo de dados deve suportar um tratamento de forma a gerar uma representação única no esquema global, garantindo a consistência das ocorrências.

O esquema global de um sistema de banco de dados heterogêneos deve ter as seguintes características

1. *completude e corretude:* o esquema global deve conter todos os conceitos presentesem algum esquema componente. O esquema global deve ser a representação da união dos domínios das aplicações associadas com os esquemas;

2. *minimalidade:* se o mesmo conceito é representado em mais de um esquema componente, ele tem que ser representado apenas uma única vez no esquema global, e deve estar relacionado com as várias ocorrências;

3. *compreensão:* o esquema global deve ser fácil de entender para o projetista e para o usuário. Dentre as várias representações possíveis de resultado de integração permitidas por um modelo de dados, somente a mais fácil de ser compreendida deve ser escolhida.

Assim, a constatação das incompatibilidades e ambigüidades entre os dados deve ser considerada natural, pois os componentes do banco de dados foram implementados de forma independente.

Entretanto outras abordagens, para permitir a integração das bases de dados heterogêneas não pressupõe a criação de um esquema global de dados. Para garantir a troca de informação entre as bases de dados é sugerido a criação de componentes específicos de software para garantir a interoperabilidade entre as bases de dados envolvidas.

Tal abordagem considera a utilização de domínios de integração específicos com os componentes de integração (*wrapper*, *mediator*, etc).

Para cada situação de integração de bases de dados heterogêneas, os componentes da arquitetura são instanciados e implementados para universos específicos de integração. Nessa abordagem é mantido a completa autonomia das bases de dados heterogêneas com uma estrutura pontual de integração para cada necessidade.

A criação do esquema global de dados e a construção de componentes específicos para integração de bases de dados heterogêneas são duas classes de alternativas de solução, disponíveis na literatura, para a integração de dados em sistemas de informação.

4.2.3 Um exemplo de Metodologia de Distribuição

Desenvolvida no Instituto Politecnico di Milano, a metodologia DATAID-D para projeto de bases de dados distribuídas, segundo a abordagem descendente, provê uma estrutura metodológica para o projetista, indicando os problemas de projeto relevantes, que parâmetros são necessários para solucioná-los, e como cada problema pode ser abordado.

A metodologia DATAID-D requer a inclusão de duas fases na arquitetura: *análise de requisitos de distribuição* e *projeto de distribuição.* A Figura 4.1 mostra todas as fases de projeto segundo a metodologia DATAID-D. As duas fases adicionais, que tratam da distribuição de dados propriamente dita, são apresentadas a seguir.

Análise de Requisitos de Distribuição:

tem por objetivo obter informações sobre distribuição, tais como predicados para fragmentação horizontal e a freqüência de ativação de cada aplicação em cada local. Essa fase utiliza os requisitos do usuário em relação a distribuição, o esquema global de dados e as operações construídas na fase de projeto conceitual, gerando três tabelas:

- *Tabela de freqüência:* contém o número de ativações de cada aplicação em cada local. Assume-se que todas as aplicações são potencialmente executáveis em todos os locais. Quando uma aplicação nunca é executada em um dado local sua freqüência é zero para esse local;

- *Tabela de fragmentação:* contém os critérios de fragmentação horizontal aplicados a cada entidade do esquema. São definidos dois tipos de fragmentação horizontal: primário, no qual a fragmentação de uma entidade E1 é expressa usando-se vários predicados disjuntos nos atributos de E1, e derivado, na qual a fragmentação de uma entidade E2 é determinado pelo relacionamento binário (do ME-R) entre E2 e outra entidade E1 que já é fragmentada;

- *Tabela de polarização:* indica, de forma quantitativa, como as partições influenciam a localidade de processamento de aplicações, ou seja, cada valor de polarização indica a probabilidade de um dado fragmento ser acessado por uma dada aplicação de um dado local. São indicados apenas os valores que não seguem uma distribuição uniforme.

Projeto de Distribuição:

tem por objetivo alocar os dados aos locais, a partir do esquema global de dados e das tabelas lógicas de acesso construídas na fase de projeto lógico global, e dos requisitos de distribuição obtidos na fase de análise de requisitos de distribuição. O esquema global da base de dados é descrito usando

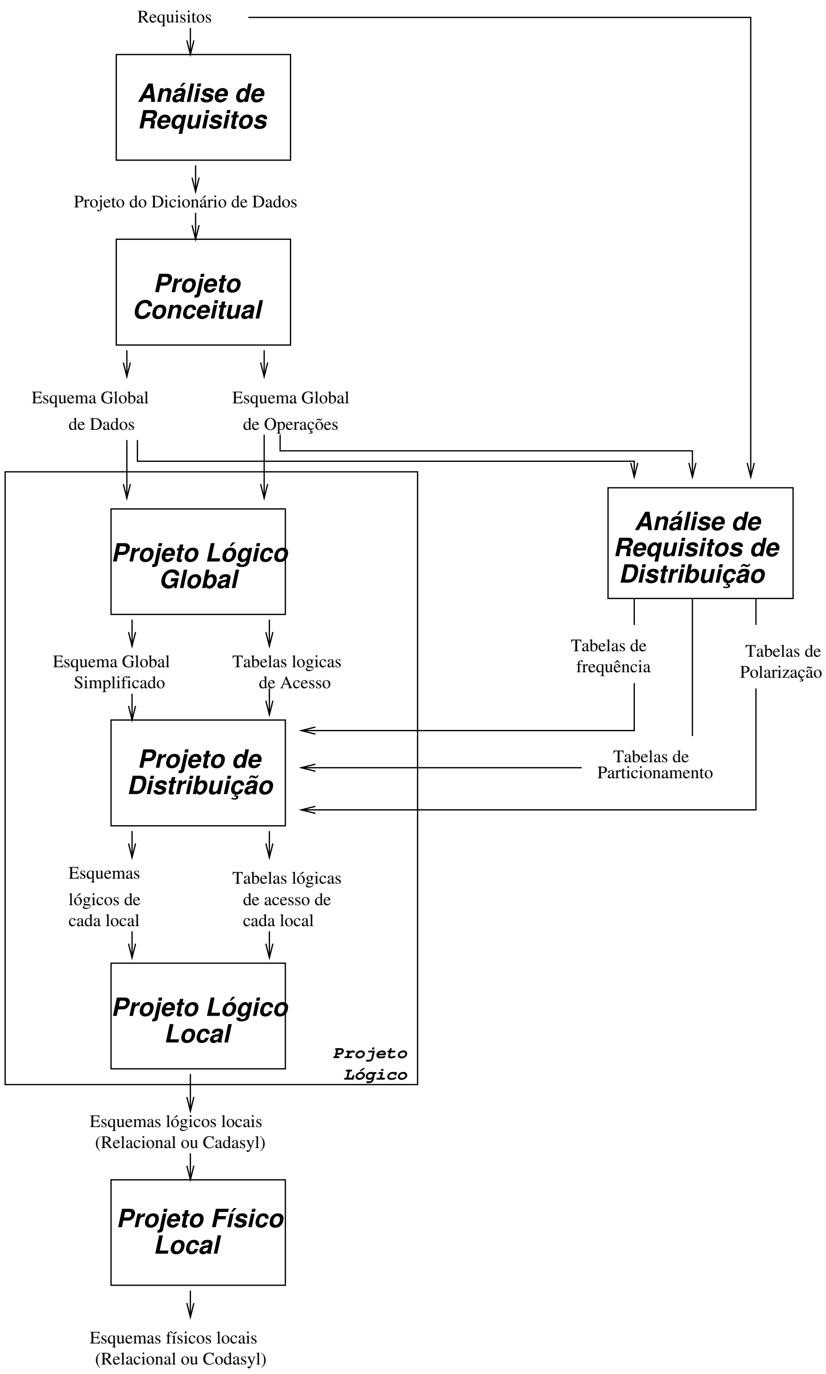


Figura 4.1: Metodologia DATAID-D

uma versão simples do Modelo Entidade-Relacionamento (ME-R), que inclui apenas relacionamentos binários (a primeira parte da fase de projeto lógico, ou seja, a fase de projeto lógico global, deve simplificar o esquema global de dados obtido na fase de projeto conceitual, pois esse esquema pode ter sido construído utilizando o ME-R Estendido), e as tabelas lógicas de acesso indicam o tipo das operações de acesso à base de dados (leitura/escrita) realizadas por cada aplicação em cada entidade. A fase de projeto de distribuição gera um esquema lógico de dados e tabelas lógicas de acesso para cada local. O projeto de distribuição na metodologia DATAID-D é subdividido em quatro fases:

- *Projeto de fragmentação:* aplica fragmentação horizontal e vertical nas entidades com o objetivo de determinar possíveis unidades de alocação para as fases subseqüentes. A grosso modo, para um fragmento ser uma boa unidade de alocação, deve conter instâncias que são acessadas com a mesma freqüência por cada aplicação executada em cada local. Obviamente, uma aplicação rigorosa desse critério resultaria na fragmentação caindo a nível de atributos ou tuplas individuais. Entretanto, existe um limite além do qual uma fragmentação adicional não é factível. Mais freqüentemente, o projeto de fragmentação é uma decisão lógica realizada através da seleção de predicados das tabelas de polarização e da composição destes dentro de fragmentos definidos logicamente;

- *Alocação não-redundante:* realizada através do mapeamento de cada fragmento ao local onde é mais freqüentemente usado. A freqüência de uso de cada fragmento em cada local é obtida pela somatória, para todas as transações emitidas daquele local, do produto entre polarizações e freqüências de uso do fragmento. Deste modo, é possível identificar o local que acessa o fragmento mais freqüentemente, e alocar o fragmento ao local;

- *Alocação redundante:* realizada através da seleção de locais adicionais para cada fragmento inicialmente alocado de forma não-redundante, usando heurísticas "gananciosas". Antes de se fazer cópias de fragmentos, deve-se levar em consideração se os benefícios obtidos em relação a recuperação da informação, compensam os custos e complexidade gerados pela necessidade de atualização das cópias. Esses benefícios são difíceis de quantificar, pois é muito difícil medir a complexidade gerada pela manutenção das cópias. Entretanto, é possível avaliar a diferença entre o número de recuperações de informação que torna-se local devido a adição de uma cópia, e o número de atualizações remotas adicionais requeridas para a manutenção daquela cópia. O resultado deve ser muito positivo para justificar a inclusão de redundância;

- *Reconstrução de esquemas locais:* constrói esquemas locais a partir da versão final da alocação de fragmentos. Essa fase também é responsável pela alocação dos relacionamentos (do ME-R) do esquema global. Assumindo que a maioria dos relacionamentos são implementados como associações entre os identificadores das entidades correspondentes, a metodologia DATAID-D sugere posicionar os relacionamentos no mesmo local das entidades ou fragmentos com a maior cardinalidade, fazendo com que poucos identificadores de entidade tenham que ser transmitidos.

4.3 Fragmentação no Modelo Relacional

O projeto de fragmentação define as unidades lógicas de alocação, podendo ser horizontal (agrupamento de tuplas), vertical (agrupamento de atributos) ou mistas(tuplas e atributos), que tenham as mesmas propriedades do ponto de vista de acesso pelas aplicações, conforme ilustrado pela Figura 4.2.

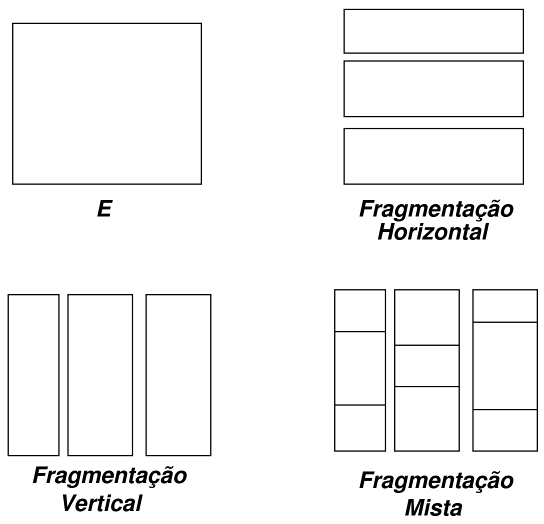


Figura 4.2: Tipos de Fragmentação

O *grau de fragmentação* é o número de operações de fragmentação aplicadas a tabela original. O grau pode variar de 0 (não há fragmentação) até n, no qual as n fragmentações aplicadas podem gerar uma única instância da entidade ou somente um atributo. Recomenda-se em [OV99] e [CP84] fragmentações até grau 2. Isto considerando-se que no caso do modelo relacional, as relações globais já estão normalizadas e a execução de muitas fragmentações verticais podem aumentar muito o custo das junções.

Para identificar a necessidade da fragmentação, é executado um processo de análise inicial que determinará se a relação será submetida ao processo de fragmentação.

O processo de análise verifica os benefícios da fragmentação, considerando o número de nós nos quais as transações originam, o volume de dados da relação e a opção do projetista. Essas informações são quantificadas da seguinte maneira:

a) Número de nós diferentes que originam as transações independente do seu tipo (escrita ou leitura) que acessam uma determinada relação:

   1: transações só iniciam a partir de um único nó;

   n: transações originam de múltiplos nós.

b) Volume de dados da relação:

   p: o volume de dados é pequeno (fragmentação deve ser evitada);

   c: o volume de dados não é pequeno, portanto deve ser considerado.

A classificação para o volume de dados das relações (pequena ou considerável) é subjetiva, porém será adotado como critério a proporção do volume de dados alvo da fragmentação em comparação com o volume de dados total das relações da base de dados de acordo com o seguinte critério:

i) Calcula-se o volume total de dados das relações (VT),

ii) Se o volume da relação analisada (V_a) para o processo de fragmentação for inferior [2] a 10% do volume total, então seu volume pode ser considerado pequeno, caso contrário é considerável para fragmentação .

[2] A percentagem mínima de volume da relação pode ser fornecida pelo projetista, como um parâmetro de configuração, caso contrário o valor de 10% é normalmente assumido.

4.3.1 Modelo de fragmentação horizontal

Inicialmente é necessário analisar qual o tipo de fragmentação horizontal será aplicada na relação: *primária ou derivada.* As informações do esquema de dados que deverão ser incluídas pelo projetista para propósito do projeto de distribuição. As informações são características entre relações, denominadas de vínculos em [OV99]. Os termos primária ou derivada são determinadas através dos conceito de vínculo entre as entidades do modelo ER. O conceito de vínculo melhor caracteriza o relacionamento entre as entidades.

Exemplo: considere-se as relações:

DEPARTAMENTO (DEP_ID, NOME, LOCALIDADE)

EMPREGADO (EMP_ID, NOME, DEP_ID, FUNÇÃO)

existe um vínculo V_1, de cardinalidade 1 - N entre essas duas relações, representado na Figura 4.3

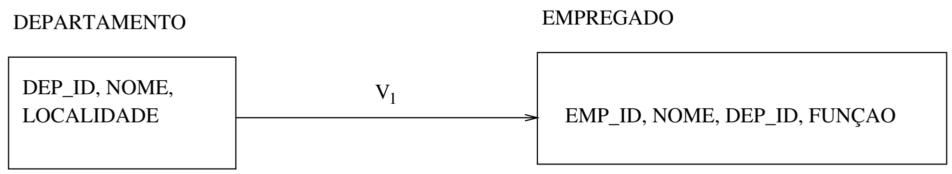


Figura 4.3: Vinculo de Cardinalidade

A relação que é origem do vínculo é chamada de *proprietária* (Departamento) e a relação destino do vínculo é denominada *membro* (Empregado).

A inclusão de um vínculo entre relações sugere a necessidade de fragmentar a relação membro de acordo com os critérios da fragmentação horizontal da relação proprietária, considerando-se também a necessidade das aplicações manipularem simultaneamente as duas relações, através de operações de junções.

Assim a fragmentação horizontal derivada será aplicada somente nas relações definidas como membros pelo projetista. Nas demais relações será aplicada a fragmentação horizontal primária.

4.3.2 Procedimento de Fragmentação Horizontal Primária

O procedimento a seguir tem por objetivo identificar os fragmentos horizontais primários das relações, definindo um conjunto de predicados (minitermos) que representam uma condição de seleção dos fragmentos. Os minitermos seguem as propriedades de serem mínimos e completos.

Definição 4.3.1 Dado uma relação $R(A1, A2, ..., An)$ onde Ai é um atributo definido sobre o domínio Di, um predicado simples definido em R tem a forma:

- pj: Ai *operador* Valor;
- *operador* $\in$ ao conjunto $\{ =, <, >, \geq, \leq, \neq \}$;
- Valor $\in$ Di;

Definição 4.3.2 Seja Pri predicados sobre a relação Ri, os membros de Pri são denotados por pij da forma:

$Pri = \{pi1, pi2, ..., pim\}$

Definição 4.3.3 Seja $Pri = \{pi1, pi2, ..., pim\}$ os predicados simples da relação Ri. A coleção de predicados minitermos $Mi = \{mi1, mi2, ..., miz\}$ é definido da seguinte forma :

$Mi = \{mij/mij = \wedge pi * k\}$, $1 \leq$ k $\leq$ m, $1 \leq$ j $\leq$z,
onde: $pi * k = pik$, ou $pi * k = \neg pik$, $\wedge$ é a combinação pik de Pri.

Considerando a Figura 4.3 predicados da entidade Empregado:
p1: função " Engenheiro elétrico"
p2: função " Analista"

Suponha que tivessemos na relação Empregado um atributo *salário*. De forma que os predicados p3 e p4 sejam descritos da seguinte forma:
p3: salário $\leq$ 2500
p4: salário $>$ 2500

Um pequeno exemplo de possibilidades dos minitermos seria:

- m1 : função = " Engenheiro elétrico" $\wedge$ salário $\leq$ 2500;
- m2 : função = " Engenheiro elétrico" $\wedge$ salário $>$ 2500;

- m3 : $\neg$ função = " Engenheiro elétrico" $\wedge$ salário ≤ 2500;
- m4 : $\neg$ função = " Engenheiro elétrico" $\wedge$ salário > 2500;

Os parâmetros importantes para o processo de fragmentação são:

- Relação a ser fragmentada (R);
- Conjunto de predicados relevantes das aplicações (P) identificados pelo projetista;
- Conjunto de regras de implicações entre os predicados relevantes (I), identificados pelo projetista;
- Freqüência de acesso mínima (FAM) a ser considerado no processo de fragmentação;
- Número de tuplas mínimo (NTM) do fragmento gerado.

O procedimento de fragmentação será demonstrado a partir de um exemplo. Considere a relação EMPREGADO(ID_EMP, NOME, DEPARTAMENTO, FUNÇÃO). Detalhes de tal procedimento e os algoritmos são encontrados em [OV99]. Os predicados relevantes de cada uma das transações consideradas são:

- T1: DEPARTAMENTO = "Informática" (predicado P1);
- T1: FUNÇÃO = "Desenvolvedor" (predicado P2);
- T2: DEPARTAMENTO = "Compras" (predicado P3);

Passo 1

Construir a matriz de utilização de predicados (MP), que representa nas linhas as transações e nas colunas os predicados relevantes. Assim cada elemento dessa matriz indica quais são os predicados utilizados pelas transações. Para cada transação é fornecido dados a respeito da freqüência de acesso. Exemplo: Considere-se a tabela a seguir representando os predicados acessados pelas transações.

Transações	Predicados			Freq. Acesso
	P1	P2	P3	
T1	1	0	0	20
T1	0	1	0	20
T2	0	0	1	50

Passo 2

Determinar o conjunto M_1 contendo todos os possíveis minitermos, a partir dos predicados fornecidos. Exemplo: o conjunto M_1 a seguir representa todas as combinações possíveis do conjunto de predicados $\{p_1,\ p_2$ e $p_3\}$

$$\begin{aligned} M_1 \quad = \{ \quad & p_1 \wedge p_2 \wedge p_3, \quad & p_1 \wedge p_2 \wedge \neg p_3, \quad & p_1 \wedge \neg p_2 \wedge p_3, \\ & p_1 \wedge \neg p_2 \wedge \neg p_3, \quad & \neg p_1 \wedge p_2 \wedge p_3, \quad & \neg p_1 \wedge p_2 \wedge \neg p_3, \\ & \neg p_1 \wedge \neg p_2 \wedge p_3, \quad & \neg p_1 \wedge \neg p_2 \wedge \neg p_3\} & \end{aligned}$$

Passo 3

Determinar o conjunto de minitermos não contraditórios denominado M_2, que será obtido removendo-se elementos do conjunto de combinações todos os possíveis minitermos M_1, utilizando-se o conjunto de regras de implicações I.

Exemplo: considere-se fornecido o conjunto I:

$$\{\ p_1 \Rightarrow \neg p_3,\ p_3 \Rightarrow \neg p_1\ \},$$

então o conjunto M_2 será:

$$M_2 = \{p_1 \wedge p_2,\ p1 \wedge \neg p_2,\ p_3 \wedge p_2,\ p_3 \wedge \neg p_2\}$$

Passo 4

Selecionar os mini-termos do conjunto M2, de acordo com os seguintes critérios:

- *Freqüência de acesso mínima (FAM)* : Identificar na matriz de acesso, qué uma combinação das frequências descritas na matriz de utilização de predicados (MP), o total de acessos em cada minitermo (TA), e verificar se $TA > FAM$;

- *Número de tuplas do fragmento gerado*: calcular o número de tuplas médio (NT), verificando se $NT > NMT$.

4.3.3 Procedimento de Fragmentação Horizontal Derivada

O procedimento a seguir define fragmentos horizontais de relações membros de outras relações. Uma relação pode ser membro de mais de uma relação, assim o procedimento deve selecionar uma relação proprietária e gerar as expressões que definem os fragmentos horizontais secundários das relações membro. Como exemplo, na Figura 4.3, o atributo *DEP-ID* é um vínculo entre a entidade Empregado e Departamento.

Os seguintes parâmetros serão necessários para a fragmentação de uma relação membro M:

- Conjunto de Vínculos (V) da relação membro: $V = \{V_1, V_2, \ldots, V_n\}$,
- $1 \leq i \leq n$, no qual membro $(V_i) = R$;
- Conjunto de relações proprietárias da relação membro: $Pr = \{PV_1, PV_2, \ldots, PV_n\}$, $1 \leq V_j \leq V_n$, $PV_j = \mathit{propriet\acute{a}rio}(V_j)$;
- A freqüência de acesso das transações que acessam simultaneamente a relação membro (M) e as relações proprietárias (Pr). Exemplo:

Transação	P_{V1}	P_{V2}	M	Freq. Acesso	Nó Origem
T_1	1	0	1	30	1
T_2	0	1	1	60	2
Total	30	60		90	

De posse da tabela acima é preciso :

i) Selecionar a relação proprietária a ser utilizada: será selecionada a relação proprietária que tenha maior freqüência de acesso ($P_V i$);

ii) Gerar os fragmentos horizontais derivados: Caso a relação selecionada não tenha sido fragmentada horizontalmente, então não haverá a fragmentação horizontal de M. Caso contrário, os fragmentos horizontais derivados serão definidos por: $M_j = M$ SJ P_{Vij}, onde P_{Vij} é o fragmento j da relação proprietária P_{Vi}, e SJ é a aplicação de uma operação semi-join da álgebra relacional.

4.4 Método de Fragmentação Vertical

O método de fragmentação vertical descrito nesta seção é baseado no modelo de grafos definido por Navathe em [NCW], em detrimento ao modelo baseado em matriz de agrupamentos apresentado em [OV99]. O modelo de Navathe simplifica o processo de obtenção de cada fragmento, pois não é necessária a utilização de função de custo usada nas técnicas de agrupamentos de atributo, além do fato da complexidade do algoritmo ser de $O(n^2)$, enquanto que nas técnicas de agrupamentos de atributos é de $O(n^2 logn)$.

Os parâmetros a serem utilizados pelo método são:

i) Conjunto de atributos da relação $R : a_1, a_2, \ldots, a_n$;

ii) Conjunto de atributos chaves da relação: $K : k_1, k_2, \ldots, k_m,\ m \leq n, K \subset R$;

iii) Conjunto de atributos não chaves, indicados pelo projetista a estarem replicados nos fragmentos, $C : c_1, c_2, \ldots, c_j,\ j \leq n$ e para $1 \leq i \leq j, c_i \notin K, C \subset R$;

iv) Matriz de utilização dos atributos U;

v) Freqüência de operações de alterações em cada atributo de K e C;

vi) Porcentagem máxima de operações de alterações em atributos replicados PM;

Transação	Atributo					Freqüência de Ativação	Nó de origem
	1	2	3	4	5		
T_1	1	1	1	0	0	Acc = 40	1
T_2	0	0	1	1	0	Acc = 15	2
T_3	0	0	0	1	1	Acc = 70	3
T_4	0	1	0	0	0	Acc = 10	3
T_5	0	0	0	1	0	Acc = 5	8

Nessa matriz estão identificadas as transações que fazem acesso a relação, as freqüências de ativação, além dos atributos manipulados. Os atributos utilizados pela transação são indicados pelo valor "1", e caso contrário por "0".

A seguir serão detalhados os passos do método, através de um exemplo.

Passo 1: Análise da viabilidade da fragmentação: para analisar a conveniência da aplicação da fragmentação vertical, é importante verificar se é alta a taxa de operações de alteração nos atributos a serem replicados ($K \bigcup C$). Para isso será necessário identificar o quanto representam as operações de alterações em relação ao total de operações efetuadas, de acordo com os seguintes cálculos.

Para cada atributo replicado (a_r), calcula-se a percentagem de alterações (PA_r), em relação ao total de operações executadas (leitura e alteração) na relação. Assim se $PA_r \geq PM$, será considerada alta a taxa de alterações e finalizado a fragmentação vertical, caso contrário continua-se o passo 2.

Passo 2: Construção da Matriz de Afinidades: esta matriz identifica, para cada par de atributos, a freqüência de acesso simultâneo realizado por todas as transações. Exemplo:

Atributo	1	2	3	4	5
1	40	40	40	0	0
2	40	50	40	0	0
3	40	40	40	15	0
4	0	0	15	90	70
5	0	0	0	70	70

A matriz de afinidades define um grafo, onde cada nó representa um atributo e os arcos indicam os valores das afinidades. O grafo correspondente será:

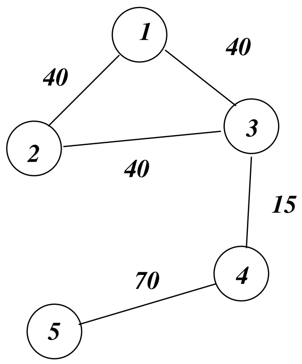


Figura 4.4: Grafo de afinidades

Passo 3: Identificação das partições: Para cada nó do grafo são efetuadas sucessivas interações incluindo-se nós a um ciclo do grafo (fragmentos) que tenham o valor de afinidade (indicado no arco) maior ou igual aos valores de afinidades já incluídos. Se não existir outro nó com essa característica é definido um fragmento e continua-se a análise anterior para outro nó que não tenha sido incluído em nenhum outro fragmento. A Figura 4.5 ilustra a aplicação do método.

De acordo com o grafo anterior foram gerados dois fragmentos formados pelos nós (1, 2, 3) e pelos nós (4, 5).

Passo 4: Inclusão dos atributos adicionais. Neste passo são incluídos os atributos chaves e os atributos indicados pelo projetista para serem replicados nos fragmentos.

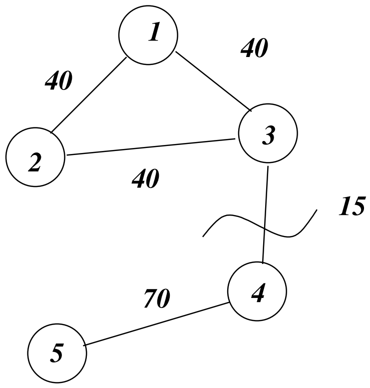


Figura 4.5: Grafos particionados

4.5 Alocação

O modelo de custo tem por objetivo encontrar uma solução que minimize o tempo de resposta das aplicações e os custos de execução da aplicação.

Os fatores a serem considerados estão relacionados com a eficiência de processamento das aplicações, relacionados a seguir:

- *Armazenamento:* custo de manter um fragmento armazenado em um nó;

- *Processamento local:* custo de execução local para a recuperação e manipulação de cada fragmento que compõe a transação;

- Mecanismos de controle de concorrência;

- Movimentação de dados na rede para a recuperação de dados em fragmentos remotos ou a propagação de alterações em fragmentos replicados.

Além da minimização de custos o projeto de alocação deve considerar fatores identificados pelo projetista ligados a restrições das aplicações e do ambiente de distribuição, relacionados a seguir:

- Tempo máximo para execução da aplicação;

- Espaço disponível para armazenamento em cada nó;

- Taxa máxima de tráfego na rede disponível num período de tempo entre dois nós;

- Carga máxima de processamento disponível em um nó.

O método de alocação empregado gera um projeto inicial sem replicação, em seguida são adicionados cópias dos fragmentos, avaliando-se os benefícios e a redução dos custos dessa inclusão. Existem dois tipos de réplicas: com atualização imediata (síncronas) e com atualização postergada (assíncrona).

O método de alocação é aplicado para cada unidade de alocação. A unidade de alocação corresponde a:

- uma relação não fragmentada;

- relações fragmentadas: horizontal, vertical ou horizontal/vertical;

- agrupamentos de fragmentos ou relações ligados através de pseudo-relacionamentos especificados pelo projetista.

O Método de alocação geral é composto dos seguintes passos:

i) Alocação inicial sem replicação e sem considerar as restrições de espaço mínimo de armazenamento disponível em cada nó;

ii) Re-alocação inicial sem replicação, avaliando-se restrições de espaço mínimo disponível;

iii) Alocação de réplicas de cada fragmento maximizando os benefícios e a minimizando os custos de execução das aplicações;

iv) Re-alocação das cópias dos fragmentos de acordo com critérios de espaço de armazenamento.

De forma geral, o método de alocação e replicação de acordo com a Figura 4.6

A primeira fase analisa a alocação de fragmentos seguindo critérios que maximizam os benefícios e minimizam os custos do processamento das aplicações. A segunda fase tem por objetivo re-alocar os fragmentos de acordo com os restrições de armazenamento, movendo-se para outros nós, ou removendo fragmentos de forma a minimizar o impacto nos custos impostos pela alteração.

Critérios de Alocação, considerando-se somente as necessidades das aplicações:

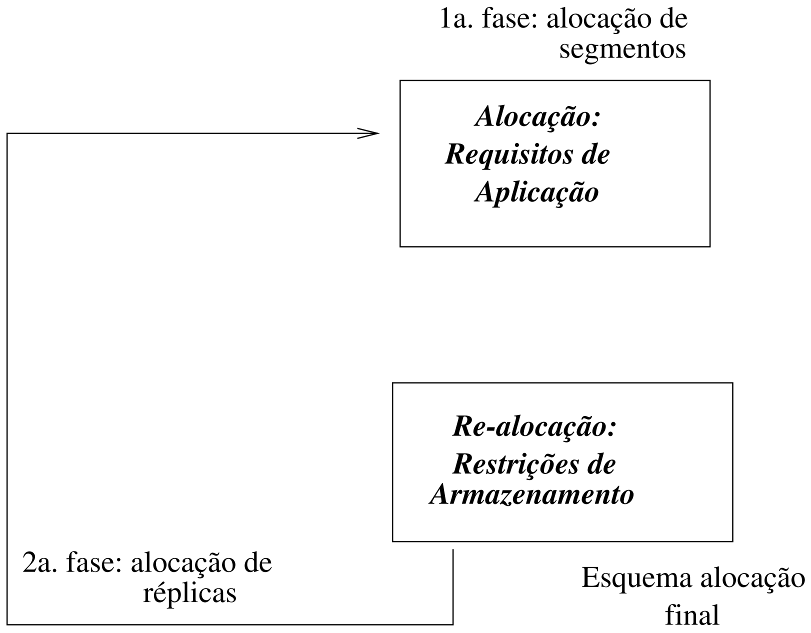


Figura 4.6: Método de alocação e replicação

i) Selecionar um fragmento que não tenha sido alocado;

ii) Calcular o custo de alocação do fragmento em cada um dos nós que executem transações que acessam o fragmento;

iii) Alocar o fragmento ao nó que tenha menor custo.

Este algoritmo aloca fragmentos somente nos nós que originam transações, pois considerou-se aqui a localidade do processamento como princípio do projeto de distribuição em detrimento a custos menores. Determinadas situações especiais no ambiente de distribuição, podem existir nós que não acessam o fragmento, mas disponibilizem recursos (rede, processamento remoto e armazenamento) que permitam o processamento mais eficiente (custos menores) do fragmento que em qualquer nó que origine a transação que acessa o fragmento.

Critérios para re-alocação, considerando-se as restrições de armazenamento:

i) Calcular o custo da alocação dos fragmentos de cada nó;

ii) Iniciando pelo nó que tenha maior custo até o nó que tenha menor, executar os seguintes passos:

   - Calcular o custo de alocação de cada fragmento no nó;
   - Enquanto o espaço de armazenamento for insuficiente, realocar o fragmento que tenha o menor custo de alocação, considerando apenas aqueles que tenham espaço de armazenamento disponível para o fragmento;

4.5.1 Modelo de Custos de Alocação

O modelo de custos a ser descrito a seguir tem por objetivo minimizar os custos de processamento das aplicações, os custos da movimentação de dados entre os nós e os custos de armazenamento de dados em cada nó, considerando as restrições relacionadas ao processamento das aplicações e ao espaço disponível para armazenamento, descrito através do seguinte modelo geral:

Custo total do projeto de alocação (CPA) = Custo total de processamento das transações (CPT) + Custo de armazenamento dos fragmentos (CAF):

$$CPA = \sum_{t \in T} CPT_t + \sum_{j \in N} \sum_{j \in F} CAF_{jk}$$

Objetivo: Minimizar (CPA)
Restrições:

- Espaço de armazenamento disponível em bytes no nó k: EAD_k;
- Carga de processamento máxima em MIPS (Milhões de Instrução por Segundos) no nó k: CPM_k;
- Tempo máximo de processamento em segundos da transação t: TMP_t.

Na expressão de custos CPT_t é o custo do processamento da transação t e CAF_{kj} é o custo de alocação do fragmento j ao nó k.

Parte II

Abordagem Não-Clássica

Capítulo 5

Controle de Concorrência não ACID

5.1 Hipóteses Implícitas nas Transações ACID e as Aplicações que as Violam

À medida que os bancos de dados deixaram de ser usados simplesmente para realizar folhas de pagamento e controle de estoques, novas áreas de aplicação surgiram e, com elas, novos requisitos ao controle de concorrência. Mais modernamente, qualquer aplicação que faz uso intensivo de dados pode se beneficiar dos serviços providos por um gerenciador de banco de dados e muitas das novas aplicações incluem um banco de dados como um componente.

Existem várias *hipóteses implícitas* ao controle de concorrência clássico que recentemente (ou nem tão recentemente) acabaram sendo violadas por tais aplicações. Muitas delas não são hipóteses necessárias para se demonstrar a *correção* do controle de concorrência. Tais propriedades pertencem mais à prática do controle de concorrência do que à sua teoria. Os projetistas de sistemas gerenciadores de dados tratavam estas hipóteses como propriedades do sistema a ser gerenciado. Quando as aplicações começaram a violar estas hipóteses, o sistema deixou de ser gerenciável.

É óbvio que aquilo que chamamos de hipóteses implícitas à prática do controle de concorrência não foi adotado gratuitamente. Sua adoção facilitava grandemente a implementação do sistema e a sua violação, necessariamente, deve levar a sistemas mais complexos.

Examinaremos a seguir as seguintes hipóteses que, por muito tempo, foram tidas como garantidas por todas as aplicações:

- transações curtas;
- sequencialidade;
- homogeneidade; e
- disponibilidade.

Transações de Longa Duração. A principal de tais hipóteses implícitas é a da *curta duração* das transações. O sistema de controle de transações foi concebido para ser implementado com transações que duram uma fração de segundo ou, nos piores casos, algumas dezenas de segundos. É claro que a teoria da seriação está correta para transações de qualquer duração, curta ou longa. No entanto, transações longas tendem a tomar conta de recursos do sistema por muito tempo.

Tomemos como exemplo o caso do controle de concorrência por bloqueios bifásicos estrito. Se uma transação escreve um dado no início de seu processamento, este dado ficará bloqueado até o final da transação. Se esta transação durar horas (ou dias), este dado estará indisponível para o resto do sistema por todo este tempo. Ou seja, não há acesso concorrente ao dado, representando uma degeneração do controle de concorrência em escalonamentos seriais.

Com transações longas, realizando milhares de operações, também aumenta chance de ocorrência de travamentos. A solução para tais travamentos é abortar e reiniciar uma das transações envolvidas no travamento. Mas abortar e reiniciar uma transação que está sendo executada a horas ou dias não é uma tarefa que pode ser feita levianamente, devido ao seu grande custo computacional.

Em geral, o aborto de transações muito longas tem que ser evitado. Isso quer dizer que, no máximo, estamos dispostos a desfazer apenas um pedaço de uma transação, e não a transação inteira. Isto constitui uma *violação do princípio de atomicidade* das transações.

Por outro lado, o fato de abortarmos uma porção da transação, e não a transação toda, implica que estaremos liberando bloqueios obtidos pela transação antes do tempo. Isto constitui uma *violação ao princípio de seriação* de transações.

Tais problemas e violações não são exclusividade do controle de concorrência por bloqueios bifásicos e ocorrem em outros métodos de controle de concorrência também.

Transações Iterativas. Transações iterativas violam uma outra hipótese implícita comumente tomada como garantida em sistemas de bancos de dados, que é a hipótese do *processamento seqüencial*, ou processamento do tipo *batch*.

No processamento seqüencial uma transação submete uma série de operações ao banco de dados, sendo que suas únicas fontes de dados são: (a) parâmetros passados à transação no início do processamento ou (b) o próprio banco de dados. Desta forma, o processamento não depende da intervenção humana para prosseguir. No caso (a), o dado de entrada está disponível imediatamente, e no caso (b) o tempo de acesso aos dados é, predominantemente, o tempo de acesso a disco do sistema de banco de dados.

Já no processamento iterativo, uma transação dialoga com o usuário, obtendo informações deste durante a execução. Em comparação com o processamento seqüencial, o acesso aos dados provenientes do usuários são milhares, milhões ou até centenas de milhões de vezes mais lentos. Um usuário pode parar durante o processamento para atender ao telefone, sair da sala, ou mesmo abandonar o programa em estado de espera.

Tradicionalmente, a interação com o usuário era (e, em muitos sistemas, continua sendo) tratada por um programa externo ao banco de dados chamado genericamente de *interface*; essa interface coleta os dados do usuário e somente então disparava uma transação. Se mais de uma interação se faz necessária, então o processo é quebrado em várias transações, cada uma resultante de uma interação com o usuário.

Neste capítulo, vamos estudar novos modelos de transações que permitem às transações terem durações mais longas, sem com isso degradar a eficiência do sistema. Em geral, transações iterativas entram nesta categoria. Os problemas devido à heterogeneidade e indisponibilidade serão tratados no Capítulo 6.

5.1.1 A Organização do Capítulo

Existem dois modelos básicos de transações que são usados para tratar transações longas e iterativas: modelo de *transações aninhadas* e o modelo de *transações compensatórias*. Cada um destes modelos parte de um pressuposto diferente para evitar a degradação de um sistema contendo transações muito longas. Cada um destes modelos propõe um novo critério de correção para o escalonamento concorrente. A idéia é tentar manter ao máximo as propriedades ACID, violando-as de forma "principiada".

O modelo de transação aninhada é apresentado na Seção 5.2. A versão que apresentamos ainda se baseia no método tradicional de ordenação de operações conflitantes, o qual é a base da definição clássica de escalonamento apresentada na Seção 2.3.2.

Mas uma noção mais genérica, que emergiu da criação dos modelos não clássicos, foi o critério de *comutatividade* de operações. Tal critério está descrito na Seção 5.3.

Por fim, descrevemos o modelo de *transações compensatórias*, o qual se utiliza do critério de comutatividade para o seu princípio básico de funcionamento.

Deixamos par o próximo capítulo a descrição de aplicações que se utilizam destes modelos.

5.2 Transações Aninhadas

O modelo de *transações aninhadas*[1] consiste num desvio do modelo tradicional de transações apresentado no Capítulo 2 que estende o conceito de atomicidade, permite o aborto de uma subtransação sem que, com isso, toda a transação tenha que ser abortada.

Na realidade, existem várias propostas na literatura de transações aninhadas. Originalmente, foram descritos e implementados sistemas contendo noções de estruturação interna de transações em [Gra81, Mos81, LS83, Mos85], e o modelo teórico subjacente a sistemas de transações aninhadas foram apresentados em [BBG+83, FLMW87, FLMW88]. Nós seguimos aqui a apresentação de [FLMW87] (que posteriormente foi refinada em [FLMW88]).

Em todo sistema de transações aninhadas, as transações possuem uma *estrutura interna*. Esta estrutura pode ser descrita como uma hierarquia de (sub)transações, dispostas em formato de árvore. Todas as transações, menos uma *transação primordial* T_p são subtransações de uma outra. A estrutura hierárquica determina o tipo de relacionamento entre transações: uma subtransação que se encontra imediatamente abaixo de outra é chamada de *filha* desta; duas subtransações filhas de uma mesma transação *mãe* são chamadas de transações *irmãs*; uma transação é *ancestral* a outra se existe um caminho descendente na hierarquia que liga a primeira à segunda.

Um exemplo de uma transação aninhada está ilustrado na Figura 5.1. A transação primordial T_p situa-se à raiz da árvore, e ela é uma ancestral de todas as outras transações no sistemas. As transações filhas de T_p são chamadas de transações *topo*; as transações topo na Figura 5.1 são T_1, T_2 e

[1] A designação original em inglês, *nested transactions*, muitas vezes é traduzida para o português como *transações embutidas*.

T_3. Temos diversos grupos de transações irmãs na Figura 5.1: (T_1, T_2, T_3), (T_5, T_6), (T_7, T_8) e (T_9, T_{10}).

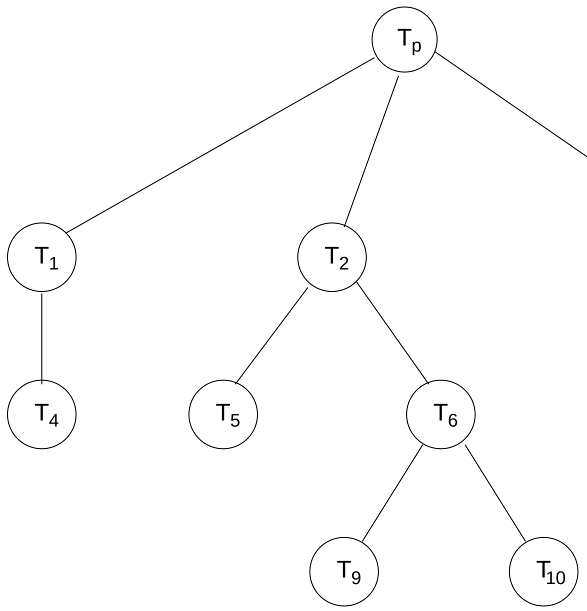


Figura 5.1: Estrutura interna de uma transação aninhada

Em nosso modelo, as transações podem ser de dois tipos:

- *Transações Internas*: Estas transações tem por função controlar a criação de outras subtransações, e decidir que atitude deve ser tomada quando uma transação filha aborta. As transações internas *não realizam acessos aos dados do banco de dados*. Todos acessos a dados serão realizadas por suas filhas e descendentes.

- *Transações de Acesso*: São as transações que realizam as operações no banco de dados; continuamos assumindo que estas operações são de escrita e leitura apenas. Transações de acesso não possuem descendentes e, portanto, ocupam as folhas da hierarquia de subtransações.

Na Figura 5.1, são transações de acesso: $T_4, T_5, T_9, T_{10}, T_7, T_8$. As demais transações são transações internas, com exceção da raiz T_p.

Com relação à terminação, as transações ainda podem apenas abortar ou se confirmar, mas o comportamento de terminação de uma subtransação está restrito a sua posição na hierarquia:

- A confirmação de uma transação é condicional à confirmação de sua mãe. Ou seja, uma subtransação confirmada ainda poderá ser abortada caso sua transação mãe venha a ser abortada.
- O aborto de uma subtransação não necessariamente implica no aborta de sua transação mãe. Esta poderá decidir entre abortar-se, reiniciar a subtransação abortada, criar uma nova transação ou simplesmente ignorar o aborto de sua filha. Uma transação de acesso abortada continua tendo todas as suas operações desfeitas.

Uma transação ativa ou confirmada que possui um ancestral abortado é dita uma *transação órfã*. Estamos assumindo neste modelo que todas as transações órfãs devem ser abortadas e desfeitas. Ou seja, ao se abortar uma transação interna, este aborto será cascateado para todas as suas transações filhas. Na realidade, este tipo de cascatemento de aborto é um pouco distinto daquele apresentado na Seção 2.2.2, onde o cascateamento de abortos era provocado pela leitura de um dado de uma transação abortada. Aqui, o cascateamento se dá simplesmente pela posição na hierarquia de transações aninhadas.

Desta forma, se uma transação interna é abortada, devemos procurar todas as suas filhas ativas e (condicionalmente) confirmadas para abortá-las. Neste processo recursivo, todas as operações feitas por alguma de suas transações descendentes nas folhas da hierarquia serão eventualmente desfeitas. Por exemplo, na Figura 5.1, se a transação T_2 abortar, as transações T_5, T_6, T_9, T_{10} ficam órfãs, e devem ser subseqüentemente abortadas.

É de se esperar que este processo do desfazer de transações condicionalmente confirmadas acarrete um aumento de complexidade ao gerenciador de dados e à manutenção do log, uma vez que a confirmação de uma transação não mais é durável.

Conforme descrito em [LS83], o valor de um objeto x pode ser modelado como uma pilha de valores. Cada vez que um valor confirmado é escrito em x, este valor vai ao topo da pilha. Se a transação é abortada após sua confirmação, este valor é retirado do topo da pilha. É responsabilidade do controle de concorrência garantir que, se uma transação T_i será desfeita, todos os valores por ela escritos estão no topo de suas respectivas pilhas.

A Figura 5.2 ilustra três momentos na evolução de um dado x. Inicialmente, em (i), supomos que o valor atual de x é b, escrito pela transação T_7 sobre um valor escrito por T_4. Em (ii), um novo valor c foi escrito em x por T_8. Em (iii), estamos supondo que a transação T_3 abortou e, por conseqüência, as atualizações de suas filhas T_7 e T_8 foram desfeitas, de forma que o valor atual é a, escrito por T_4.

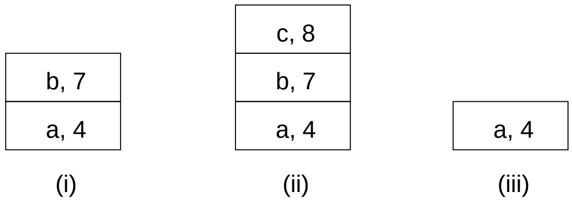


Figura 5.2: Atualização de um dado x

Para garantir tal efeito, um escalonador de transações aninhadas precisa garantir que *transações irmãs são seriáveis.* O conceito de transações irmãs seriáveis precisa ser definido recursivamente e da maneira similar à maneira tradicional: equivalência a um escalonamento serial.

Definição 5.2.1 [Transações Aninhadas Seriáveis]

Dizemos que duas transações são executadas em *ordem estrita* se todas as operações de uma transação precedem todas as operações da outra.

As *operações de uma transação de acesso* são as operações de leitura, escrita e término (confirmação ou aborto) realizada por esta. As *operações realizadas por uma transação interna* são dadas pelo conjunto de operações realizadas por suas transações filhas, mais a sua operação de término.

Uma *transação aninhada é serial* se transações irmãs são executadas na estrita ordem em que aparecem na hierarquia.

A noção de *equivalência* é definida recursivamente sobre a estrutura da transação de forma similar. Desta forma, a execução concorrente de um conjunto de transações filhas de uma mesma transação mãe é *equivalente* a um outro escalonamento das mesmas transações se satisfizerem o critério tradicional de equivalência dado pela Definição 2.3.5, ou seja, se estes dois escalonamentos ordenam igualmente as operações conflitantes.

Desta forma, duas transações aninhadas para serem equivalentes devem ter a mesma estrutura.

Finalmente, uma transação aninhada é *seriável* se for equivalente a uma transação serial. A definição de transação seriável é exatamente igual ao caso clássico. □

Para verificar a serição de uma transação, procedemos da seguinte maneira. Para os uma transação interna, toma-se o conjunto de transações filhas e verifica-se se elas são seriáveis. Para isso, toma-se as filhas destas, e assim sucessivamente, até chegarmos às transações de acesso. Por exemplo, na Figura 5.1, precisamos garantir a seriação dos seguinte grupos de transações: (T_1, T_2, T_3), (T_5, T_6), (T_7, T_8) e (T_9, T_{10}).

Nota-se que o modelo de transações aninhadas permite um maior grau de concorrência interna à cada transação, pois permite que as subtransações sejam executadas concorrentemente. Independentemente de qual a seriação entre as filhas de uma transação T, requer-se que T seja seriável com suas transações irmãs. A seguir descrevemos uma forma de garantir tal seriação de transações aninhadas.

5.2.1 Controle de Concorrência em Transações Aninhadas

O primeiro método de controle de concorrência para transações aninhadas foi proposto por [Mos81] e sua correção demonstrada em [FLMW87]. Este método é baseado em bloqueios e será descrito a seguir sem, contudo, entrar nos detalhes de demonstração[2]. Posteriormente, outros métodos que não utilizam bloqueios foram propostos, por exemplo, um método baseado na comutatividade de operações em [FLMW88].

A idéia de bloqueios permanece a mesma: antes de realizar uma operação $p_i[x]$, a transação T_i necessita obter um bloqueio $pl_i[x]$ para tal operação. No entanto, a noção de conflito entre bloqueios foi alterada. Dois bloqueios $pl_i[x]$ e $ql_j[x]$ conflitam se:

- pelo menos uma das operações p_i e q_j é uma operação de escrita; e
- T_i não é ancestral de T_j nem T_j é ancestral de T_i.

Desta forma, não há conflitos entre um ancestral e seus descendentes.

O protocolo de controle de concorrência por bloqueios em transações aninhadas obedece às seguintes regras:

1. Antes de realizar uma operação $p_i[x]$, uma transação deve obter o bloqueio $pl_i[x]$.
2. A transação só obtém o bloqueio $pl_i[x]$ se nenhuma outra transação T_j no sistema possuir um bloqueio conflitante $ql_j[x]$; caso contrário, a transação é posta em espera.
3. Ao abortar, todos os bloqueios de uma transação são liberados.

[2] A demonstração de [FLMW87] utiliza um arcabouço teórico diferente do que utilizamos no Capítulo 2, e a limitação de espaço não nos permite introduzi-lo com seus detalhes para provarmos a seriação.

4. Uma transação interna só se confirma se todas as suas filhas já se terminaram (confirmadas ou abortadas).

5. *Ao se confirmar, todos os bloqueios de uma transação passam para sua mãe.*

Esta última regra é fundamental para o funcionamento do sistema. Não há liberação de bloqueios em uma subtransação ativa, todas as mudanças de posse de bloqueios se dá ao término (com sucesso ou não) de uma transação.

Por exemplo, suponhamos que na Figura 5.1 a transação T_5 deseja obter um bloqueio de escrita $wl_5[x]$, só que a transação T_9 possui um bloqueio conflitante $rl_9[x]$. Neste ponto, a transação T_5 será posta em espera. Quando T_9 se confirma, o bloqueio $rl_9[x]$ passa para sua mãe, T_6, mas $rl_6[x]$ também conflita com $wl_5[x]$, e T_5 permanece em espera. Quando T_6 se confirma, o bloqueio $rl_6[x]$ passa para T_2. Neste momento, $rl_2[x]$ não mais está em conflito com $wl_5[x]$ pois não há conflitos entre T_5 e seus ancestrais. T_5 pode finalmente sair da espera e executar $w_5[x]$.

Desta forma, fica claro que se uma transação T_i requer um bloqueio $pl_i[x]$ conflitante com $ql_j[x]$, T_i ficará em espera até que T_j seja abortada, ou até que o bloqueio $ql[x]$ chegue a um ancestral comum de T_i e T_j.

A correção deste protocolo de controle de concorrência entre transações aninhadas está baseada no seguinte fato. Para duas transações de acesso irmãs, tudo se passa como se estivéssemos num protocolo BBF estrito, pois os bloqueios são obtidos durante a execução e não são liberados até o término da transação; outros bloqueios conflitantes, retidos por outras transações, não causam a violação da serialidade entre irmãs. Desta forma, garante-se que duas transações de acesso irmãs serão sempre seriáveis.

O próximo passo na verificação da correção é notar que, para uma transação interna, tudo se passa como uma transação normal, em que os bloqueios são obtidos "em bloco". Cada bloco de bloqueios obtidos constitui os bloqueios recebidos de alguma transação filha que se confirmou. Como não há liberação de bloqueios até o término da transação interna, esta transação também se comporta de acordo com o protocolo BBF estrito. Desta forma duas transações irmãs quaisquer também são seriáveis.

Por fim, considere todas as transações de acesso que são dependentes de uma mesma transações internas T_I. Como há uma seriação de cada filha confirmada de T_I, e cada uma destas filhas tem uma seriação de suas filhas, podemos facilmente construir uma seriação das netas de T_I. Iterando-se este processo recursivamente, obtemos uma seriação de todas as transações confirmadas de acesso que são descendentes de T_I. Como isto vale para toda transação interna T_I, vale também para a transação primordial T_p que sempre está na raiz da hierarquia do sistema.

Ou seja, de acordo com o protocolo de escalonamento de transações aninhadas acima, sempre existe uma seriação para o conjunto de todas as transações confirmadas de acesso. Esta propriedade nos lembra o modelo tradicional, onde sempre se pode ter um escalonamento serial de um conjunto de transações.

5.2.2 Comparações entre Transações Tradicionais e Aninhadas

Com base no que acabamos de afirmar, é natural que exista uma dúvida neste ponto é a seguinte: qual a diferença entre os modelos tradicionais e o aninhado, no que se refere à transações de acesso? Esta é uma dúvida válida, pois as transações de acesso podem ser vistas como as transações básicas do modelo clássico. Como só elas realizam operações no sistema, tudo se passaria como se só houvesse transações de acesso.

A diferença está na granularidade das transações. Uma transação pode dividir seu trabalho em diversas subtransações. Pode ser que uma subtransação aborte, liberando bloqueios, sem que com isso sua transação mãe seja obrigada a abortar. Estes abortos parciais permitem um maior grau de concorrência e evitam que uma transação que está sendo executada há muito tempo tenha que abortar por um motivo qualquer. Dizemos que há uma *maior granularidade* na concorrência das transações, ou seja, para um mesmo conjunto de operações a dados, as transações aninhadas permitem um maior número de transações e, a falha de uma dessas transações menores leva ao desfazer de um menor número de operações.

Os travamentos permanecem um problema neste tipo de controle de concorrência baseado em bloqueios. No entanto, como a granularidade das transações de acesso é bem maior, a probabilidade de ocorrência de um travamento diminui. Se cada transação de acesso executar uma única operação, não haverá travamentos no sistema; porém, esta não é uma condição possível de ser verificada na maioria dos sistemas práticos. Em geral, uma transação executa um grupo de operações. Por exemplo, se vamos alterar uma tabela em várias posições com um único comando SQL, este grupo de operações é executado dentro de uma única (sub)transação.

Como desvantagens das transações aninhadas, podemos citar o custo da recuperação de falhas, haja visto que uma transação confirmada pode ser abortada se algum ancestral seu abortar, levando a um complicado mecanismo de manutenção de log e o desfazer de transações abortadas.

A sobrecarga do escalonador é bem mais alta do que a do sistema de transações tradicional, uma vez que a verificação de conflitos entre operações é mais complicada e, conseqüentemente, mais lenta. Por fim, a decisão do que fazer quando uma transação filha aborta deve ficar a cargo do programador da aplicação, fazendo com que as transações internas sejam de complexidade mais elevada.

5.3 Operações Comutativas

Considere a seguinte operação $p_i[x]$ que atualiza o saldo da conta x, depositando nesta conta \$100,00:

$$x := x + 100$$

Considere agora a seguinte operação $q_j[x]$ que saca \$50,00 desta mesma conta:

$$x := x - 50$$

É claro que, se cada uma destas operações forem executadas atomicamente, a ordem relativa em que estas operações foram executada não importa. Que p_i seja executada primeiro, quer q_j seja executada primeiro, o valor de final x será exatamente o mesmo. Isto se dá pois p_i e q_j são operações *comutativas*.

Para podermos definir melhor a noção de comutatividade, temos que mencionar a noção de *estado de um banco de dados*. O estado S de um banco de dados é uma função que mapeia cada item de dados x em um valor pertencente ao domínio D_x de valores de x, tal que $S(x) \in D_x$. Ou seja, se x é um dado com valor Inteiro, em qualquer estado do banco de dados, o valor associado a x, $S(x)$, deve ser um número inteiro.

Segundo esta nova visão do banco de dados, uma operações no banco de dados também é uma função. Note que isto é uma *grande* diferença da teoria clássica. Na teoria clássica, sequer tratávamos do valor de um item de dados. Toda a teoria era organizada em torno da ordenação das *operações conflitantes*. Uma transação ou um escalonamento eram ordens parciais nas quais as todas as operações conflitantes deviam estar ordenadas.

Nesta nova visão, uma operação é *uma função que transforma um estado do banco de dados em outro*. Desta forma, se estamos inicialmente com um banco de dados num estado S, e aplicamos a este banco de dados uma operação O, o valor do item de dados x após esta operação é $O(S(x))$. Ou seja, o estado do banco de dados S' após a a aplicação de uma operação O em S é a *composição das funções S e O*:

$$S' = S \circ O,$$

onde $\circ$ indica a composição de funções.

Neste cenário, uma transação é uma seqÜência de operações

$$T = O_1; O_2; \ldots; O_k.$$

Se não há nenhuma outra transação sendo executada no sistema, o efeito da transação é o efeito combinado de suas operações:

$$S' = S \circ O_1 \circ O_2 \circ \ldots \circ O_k.$$

Neste caso, podemos escrever que o estado após a execução de T é $S' = S \circ T$. Note que não há necessidade de ordenar-se operações conflitantes; por outro lado, não há nenhuma noção explícita de concorrência interna a uma transação, como no modelo tradicional.

Note que nesta nova visão, as operações não estão restritas apenas a leitura e escrita, como no modelo tradicional. No exemplo acima, as operações p_i e q_j são operações de atualização. Apenas duas propriedades básicas são exigidas das operações:

- *Atomicidade*: uma operação O deve ser executada totalmente, ou não ter nenhum efeito sobre o estado do banco de dados.

- *Inversão*: um operação O deve ser inversível, de forma a desfazer o seu efeito sobre o banco de dados. Por exemplo, se W é uma operação de escrita, então W^{-1} é a operação que desfaz esta escrita, ou seja, escreve o seu valor inicial. Isto é expresso matematicamente como $W \circ W^{-1} = I$, onde I é a função identidade tal que $I(x) = x$. Se R é uma operação de leitura, sua inversa é o próprio I, uma vez que a leitura de um dado não provoca nenhuma alteração no estado do banco de dados. No entanto, operações de leitura devem ser representadas, para fins de controle de concorrência.

Note que neste modelo, não há representação explícita das operações de confirmação e aborto. Assume-se que toda transação bem sucedida é confirmada, e que as transações abortadas são seguidas das operações que desfazem, em ordem, as operações por ela executadas, ou seja, se uma transação executou as operações $O_1; \ldots; O_k$ até ser abortada, a operação de aborto é seguida da aplicação das operação composta $O_k^{-1} \circ \ldots \circ O_1^{-1}$ que desfaz a transação.

Dados um conjunto de transações $T_1, \ldots, T_n$, seja $O_{i,j}$ a j-ésima operação da transação T_i. Um *escalonamento* destas transações é uma função resultante da composição das operações que formam as transações, respeitadas a ordem em que as operações aparecem em cada T_i, ou seja, se $j_1 < j_2$ então um escalonamento E deve ter a forma:

$$E = \ldots \circ O_{i,j_1} \circ \ldots \circ O_{i,j_2} \circ \ldots$$

para todo i, $(1 \leq i \leq n)$.

A noção de concorrência é expressa neste modelo através da *comutatividade de operações*. Duas operações $O - 1$ e O_2 são comutativas se, para todo estado S do banco de dados, temos que:

$$S \circ O_1 \circ O_2 = S \circ O_2 \circ O_1$$

As operações comutativas são aquelas que podem ser executadas em qualquer ordem e, portanto, podem ser executadas concorrentemente. No exemplo acima, vimos que duas atualizações do mesmo saldo, uma de depósito e outra de saque:

$$O_1 = (x := x + v) \qquad O_2 = (x := x - u)$$

são operações comutativas e podem ser realizadas em qualquer ordem, desde que sejam executadas atomicamente. Em geral, operações que envolvem conjuntos de dados distintos são comutativas, e podem ser realizadas concorrentemente.

Definição 5.3.1 Dois escalonamentos, E e E' sobre o mesmo conjunto de operações são ditos *c-equivalentes* (equivalentes por comutatividade), $E \equiv E'$, se um pode ser obtido a partir do outro simplemente permutando-se as operações comutativas.

Equivalentemente, podemos dizer que $E \equiv E'$ se $E_1(S) = E_2(S)$ para todo estado S.

Note que as operações ditas conflitantes no modelo clássico são exatamente as operações não-comutativas. Neste novo modelo, temos uma maior variedade de operações possíveis, e uma maior flexibilidade no tratamento do desfazer de uma operação. Este último fato será extensivamente usado no tratamento de transações compensatórias, na Seção 5.4, onde veremos que o restrição de que e e E' sejam escalonamentos sobre o *mesmo* conjunto de transações pode ser abandonado ao tratarmos da compensação de uma transação.

Podemos então definir a noção se seriação neste novo modelo de forma análoga ao caso clássico. Um escalonamento E sobre transações $T_1, \ldots, T_n$ é dito *serial* se

$$E = T_{i_1} \circ \ldots \circ T_{i_n}$$

onde $i_1, \ldots, i_n$ é uma permutação de $1, \ldots, n$.

Por fim, o conceito de seriação é idêntico ao caso clássico: um escalonamento E é dito c-serial se for c-equivalente a um escalonamento serial E_s.

Note que se as operações permitidas forem apenas as de leitura e escrita, os escalonamentos seriáveis no caso clássico sempre serão seriáveis no enfoque comutativo. No entanto, a demonstração da correção do brotocolo BBF é muito mais simples e natural no modelo clássico, e não o faremos aqui para o modelo comutativo.

5.3.1 Comutatividade e Transações Aninhadas

O modelo de comutatividade pode ser facilmente extendido para o caso de transações aninhadas. No caso de transações de acesso, duas transações irmãs são c-seriáveis (ou seja, seriáveis por comutatividade) se sua execução concorrente é equivalente a uma execução serial destas duas transações de acordo com o conceito de c-seriação exposto acima. Isto é duas transações de acesso são c-seriáveis se através da permutação de operações comutativas, obtém-se um escalonamento serial.

Recursivamente, podemos obter uma definição de c-seriação para transações aninhadas baseada em comutatividade de operações de maneira semelhande a usada na Definição 5.2.1. Esta definição segue o método padrao: duas transações irmãs serão c-seriáveis se houver um escalonamento serial c-equivalente de suas transações filhas.

Como a noção de transação aninhada serial independe do critério de de seriação, vamos assumir a mesma definição de transação aninhada serial que a usada na Definição 5.2.1. Resta apenas a definição de c-equivalência.

Definição 5.3.2 [Transações Aninhadas c-Seriáveis]
A noção de *c-equivalência* é definida recursivamente:

- No nível de acesso, a execução concorrente de um conjunto de transações filhas de uma mesma transação mãe é *c-equivalente* a um outro escalonamento das mesmas transações se satisfizerem o critério não-aninhado de equivalência dado pela Definição 5.3.1.

- Para os nós internos, transações irmãs $T_{i_1}, \ldots T_{i_n}$ são c-equivalentes a execução de $T_{j_1}, \ldots T_{j_n}$ se as transações filhas de T_{i_k} forem c-equivalentes às transações filhas de T_{j_k}, para $1 \leq k \leq n$.

Como no caso anterior, duas transações aninhadas para serem c-equivalentes devem ter a mesma estrutura.

Finalmente, uma transação aninhada é *seriável* se for equivalente a uma transação serial. A definição de transação seriável é exatamente igual ao caso clássico. □

Em [FLMW87] mostra-se que é possível criar um controle de concorrência baseado na comutatividade de operações que seja mais flexível que o método baseado em bloqueios apresentado na Seção 5.2.1. Este método assume que haja uma forma de detectar se duas operações são comutativas ou não. O mecanismo de bloqueio da Seção 5.2.1 pode ser extendido de bloqueio de "operações conflitantes" para bloqueio de "operações não-comutativas". Os detalhes de tal implementação não serão apresentados aqui.

O critério de seriação baseada em comutatividade de operações é crucial para a compreensão do mecanismo de funcionamento de transações compensatórias descrito a seguir.

5.4 Transações Compensatórias

As *transações compensatórias* foram introduzidas por [KLS90, KLS98] para lidar com situações em que se faz necessário desfazer transações confirmadas ou transações ativas que afetam outras transações, sem que com isso seja preciso recorrer à abortos cascateados.

Transações compensatórias podem fazer parte tanto de um sistema de transações aninhadas quanto de um sistema de transações mais tradicional. No entanto, a introdução de transações compensatórias consiste numa ruptura com relação aos métodos tradicionais de recuperação, onde, conforme visto no Capítulo 2, se requer que:

- transações confirmadas sejam tratadas como permanentes e irreversíveis; e

- para evitar o cascateamento de aborto, uma transação só deve ler dados de transações confirmadas.

Ambos princípios são violados pelas transações compensatórias, que são utilizadas por aplicações modernas onde há transações muito longas e que ficariam impraticáveis caso os princípios acima fossem rigorosamente observados (como já havia sido notado em [Gra81]). Em especial, o mecanismo de transações compensatórias se aplica ao caso em que há necessidade de se permitir que transações leiam *dados não-confirmados* de outras transações. Este é o caso de dados escritos por transações muito longas, em que não é possível esperar por uma confirmação para ler os dados escritos por ela.

Outra área de aplicação das transações compensatórias são as transações usadas em computação colaborativa, muito comuns em atividades de projeto grupais assistido por ferramentas automáticas [KaF88].

A noção principal aqui é a da *compensação correta*. Existe uma gama muito ampla de tipos de compensação correta, que pode ir desde o tradicional desfazer de transações, num extremo, à compensação especial e dependente de aplicação, no outro extremo.

Para a descrição dos critérios de compensação correta, é necessário considerar a semântica da transação. No entanto, o modelo tradicional desenvolvido no Capítulo 2, ignora totalmente o código das aplicações e os valores escritos e lidos por uma transação, concentrando-se apenas na ordem em que estes foram valores foram lidos e escritos. Para podermos lidar com transações compensatórias um outro modelo se faz necessário para que elementos semânticos sejam incorporados.

5.4.1 Um Modelo de Transações que Permite Compensação

Esta seção trata de criar uma instanciação em detalhes para um conjunto de operações comutativas. Ele pode ser visto como uma particularização do esquema genérico apresentado na Seção 5.3.

Para compensar uma transação é necessário levar em consideração a semântica do *programa da transação*. Este programa pode estar definido em qualquer linguagem de alto-nível, contendo suas próprias *variáveis locais* (entidades não armazenadas no BD). Portanto, definimos o conceito de *estado aumentado*, que compreende os itens de dados do banco de dados e as variáveis locais a um programa de transação.

Num estado aumento, podemos tratar uniformemente leitura e escrita como um atualização. A leitura torna-se a atualização de uma variável local, e a escrita torna-se uma atualização de um item de dados no BD. Representaremos por BDA o banco de dados unido com as variáveis locais, onde BDA $\supset$ BD. Um estado S do BDA é um mapeamento de nomes de entidades do BDA a valores de entidades do BDA.

O conceito de *operação* neste modelo difere do conceito de operação no modelo clássico (que só permite leituras e escritas). Uma operação é uma função $f : BDA \rightarrow BDA$ que respeita as seguintes restrições:

- Cada operação atualiza no máximo uma entidade (item de dado ou variável local).
- Uma operação lê, no máximo, um item de dados, mas pode ler diversas variáveis locais.
- Se uma operação só lê e atualiza o BD se ambas operações se deserm sobre o mesmo item de dado.
- Cada operação é realizada *atomicamente*.

Representa-se uma operação f como: $e := f(x; u_1, \ldots, u_n)$, onde e é a entidade sendo atualizada, x é o item de dados *lido* (que pode ser omitido) e $u_1, \ldots, u_n$ são as variáveis locais *lidas*. Além disso, temos também as operações de aborto (a) e confirmação (c) de uma transação.

Um comando de um programa de transação pode ser uma operação, uma sequência de operações e, além destas operações normais, o modelo também permite comandos condicionais do tipo `if` B `then` C_1 `else` C_2, onde C_1 e C_2 são seqüências de comandos[3].

Exemplo 5.4.1 Considere, por exemplo uma programa de transação bancária que transfere todo o valor de uma conta y para uma conta x. A transferência só ocorre se o valor de y for positivo.

```
T :  u := y;
     if u > 0 then
        begin
          x := x + u;
          y := 0
        end;
```

A operação `x := x + u` é do tipo $e := f(x; u)$, onde $e = x$; a operação `u := y` é do tipo $e := f(y)$, e seria o equivalente, no modelo tradicional, à leitura da item de dado y, que fica escrito na variável local u; a operação `y := 0` é uma escrita simples, sem leitura associada. □

[3]Note que esta definição, ao contrário da definição tradicional, não permite concorrência interna a uma transação.

Supõe-se ainda as seguintes restrições sobre programas de transação:

- O conjunto de variáveis locais é particionado entre as diversas transações, de forma que duas transações nunca acessam as mesmas variáveis locais.
- Variáveis locais são atualizadas apenas uma vez durante a transação.
- Uma operação só lerá uma variável local se ela já tiver sido escrita por uma outra operação anterior.

Todas estas restrições são verificadas pelo exemplo anterior. Vamos agora mudar o foco de programas de transação para execuções de transações, ou simplesmente transações.

Cada transação T está munida de um par de predicados, P_T e Q_T. O *predicado de entrada* P_T é uma pré-condição que deve ser satisfeita pelo estado do BD lido pela transação no seu início. O *predicado de saída* Q_T é uma pós-condição que a transação garante ser válida no estado do BD ao seu término, na ausência de concorrência, se o predicado de entrada era válido no seu início. Por composição de efeitos, um conjunto de transações $\{T_1, \ldots, T_n\}$ pode ser associado a um predicado de saída, $Q_{\{T_1,\ldots,T_n\}}$, representando o fato de que um conjunto de transações coletivamente pode estabelecer uma tarefa coerente. Tal fato será crucial para a definição de correção de transações compensatórias.

Este modelo não adota a noção de ordem parcial entre as operações, sendo baseado simplesmente na comutatividade entre operações de diversas transações, conforme descrito na Seção 5.3. Como operações são funções de estados BDAs em estados BDAs, um *escalonamento* E é definido simplesmente como uma sequência de operações ou, no caso, uma composição de funções $E = f_1 \circ \ldots \circ f_n$. Desta forma, um escalonamento também é uma função de estados do BDAs em estados do BDAs, e escalonamentos também podem ser compostos por $\circ$. Assim, uma sequência de operações F é uma *subsequência* de E se existirem E' e E'' tal que $E = E' \circ F \circ E''$.

Dois escalonamentos E_1 e E_2 são *equivalentes*, $E_1 \equiv E_2$, se para todo estado S do BDA, $E_1(S) = E_2(S)$. Note que esta definição não obriga os escalonamentos E_1 e E_2 a serem sobre o mesmo conjunto de transações, o que será útil pois ao se falar em compensação teremos que igualar escalonamentos sobre conjuntos distintos de transação. Neste fato, o modelo aqui apresentado é mais genérico que o da Seção 5.3, o qual requer que E_1 e E_2 sejam definidos sobre o mesmo conjunto de transações.

Uma noção básica para o tratamento de compensação é a *comutatividade*. Conforme visto na Seção 5.3, duas sequências de operações E e E' comutam se $(E \circ E') \equiv (E' \circ E)$. Duas operações são *conflitantes* se elas não comutam. Esta noção de conflito baseada em comutatividade é muito mais genérica que a noção tradicional baseada em escrita de dados. O exemplo a seguir mostra que a noção de comutatividade pode eliminar certos conflitos, permitindo um maior grau de concorrência.

Exemplo 5.4.2 Suponhamos que temos duas transações bancárias, T_1 e T_2, que realizam depósitos numa mesma conta x. Os valores y e z são itens de dados e u_1 e u_2 são variáveis locais a T_1 e T_2, respectivamente. Na notação deste novo modelo, temos:

$$\begin{array}{l} T_1 : [u_1 := y; \quad x := x + u_1; \quad c_1] \\ T_2 : [u_2 := z; \quad x := x + u_2; \quad c_2] \end{array}$$

Note que todas as operações de T_1 comutam com as operações de T_2 (mas as operações de T_1 não comutam entre si, pois u_1 deve ser escrito antes de ser adicionado a x), e vice-versa, muito embora ambas as transações leiam x escrevam sobre x. Isto se dá devido a comutatividade da adição; não importa em que ordem os depósitos são feitos, o valor final de x será o mesmo. □

É claro que só necessitamos uma ordenação total de transações conflitantes, pois as demis comutam e podem ser executadas em qualquer ordem.

Neste contexto, um escalonamento E_s é *serial* se para todo par de transações T_i e T_j em E_s, todas as operações de T_i precedem as de T_j, ou vice-versa.

Um escalonamento E será seriável se existir um escalonamento serial E_s tal que $E \equiv E_s$. O conceito de seriação permanece o critério básico de correção no sistema.

Exemplo 5.4.3 Considere as transações T_1 e T_2 do exemplo acima. Um possível escalonamento de suas operações seria:

$$\begin{array}{rl} E = & [T_1 : u_1 := y; \;\; T_2 : u_2 := z; \\ & T_2 : x := x + u_2; \;\; c_2; \;\; T_1 : x := x + u_1; \;\; c_1] \end{array}$$

Este escalonamento é seriável e admite tanto a seriação $T_1; T_2$ quanto $T_2; T_1$, devido a comutatividade das operações entre as transações. Note que o método tradicional apenas permitiria a segunda seriação. □

Se CT for uma transação que irá compensar a transação T, então CT deverá ser sempre seriada após T, caso contrário a compensão não faria sentido.

Por fim, umas últimas definições. Se T é uma transação, denotaremos por E_T a sequência de operações de T em E. Se e é uma entidade do BDA, então E_e é a sequência das operações de E que atualizaram e; E_e é a *projeção* de E sobre e. Se E_T for projetado sobre e escreveremos $E_{T,e}$. Um escalonamento E é *seriável por entidades* se para toda entidade e existir um escalonamento serial E' tal que $E_e \equiv E'_e$. É óbvio que se E é seriavel então ele é seriável por entidades.

5.4.2 Critérios de Compensação

Muito embora a noção de compensação seja dependente da aplicação, alguns critérios globais devem ser satisfeitos pelas transações compensatórias. Representaremos por CT a transação que irá compensar a transação T.

Boa parte das atividades de CT está ligada ás transações cuja execução depende da execução de T. Definimos o conjunto $dep(T)$ de transações dependente de T como o conjunto de transações que leram algum dado escrito por T (conforme definido na Seção 2.2.2). Uma transação dependente de T pode estar confirmada ou ativa.

Estamos agora em condições de definir os critérios básicos de transações compensatórias. Seja E um escalonamento sobre T, sua transação compensatória CT e eventualmente um conjunto $dep(T)$ de transações dependentes de T.

Critério 1:

Se $E_{T,e} \circ E_{CT,e}$ é uma subsequência de E_e, então $(E_{T,e} \circ E_{CT,e}) \equiv I$, onde I é a transformação identidade.

Basicamente, este critério diz que, se T atualizou uma entidade e e nenuma outra transação leu ou escreveu sobre e até que T fosse compensada, então a atualização de e por T deve ser desfeita por CT, restabelecendo o valor de e anterior à execuçÃo de T. Note que se alguma transação tivesse lido ou escrito sobre e após sua atualização por T, o par $E_{T,e} \circ E_{CT,e}$ não seria uma subsequência de E_e.

Este critério deixa claro que, apesar de uma transação compensatória ser dependente de aplicação, uma transação compensatória deve se comportar como um procedimento de desfazer sempre que possível. Se não há nenhuma transação que tenha lido ou escrito sobre dados alterados por T, o critério acima afirma que CT se resume no desfazer de T.

Se há alguma transação dependente de T, então a compensação permanece dependente da aplicação.

O próximo critério está relacionado com a atomicidade da compensação.

Critério 2:

Se CT compensa T no escalonamento E, então deve haver subsequências de E, E' e E'', tais que $E \equiv E' \circ CT \circ E''$, e nenhuma transação possui operações em ambos E' e E''.

Ou seja, uma transação de E ou deve ver um estado afetado por T antes de CT, ou deve enchergar um estado após o término de CT. É de responsabilidade do controle de concorrência garantir que tal comportamento seja verificado.

Uma das formas possíveis de se garantir tal comportamento seria a seguinte. Suponhamos que o banco de dados possui um mecanismo de bloqueio de itens de dados. Seja $mod(T, E)$ o conjunto de itens de dados modificados por T em E. Então a compensação CT deve tratar os itens de dados em $mod(T, E)$ de acordo com o protocolo BBF. Isto garante que todas as transações que alteravam algum dado em $mod(T, E)$ serão seriadas ou antes ou após CT, respeitando o critério acima.

Outra decorrência deste fato é que a transação de compensação deve conhecer exatamente quais os dados que foram alterados por T. Para que isso seja possível na prática, é necessário que CT vasculhe o log de operações desde o fim de T até seu começo. Portanto, um sistema que dê suporte a transações compensatórias deve ser capaz de prover acessos de leitura ao log enquanto outras transações estão escrevendo sobre o log, caso contrário a leitura do log por CT iria interromper a escrita sequencial sobre o log, o que na prática tem o efeito de parar a execução de todas as outras transações no sistema.

Estaremos supondo que todas as transações compensatórias obedecem sempre aos critérios 1 e 2. O terceiro critério de compensação leva explicitamente em consideração a semântica das transações. Ele é, portanto de implementação mais difícil.

Seja o predicado sobre o estado do banco de dados P_T a pré-condição de T, seja $Q_{dep(T)}$ a pós-condição do conjunto de transações $dep(T)$ e seja Q_{CT} a pós-condição da execução de CT.

Critério 3:

Seja O um predicado sobre o estado do banco de dados tal que $(P_T \Rightarrow O) \wedge (Q_{dep(T)} \Rightarrow O)$. Então $Q_{CT} \Rightarrow O$.

Este critério deve ser entendido como O sendo uma restrição de integridade ou uma propriedade estabelecida pela execução conjunta de $dep(T)$. Neste caso, como O era válido antes da execução de T e foi preservado pela execução de $dep(T)$, O deverá ser restabelecido por CT. Semanticamente falando, este critério diz que devem ser preservadas pela compensação as propriedades que eram válidas antes de T e que permaneceram ou foram restabelecidas por $dep(T)$. As propriedades estabelecidas por T e preservadas por $dep(T)$ não precisam ser preservadas pela compensação.

Vamos admitir, a seguir, os critérios 1 e 2 são sempre verificados, deixando as complicações semânticas do critério 3 para a Seção 5.4.4.

5.4.3 Compensação Correta

Em algumas aplicações é possivel tolerar que o resultado da execução de uma transação compensatória *não* deixe o banco de dados no mesmo estado que teria sido obtido caso nem a transação compensada nem a compensatória tivessem sido executadas.

No entanto, algumas aplicações podem proibir a compensação a menos que seja garantido que a compensação CT tem o mesmo efeito que teríamos se nem T nem CT jamais tivessem sido executadas. Esta situação é que chamamos de uma *compensação correta*, que é o tipo de compensação mais importante.

Definição 5.4.4 Seja E um escalonamento sobre T, $dep(T)$ e CT, cujo estado inicial é S. Seja E' um escalonamento apenas sobre $dep(T)$, cujo estado inicial também é S. A compensação CT é dita correta se $E(S) = E'(S)$.

A construção de E', o escalonamento hipotético na ausência de T e CT, se dá da seguinte maneira. Inicialmente, com estado inicial S, começamos a executar o programa das transações em $dep(T)$. Pode ser que o conjunto de transações confirmadas em E e E' seja distinto, e pode ser que os conflitos sejam ordenados de forma diferente. Tudo que a definição requer é que $E(S) = E'(S)$.

Visivelmente, é muito difícil conseguir saber, em geral, se é possível construir uma transação compensatória correta para aquelas aplicações que requerem esta certeza para autorizarem a compensação. No entanto, um caso mais simples pode ser visualizado, onde E' é obtido simplesmente apagando-se de E todas as operações de T e CT. E como garantir que um tal E' satisfaz o critério de correção $E(S) = E'(S)$? A resposta é dada pelo seguinte teorema:

Teorema 5.4.5 *Seja E um escalonamento sobre T, $dep(T)$ e CT. Se cada operação em $dep(T)$ comutar com cada operação em CT, então a compensação é correta.*

Em vez de uma prova formal, vamos dar um exemplo que ilustra a compensação correta e a construção de E'.

Exemplo 5.4.6 Considere as transações T_1 e T_2 do Exemplo 5.4.2 com o escalonamento dado no Exemplo 5.4.3. Claramente, a transação T_1 é dependente de T_2, pois T_1 lê x de T_2.

Suponhamos agora que queremos compensar T_2, e que criamos a seguinte transação compensadora CT_2:

$$CT_2 : [u_3 := z; \quad x := x - u_3; \quad c_3]$$

Note que todas as operações de CT_2 comutam com as operações da transação dependente T_1. Vamos supor que CT_2 que será escalonada após o término de T_1, gerando:

$$\begin{aligned} E_1 = \quad & [T_1 : u_1 := y; \quad T_2 : u_2 := z; \\ & T_2 : x := x + u_2; \quad c_2; \quad T_1 : x := x + u_1; \quad c_1; \\ & CT_2 : u_3 := z; \quad CT_2 : x := x - u_3; \quad c_3] \end{aligned}$$

Por comutatividae, E_1 é equivalente ao escalonamento E_1', dado por:

$$\begin{aligned} E_1' = \quad & [T_1 : u_1 := y; \quad T_2 : u_2 := z; \\ & T_2 : \mathbf{x} := \mathbf{x} + \mathbf{u_2}; \quad c_2; \\ & CT_2 : u_3 := z; \quad CT_2 : \mathbf{x} := \mathbf{x} - \mathbf{u_3}; \quad c_3 \\ & T_1 : x := x + u_1; \quad c_1;] \end{aligned}$$

Como $u_2 = u_3 = z$, as operações em negrito de T_2 e CT_2 se cancelam e portanto é equivalente a

$$\begin{aligned} E_1'' = \quad & [T_1 : u_1 := y; \quad T_2 : u_2 := z; \\ & T_2 : c_2; \\ & CT_2 : u_3 := z; \quad c_3 \\ & T_1 : x := x + u_1; \quad c_1;] \end{aligned}$$

Não há em E_1'' mais nenhuma alteração de item de dados e, portanto, no que se refere ao banco de dados, este escalonamento é equivalente a:

$$E_1''' = [T_1 : u_1 := y; \quad T_1 : x := x + u_1; \quad c_1;]$$

que é o escalonamento obtido removendo-se de E_1 todas as operações de T_2 e CT_2. Desta forma, $E_1 \equiv E_1'''$ e $E_1(S) = E_1'''(S)$ para todo estado S do banco de dados. □

5.4.4 Compensação Relativamente Correta

Vamos supor agora que uma transação compensatória *não* deixe o banco de dados no mesmo estado que teria sido obtido caso nem a transação compensada nem a compensatória tivessem sido executadas. Isto viola a definição de correção acima e, no entanto, é uma situação bastante frequente na prática. Como poderíamos falar de correção neste caso?

Claramente, para podermos falarmos de correção neste caso, deveremos entrar no domínio da semântica das transações, ou seja, o que o conjunto de transações dependentes de T deve realizar e, tentarmos garantir que estes objetivos são atingidos quando T é compensada. Ou seja, tentamos garantir que haja uma *relação semântica* entre dois escalonamentos. Esta seção tenta formalizar esta noção de relação semântica entre estados e apresenta a correção da compensação relativizada à esta relação semântica.

A formulação do problema passa por tentar garantir que a execução com o par transação compensatória/transação compensada obtenha um estado final que esteja *relacionado* com o estado que teríamos obtido se nenhuma das duas transações tivesse sido executada. Neste caso, teremos a noção de correção *relativizada* por esta relação entre os estados finais, que deverá ser uma relação semântica.

Formalmente, temos os seguinte quadro. Seja E um escalonamento envolvendo as transações T, $dep(T)$ e CT, e seja E' um escalonamento envolvendo apenas $dep(T)$. Seja $\mathcal{R}$ uma relação entre estados estendidos, ou seja $\mathcal{R} \subseteq \text{BDA} \times \text{BDA}$. Assumimos ainda que $\mathcal{R}$ é uma relação simétrica, ou seja, $a\ \mathcal{R}\ b \Longleftrightarrow b\ \mathcal{R}\ a$.

Definição 5.4.7 Seja S um estado do banco de dados estendido ($S \in$ BDA). Então:

- Duas seqüências de operações E_1 e E_2 $\mathcal{R}$-*comutam* se a relação $\mathcal{R}$ é preservada pela comutatividade de E_1 e E_2, ou seja, para todo S temos que:
$$(E_1 \circ E_2)(S)\ \mathcal{R}\ (E_2 \circ E_1)(S)$$
- Os escalonamentos E e E' são corretos com relação a $\mathcal{R}$, ou seja, são $\mathcal{R}$-*corretos*, se para todo estado S, temos que $E(S)\ \mathcal{R}\ E'(S)$.

Note que se a relação $\mathcal{R}$ é a igualdade, esta definição se reduz à definição normal de comutatividade e de correção.

A natureza semântica da definição acima está na origem da relação $\mathcal{R}$. Seja $Q_{dep(T)}$ a pós-condição da execução do conjunto de transações $dep(T)$, e seja O um predicado sobre o estado estendido do banco de dados tal que $Q_{dep(T)} \Rightarrow O$. A condição O pode ser encarada como uma restrição de integridade ou uma propriedade que o conjunto de transações $dep(T)$ deve conjuntamente obter. Neste caso, define-se a relação $\mathcal{R}$ de forma que, para dois escalonamentos E e E' e para um estado S:

$$E'(S) \ \mathcal{R} \ E(S) \text{ se e somente se } O(E'(S)) \Longleftrightarrow O(E(S))$$

Vamos supor que E seja um um escalonamento sobre as transações T, $dep(T)$ e CT, e que E' seja um escalonamento sobre $dep(T)$. Neste caso, uma compensação $\mathcal{R}$-correta preserva o valor verdade de O. Vamos supor ainda que E seja equivalente a um escalonamento $D = T \circ D_{dep(T)} \circ CT$. Note que não estamos supondo que E seja seriável, pois $D_{dep(T)}$ não é necessariamente seriável, mas estamos usando o Critério 2 de correção da compensação para garantir que CT ocorre após $dep(T)$. Diremos que $dep(T)$ e CT são *fortemente $\mathcal{R}$-comutáveis* se para todo D como acima e para todo estado S existir um escalonamento E' sobre $dep(T)$ tal que:

$$(CT \circ E')(S) \ \mathcal{R} \ (D_{dep(T)} \circ CT)(S)$$

O principal resultado sobre $\mathcal{R}$-correção é o seguinte:

Teorema 5.4.8 *Seja E como acima. Se $dep(T)$, e CT forem fortemente $\mathcal{R}$-comutáveis então o escalonamento E é $\mathcal{R}$-correto.*

Vamos ilustrar o resultado com um exemplo, sem apresentar sua prova.

Exemplo 5.4.9 Suponhamos que temos um banco de dados com os itens de dados a e b. Suponhamos que a transação T seja simplesmente $T : [a := a + 1]$, e que, portanto, $CT : [a := a - 1]$.

Vamos supor que $dep(T)$ é formado por uma única transação:

$$\begin{array}{ll} dep(T) : [& u := a; \\ & \textsf{if } u \geq 5 \wedge b > 2 \textsf{ then } b := b - 2 \\ & \textsf{elseif } b = 1 \textsf{ then } b := 0] \end{array}$$

Vamos considerar a restrição de integridade $O = (b \geq 0)$, donde derivamos a relação entre estados:

$$S_1 \mathcal{R} S_2 \text{ se e somente se } S_1(b) \geq 0 \Longleftrightarrow S_2(b) \geq 0$$

Como $dep(T)$ apenas lê o valor de a, fica claro que CT e $dep(T)$ são (fortemente) $\mathcal{R}$-comutáveis, e que, muito embora o ramo de $dep(T)$ executado na presença de T pode ser diferente do ramo executado em sua ausência, temos que $E = T \circ dep(T) \circ CT$ é uma compensação $\mathcal{R}$-correta. □

Capítulo 6

Aplicações do Controle de Concorrência Não-Clássico

6.1 Hipóteses Implícitas Revisitadas

No Capítulo anterior, mencionamos como hipóteses implícitas do modelo ACID de transações as seguintes condições.

- transações curtas;
- sequencialidade;
- homogeneidade; e
- disponibilidade.

Apresentamos então diversos modelos que permitem transações longas e iterativas, sem com isso degradar a eficiência do sistema. Cada um destes modelos continha um novo critério de "escalonamentos corretos", uma vez que os critério clássico de seriação exposto no Capítulo 2 não poderia ser diretamente aplicado.

Neste capítulo vamos analisar aplicações que fazer uso destes novos modelos apresentados no Capítulo 5. Estas aplicações têm como característica fundamental a distribuição de dados. Ou seja, na maioria dos casos, sistemas modernos tratam de bancos de dados distribuídos em rede com vários programas e usuários realizando diversas tarefas concorrentemente.

Neste caso, surgem dois novos problemas. O primeiro está relacionado à heterogeneidade dos sistemas interligados em rede. em termos de bancos de dados, tal sistema é chamado de um Multibanco de Dados. Tais sistemas são gerados, normalmente, pela aglutinação de diversos sistemas individuais ou de pequenas redes, para formar uma rede de bancos de dados distribuídos. Obviamente, cada um dos sistemas aglutinados possui características de funcionamento próprias, o dando origem a heterogeneidade entre os bancos de dados assim reunidos.

A principal característica, no que se refere ao controle de concorrência, dos multibancos de dados é a tensão que há entre a *autonomia* dos sistemas no multibanco de dados e a necessidade de realização de tarefas *coordenadas*. A autonomia exige que aplicações locais existentes antes da aglutinação do sistema, ou criada após esta aglutinação, possam continuar a ser executadas sem alterações. Por outro lado, a existência de tarefas globais necessita de concessões de parte desta autonomia para que haja coordenação entre diversos bancos de dados na realização de uma aplicação global.

Por outro lado, quando o sistema distribuído passa a ter elementos móveis, há a desconexão freqüente de participantes da rede. Em um sistema tradicional, a desconexão de um participante normalmente indica a falha do componente. Desta forma, os métodos para se lidar com desconexões ficavam restritos ao tratamento de recuperação de falhas, conforme descrito no Capítulo 3. No entanto, em sistemas móveis, em que usualmente vários participantes corriqueiramente saem da área de abrangência das comunicações sem fio, tal desconexão faz parte da rotina e não pode ser tratada apenas como falha. Tal comportamento está violando o princípio implícito de disponibilidade.

Neste capítulo, estudaremos *aplicações*, e não apenas modelos, que violam os princípio ACID. Apresentamos a seguir alguns exemplos de sistemas que violam as hipótese de transações curtas, sequenciais, sistemas homogêneos e normalmente diponíveis.

Banco de dados para CAD. Em programas de CAD (*Computer Aided Design*: projeto auxiliado por computador), o processamento é altamente iterativo, e a noção de sequencialidade é quase uma excessão. Por exemplo, suponha que temos um programa de CAD para o projeto de um circuito integrado composto por bilhões de componentes, em que o desenho do circuito está armazenado no banco de dados. Vários projetistas podem estar trabalhando no projeto de um mesmo circuito.

Quando um projetista está trabalhando em uma área do circuito, ele executa diversas transações pequenas, e não pode permitir que outros projetistas alterem o seu trabalho. Isto poderia acontecer com o esquema de seriação tradicional, pois uma transação de um outro usuário poderia ser intercalada às diversas transações do projetista. Neste caso, o projetista se considera, temporariamente, o "dono" de uma área do banco de dados, por mais de uma transação.

A porção do banco de dados em que um projetista está trabalhando deve ficar bloqueada para escrita. O projetista pode ficar horas, ou até dias, trabalhando numa mesma área, e isto constitui uma única transação de projeto, composta por uma série de pequenas transações "operacionais".

Neste caso, a exclusividade de uso de uma área do BD é necessária, pois evita que o trabalho de uma semana seja destruído inadvertidamente por um outro usuário. Por este mesmo motivo, uma transação de projeto tão longa não pode ser abortada. Vê-se que o controle de concorrência tradicional não pode ser aplicado a um sistema de CAD como o descrito.

Transações Grupais ou *Groupware* A situação anterior se complica ainda mais quando uma transação recebe dados de mais de um usuário. Isto ocorre nos chamados *groupwares*, programas concebidos para permitir que um grupo de pessoas possa simultaneamente trabalhar sobre um mesmo objeto.

No caso de um groupware para CAD, o grupo deve ter um bloqueio na parte do sistema em que seus membros estão trabalhando. Parte das alterações realizadas pelo grupo são "negociadas" entre os membros do grupo durante a transação. Dentro do grupo, no entanto, pode haver áreas de bloqueio exclusivas de um único usuário, ou de um outro subgrupo.

Dentro de uma transação longa de um grupo pode haver outras *transações aninhadas*, que podem interfacear com um usuário, com um subgrupo, ou podem ser do tipo seqüencial. Estas transações aninhadas são ditas *filhas* da transação longa, que é chamada transação *mãe*. Este processo foi descrito na Seção 5.2 do Capítulo 5.

Conforme visto, se uma transação filha é abortada, e desfeita, não é necessário que a transação mãe seja abortada também. A transação mãe pode decidir por abortar-se, ou reiniciar a subtransação abortada, ou ignorar o aborto, ou qualquer outro procedimento possível.

Desta forma, cria-se uma *hierarquia de transações aninhadas*, em que cada transação pode ter um conjunto de transações filhas, que por sua vez também podem ter suas filhas, etc. As transações aninhadas constituem um novo modelo de transações, diferente do modelo tradicional, como foi descrito na Seção 5.2.

6.1.1 Homogeneidade.

Conforme mencionado, outras hipótesse implícitas do modelo tradicional estão relacionadas a sistemas distribuídos. Tais sistemas são implicitamente assumidos com algum grau de *homogeneidade.* Isso não significa que todos os bancos de dados de um sistema distribuídos devam sempre ser do mesmo tipo ou do mesmo fabricante, mas assume-se que, pelo menos, todos eles sejam capazes de suportar o protocolo de Confirmação Bifásica (conforme descrito na Seção 3.6.2.

No entanto, muitos dos bancos de dados distribuídos que se obtém na prática são compostos pela aglutinação de bancos de dados existentes, chamados *multi bancos de dados*, e esta hipótese de homogeneidade acaba sendo violada.

Multibancos de Dados (MBDs)

As características básicas dos MBDs são a *heterogeneidade* e a *autonomia.* Desta forma, é possível que muitos dos BDs componentes de um MBD não possuam nenhum suporte ao controle de dados distribuído, pois foram concebidos como unidades autônomas e, mesmo após sua incorporação a um MBD, continuarão a ser empregados por aplicações locais, independentes dos outros bancos de dados. Eventualmente, este componente do MBD pode ter parte de seus dados acessados por uma aplicação distribuído . Neste caso, é bem capaz que alguns componentes do MBD não tenham capacidade de suportar o protocolo de Confirmação Bifásica.

Considere uma transação global rodando num MBD. Se algum dos BDs participantes nesta transação não suportar o protocolo de confirmação bifásica, a transação será confirmada localmente ao receber um pedido de confirmação. O problema é que algum outro participante da transação distribuída pode decidir abortar a transação, e então teremos uma inconsistência: a transação estará confirmada em alguns participantes e abortada em outros. Precisamos, portanto, de algum outro método para controlar a concorrência de transações em sistemas heterogêneos.

6.1.2 Disponibilidade.

Outra hipótese básica do funcionamento de um banco de dados distribuído é que todas os participantes de uma transação global estão disponíveis durante a toda a duração da transação.

No entanto, existem aplicações mais modernas para as quais esta hipótese é violada. Note que, se a transação é longa, fica cada vez mais difícil garantir que todos os BDs participantes desta transação fiquem conectados à rede durante toda a duração da transação.

Sistemas de Computação Móvel

Em computação móvel, é muito comum que alguns dos nós móveis tornem-se temporariamente indisponíveis, por exemplo, ao entrarem em túnel (onde as ondas de rádio não chegam) ou entrarem em uma área não coberta pela rede de comunicação. Neste tipo de aplicação não é aceitável toda a transação seja abortada toda vez que um dos seus participantes móveis estiverem temporariamente indisponíveis. O estado da transação pode se tornar indeterminado, onde alguns nós estão abortados e outros estão indisponíveis. Uma transação pode ficar em suspenso, tentado se confirmar, até que todos os participantes indisponíveis tornem-se disponíveis.

6.1.3 A Organização do Capítulo

Veremos a seguir, como as aplicações acima descritas (bancos de dados de Groupware CAD, Multibancos de dados e Bancos de Dados Móveis) tratam o problema do controle de concorrência.

Para o caso do banco de dados de CAD, estudaremos na Seção 6.2 as *transações partidas*. Em seguida, nostraremos na Seção 6.3 como transações hierárquicas e partidas podem ser utilizadas no controle de concorrência de transações em multibancos de dados.

Por fim, na Seção 6.4 apresentaremos um novo modelo de transações para bancos de dados móveis. Tal modelo é chamado de *Modelo Canguru* e é um modelo de alta complexidade e interesse pois envolve *todos* os conceitos de controle de concorrência não ACID vistos neste livro. As transações canguru são aninhadas, suportam a operação de partição e possuem um modo compensatório, servindo de resumo integrado de todos os conceitos vistos numa aplicação com grande potencial prático. Este é o motivo pelo qual decidimos encerrar a nossa apresentação de transações não-ACID com a apresentação do Modelo Canguru.

6.2 Transações Partidas (Split Transactions)

As transações partidas (*split transactions*) foram propostas originalmente para lidar com transações de durações muito longas, de final indeterminado, em ambientes de CAD [PKH88]. O modelo de transações tradicional sempre supõe que as transações terminam. No entanto, para atividades em que este final não é vislumbrado mp horizonte próximo, o modelo de transações partidas permite que a transação requeira uma operação de *partição.*

A operação de partição divide uma transação em andamento em duas transações seriáveis. O modelo de transações partidas pode ser aplicado tanto no modelo traditional de transações simples, como no modelo de transações aninhadas.

A Figura 6.1 ilustra a operação de partição executada sobre o modelo de transações aninhadas. Numa primeira etapa, a transação T_1 ainda está “inteira”, ou seja, a transação não foi partida e se comporta como uma transação aninhada normal.

Numa segunda etapa, a transação T_1 requisita uma operação de partição. A partição ocorre apenas por requisição, nunca “naturalmente” ou por uma força externa. Um dos motivos que a transação pode ter para requisitar a partição é o conhecimento interno de que uma parte de suas tarefas ja’ está completa, ou então que uma parte de suas tarefas já pode ser completada independentemente das operações a serem realizadas futuras.

A transação então é partida em T_1 e T_2 e T_1 não mais executará nenhuma operação. Como T_1 é uma transação interna, isto quer dizer que nenhuma nova subtransação será iniciada a partir de T_1. Uma nova transação, no nível imediatamente inferior a T_1 somente poderá ser criada a partir de T_2, gerando T_{21}. No entanto, as subtransações descendentes de T_1 podem ainda “terminar” suas tarefas, e a Figura 6.1 ilustra tal fato pela criação de T_{121} e T_{122} após a divisão de T_1.

Em seguida, a transação T_2 pode requisição outra partição, e assim criar T_3. Note que, após a partição de T_1, algumas de suas descendentes podem abortar. T_1 não pode mais gerar novas transações, mas ainda pode decidir abortar totalmente. Fica a cargo da aplicação decidir se T_2 e T_3 devem ser abortadas.

Por outro lado, se T_3 abortar após a confirmação de T_1 e T_2, fica tambéma cargo da aplicação decidir se as suas fases anteriores devem ser compensadas ou não. Ambas as possibilidades são contempladas, e o modelo não nos força a nenhuma delas. Mas note que se a optar-se por não compensar as transações confirmadas, estaremos violando a atomicidade.

É importante notar que a partição de transações *não é equivalente* a termos uma transação aninhada hipotética, digamos, T_0, que é mãe de T_1, T_2 e T_3. Independentemente de haver compensação ou não, *as transações partidas não preservam o isolamento* das outras transações no sistema. Uma transação partida se comporta como um série de transações seriáveis, não como uma única transação aninhada T_0. O problema está em que, por exemplo, quando T_1 se confirma, seus bloqueios são liberados e podem ser

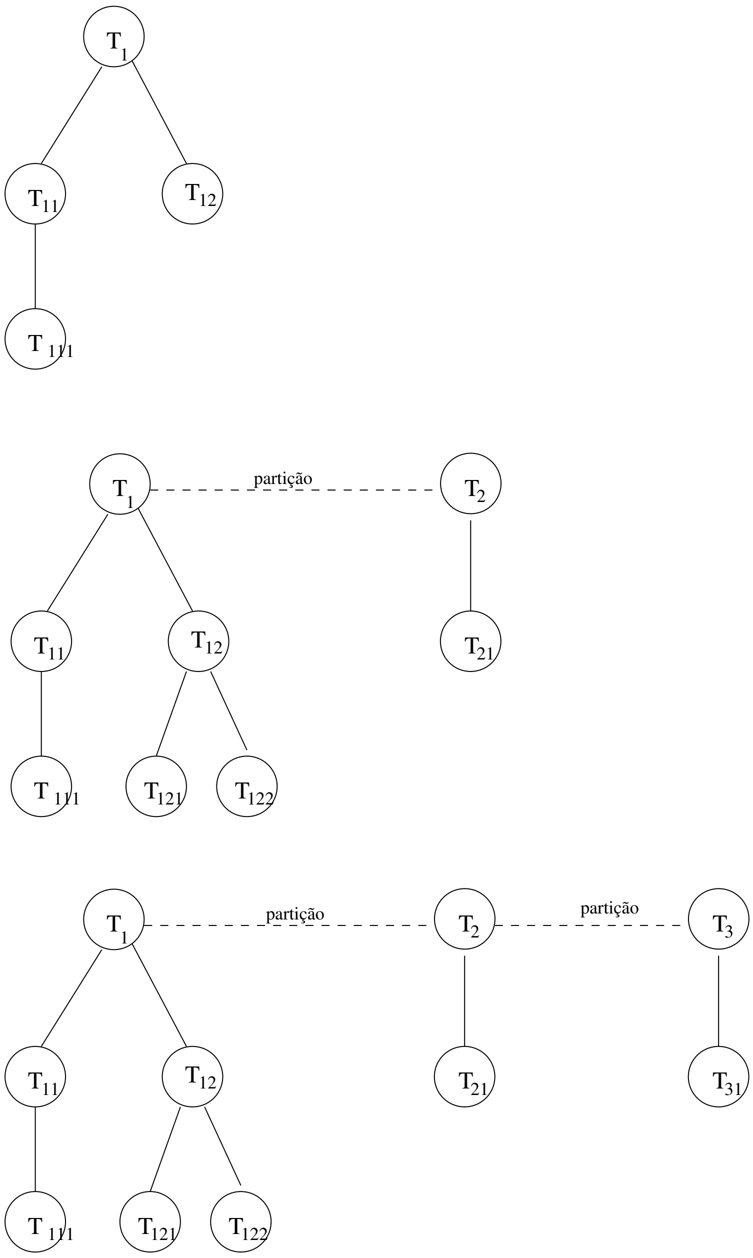


Figura 6.1: Três etapas de uma transação aninhada partida

adquiridos por uma outra transação, que em seguida se confirma e os libera, para serem então adquiridos por T_3. Isto jamais aconteceria se a transação não fosse partida, pois os bloqueios de T_1 passariam para T_0, e de lá para para T_3, sem serem "emprestados" a nenhuma outra transação do sistema. Desta forma, não podemos dizer que a sequência T_1, T_2, T_3 está sendo executada isoladamente das outras transações do sistema.

Por outro lado, é obvio que a partição de transações permite um grau de concorrência muito maior. Ou seja, ao adotarmos transações partidas, estamos adotando um compromisso: estamos sacrificando a indenpendência da transação como um todo para obtermos maior grau de concorrência, sem com isso abandonar totalmente a noção de seriabilidade, uma vez que as transações partidas são seriáveis entre si.

As transações aninhadas partidas são utilizadas pelo sistema de transações em ambientes de computação móvel descritos em 6.4.

6.3 Transações em Multibancos de Dados

Nesta seção vamos discutir um possível uso de transações aninhadas para o processamento de transações em *multibancos de dados* (MBDs). Por multibanco de dados entendemos, seguindo [OV99], um sistema de bancos de dados distribuídos em que cada banco de dados envolvido possui *autonomia* para rodar aplicações locais (que não são afetadas pela participação do BD no multibanco de dados), bem como participar de transações distribuídas. Frequentemente, os sistemas de multibancos de dados também são *heterogêneos*, ou seja, os diversos bancos de dados participando no MBD possuem diferentes modelos de dados, linguagens de consulta e gerenciadores de transações.

Para lidar com transações em sistemas de multibancos de dados (posivelmente heterogêneos) apresentamos um refinamento do sistema de transações aninhadas no qual as transações internas são de dois tipos:

- *Transações Locais* (TLs): são transações cujas operações básicas, situadas nas folhas da hierarquia, são executadas todas num mesmo banco de dados. Os descendentes de uma transação local podem ser apenas operações sobre dados, conforme ilustrado na Figura 6.2, onde podemos ver que a transação local TL_1 possui como filhas apenas operações aos dados. Todas as operações op_1, op_2, op_3 necessariamente referem-se ao mesmo banco de dados.

- *Transações Globais* (TGs): São transações internas cujas operações

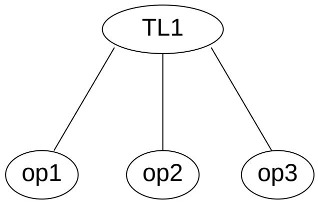


Figura 6.2: Transações locais em multibancos de dados

básicas, situadas nas folhas da hierarquia, podem ser executadas em mais de um banco de dados participante do MBD. As filhas diretas de uma TG outras TGs ou TLs; uma TG *não* pode ter como descendente direta uma operação em algum banco de dados.

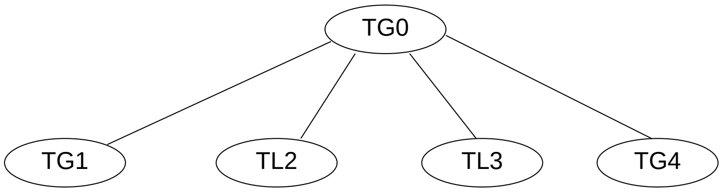


Figura 6.3: Transações locais em multibancos de dados

Por exemplo, na Figura 6.3, a transação global TG_0 possui quatro descendentes diretas, as transações locais TL_2 e TL_3 e as transações globais TG_1 e TG_4; estas duas últimas necessariamente possuem como descendentes outras transações internas, que podem, por sua vez, ser tanto transações locais ou globais.

Desta forma, uma operação a algum DB participante no MBD é sempre descendente direta de uma transação local. Tudo o que assumimos do sistema local de gerenciamento de transações é que este gere escalonamentos seriáveis. Destas forma, encapsulamos nas transações locais as características heterogêneas dos gerenciadores de transação em cada um dos bancos de dados.

Os sistemas locais permanecem autônomos, pois as aplicações que são localmente executadas percebem as transações distribuídas apenas como transações normais do sistema.

No entanto, isto não resolve o problema do ponto de vista das transações distribuídas no MBD. Se uma TG possuir duas descendentes TL_1 e TL_2 que acessam o mesmo banco de dados, embora o gerenciamento local garanta que estas duas LTs são seriáveis, ainda assim poderemos ter uma violação na correção do escalonamento aninhado de GT, pois uma transação autônoma local TAL_3 pode ser seriada entre TL_1 e TL_2. Uma maneira de evitar tal problema é garantir que uma TG possua no máximo uma TL descendente para cada BD do MBD; desta forma, a transação autônoma local, para efeitos do mecanismo de serição, pode ser vista também como uma transação global do MBD.

Toda transação no MBD inicia-se como uma transação global. Para garantir que os escalonamentos de transações sejam corretos no MBD é necessario garantir que haja algum controle extra nos nós internos (e não apenas o controle local nos diversos BDs), de forma a se respeitar o controle de transações aninhadas descrito na Seção 5.2. Como tal protocolo é baseado na existência de tabelas de bloqueio, isto faz copm que haja um certo grau de replicação das informações já mantidas nos gerenciadores locais (assumindo que estes gerenciadores operem por bloqueios também). Além disso, uma transaçãdo local só poderá emitir uma operação local após obter o bloqueio de sua transação global imediatamente superior. Esta burocracia extra é o preço que se paga para podermos gerenciar transações em multibancos de dados heterogêneos, e permitir autonomia aos bancos de dados locais.

6.4 O Modelo de Canguru de Transações Móveis

Reservamos para a seção final deste capítulo o modelo de transações em ambiente de computação móvel conhecido como *Transações Canguru* e apresentando em [DHB97]. Este sistema combina em, um único modelo, todos os aspectos de transações não-ACID vistos até agora, ou seja,

- transações hierárquicas;
- transações compensatórias;
- transações partidas;
- multibancos de dados heterogêneos; e, além disso,
- mobilidade.

Antes de falarmos sobre transações em sistemas de computação móvel, precisamos entender exatamente como é a arquitetura de um sistema móvel. Estamos nos baseando aqui na arquitetura apresentada em [DHB97].

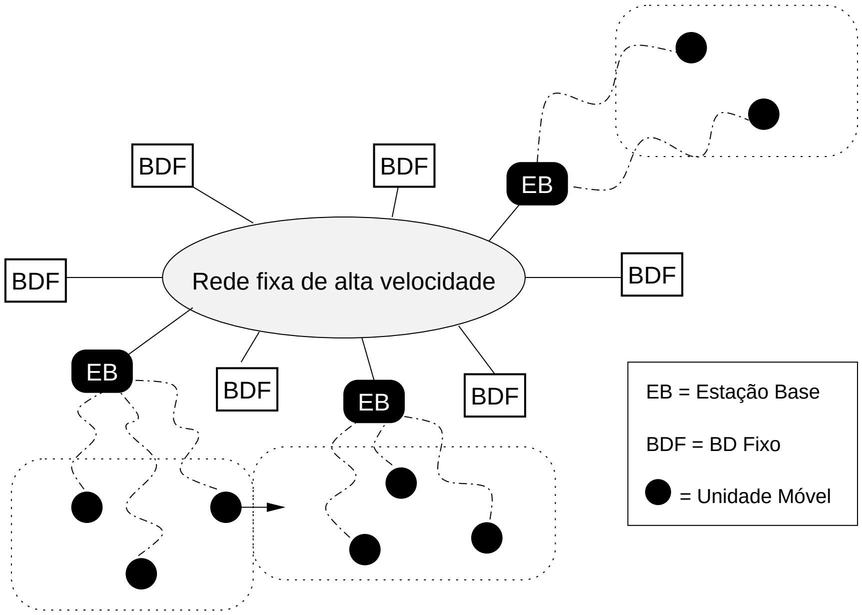


Figura 6.4: Sistema de bancos de dados móveis

A figura 6.4 ilustra os aspectos básicos de um sistema de bancos de dados distribuído móvel. No centro do sistema está uma rede de computadores fixa de alta velocidade; esta rede pode abrigar um ou mais multibancos de dados ou serviçoes de informações que compõe a camada do *sistema básico* do banco de dados móvel, a ser detalhada abaixo.

Fisicamente conectados a esta rede, temos os diversos bancos de dados fixos (BDFs). Até aí, nenhuma diferença com um banco de dados distribuído usual. No entanto, ligados à rede temos também *Estações Bases* (EBs). Estas estações servem como porta de entrada para a rede para as diversas *Unidades Móveis* (UMs), que usalmente são computadores portáteis que se conectam a uma estação base por meio de cominicação sem fio.

Cada estação base pode capturar unidades móveis dentro de uma certa área de abrangência, chamada *célula*, representada na Figura 6.4 pela área pontilhada. Cada UM está conectada a no máximo uma EB e esta conexão está representada na Figura 6.4 pela linha tracejada conectando a UM ao uma EB. Também é póssível que uma UM fique temporariamente desconectada do sistema devido a uma dentre várias razões:

- a UM está fora de alcance de qualquer estação base, ou entrou dentro de um túnel;

- a bateria do computador portátil acabou;
- o computador portátil foi desligado; etc

Devido a sua movimentação, uma UM pode sair de uma célula e entrar na outra sem se desconectar da rede. Este fenômeno é chamado de *handoff*, e ocorre quando uma UM muda sua porta de entrada para a rede de uma EB para a outra. Na figura 6.4, o handoff de uma UM está representada pela UM na célula mais a esqueda junto a uma seta, indicando o seu deslocamento para a célula da direita.

O modelo, por simplificação, adota a seguinte restrição. Uma UM pode iniciar uma transação distribuída no sistema como um cliente qualquer, mas esta transação será executada apenas nos BDs Fixos. Ou seja, uma UM não possui um sistema de banco de dados acessível à rede. À parte isso, os BDFs ligados a rede são genéricos e podem ser heterogêneos e ter autonomia para rodar aplicações locais.

O modelo de referência para bancos de dados móveis consiste de três camadas:

- O *Sistema Básico*, composto pela coleção de sistemas que oferecem serviços de informação a usuários móveis. Estes serviços podem ser de páginas amarelas, informaçoes sobre o trânsito, bolsas de valores ou qualquer sistema privado de informações. Um sistema nesta camada pode ser qualquer sistema estacionário existence, que funcione nos moldes de comunicação cliente/servidor ou ponto-a-ponto. Um sistema de banco de dados distribuído é um exemplo perfeito de Sistema Básico.

- O *Agente de Acesso a Dados*(AAD) serve de interface entre uma transação móvel e o sistema básico. Cada Estação Base necessariamente abriga um AAD. O AAD não mantém informação sobre a localização de usuário móvel, embora ele vá precisar de tal informação. Quando ele recebe um pedido de transação de um usuário móvel (o qual chamamos, apropriadamente, de *transação móvel*, a qual assume-se do tipo aninhado), o AAD redireciona o pedido para a EB ou o BDF específico que contém o dado ou sistema básico requisitado. Num caso simples, toda as requisições são repassadas para um mesmo elemento do multibanco de dados que determina onde serão executadas as subteransações. Numa situação mais genérica, é o próprio AAD que decide onde processar cada requisição.

Do ponto de vista do processamento de transações, o *handoff* faz com que uma transação "salte" de uma estação base para outra. Quando uma unidade móvel passa para outra EB, o AAD na nova estação recebe as informações sobre as transações daquela UM em andamento na velha estação. Desta forma, a principal função do AAD é a gerência de transações móveis, o que inclui manter informações sobre as transações ativas que passaram pelo AAD e manter um log das operações requisitadas a partir deste AAD.

- Uma *Transação Móvel* é a unidade básica de processamento no ambiente móvel. A transação móvel é identificada pelo coleção de EBs pelas quais ela saltou, mas o conjunto completo só é conhecido quando a transação termina.

  Uma transação móvel é uma transação aninhada com características especiais que discutiremos a seguir.

Os grandes problema deste sistema, em termos de processamento de transações móveis, está nas desconexões frequentes das UMs e nos handoffs. As desconexões frequentes podem levar a transações de duração muito longas, e os *handoffs* podem levar a uma sobrecarga de comunicação no sistema, com uma operação de leitura iniciada em EB1 tendo que ter sua resposta redirecionada para EB2, e desta para EB3, etc, até se encontrar a EB que está correntemente ligada a UM que solicitou o dado. Para tratar destes problemas, foi proposto o modelo de *Transações Canguru* em [DHB97], que visa atender aos seguintes princípios básicos:

- Apoiar-se em sistemas existentes multibancos de dados, sem duplicar as funções providas pelo sistema básico.
- Capturar a movimentação das transações móveis, e não apenas o acesso a dados. O controle da transação deve mover-se quando a transação se mover.
- Prover flexibilidade em termos da propriedade de atomicidade.
- Suportar transações longas.

6.4.1 Transações Canguru

O modelo de transações canguru se apoia sobre aquele desenvolvido para multibancos de dados na Seção 6.3, envolvendo Transações Globais e Transações Locais. As transações globais, no entanto, não capturam os saltos dados por uma tranasação móvel.

Desta forma, uma transação canguru subdividi-se em uma sequência de *transações filhotes*[1]. Cada transação filhote é executada em uma mesma estação base e se comporta como uma transação global de alto nível, podendo ter subtransações globais ou locais; a transação filhote é coordenada pelo AAD da estação base em que ela reside. Quando uma UM se move de uma célula a outra, o controle da transação canguru se transfere para uma outra estação base, criando-se uma nova transação filhote.

A formação de uma transação filhote se dá através de uma operação de partição, conforme descrito na Seção 6.2, em ambos os modos de operação descritos abaixo. A operação de partição é requisitada por uma transação no momento do *handoff*.

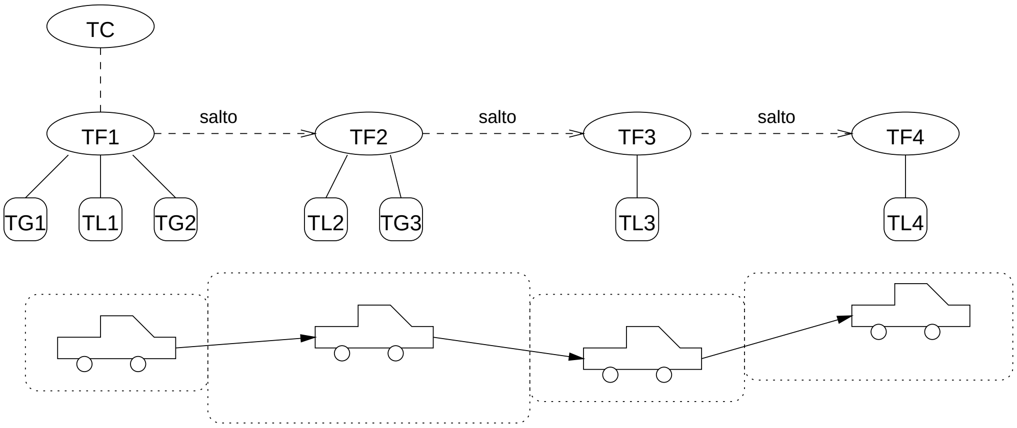


Figura 6.5: Uma transação canguru e sua relação com a movimentação de um usuário móvel

Um exemplo de transação canguru está ilustrado na Figura 6.5. Nela vemos um automóvel, supostamente contendo um computador portátil, se deslocando dentre diversas células. A cada mudança de célula, uma nova transação filhote é criada, e pode ter subtransações locais e globais. A confirmação de uma transação filhote se dá independentemente da outra. Caso TF3 aborte quando TF1 já está confirmada, pode-se desejar desfazer toda a transação canguru TC; neste caso, a única coisa a se fazer é emitir uma transação compensatória pata TF1.

Existem dois modos de processamento para uma transação canguru: o *modo compensatório* e o *modo partido*.

[1] O nome original é *joey transaction*, onde *joey* é o nome em inglês para o filhote de canguru.

Quando uma transação canguru é executada no modo compensatório, o aborto de uma transação filhote causa o desfazimento desta transação filhote e de todas as outras transações filhotes em TC. Transações filhotes já confirmadas terão de ser compensadas.

O modo partido é o modo padrão. Neste modo, se há uma falha em alguam transação filhote, outras transações filhotes já confirmadas não serão compensadas. Transações filhotes ativas podem terminar or serem desfeitas, e a decisão é deixada a cargo de cada transação filhote. Ou seja, a seriação da transação não é garantida neste modo.

No modo compensatório, teremos a seriação das transações filhotes. Com já notamos na Seção 6.2, isto não é equivalente a dizer que a transação canguru é uma transação aninhada seriável.

Por fim, definimos a noção de *equivalência* entre duas transações canguru. Esta noção tem a seguinte idéia subjacente: duas transações são equivalentes se seus efeitos são os mesmos, independentemente de onde estava o usuário na hora da requisição das operações. Desta forma, definimos a *bolsa* de uma transação canguru como a sequência de transações globais e locais executadas pela transação canguru. Então, duas transações canguru são equivalentes se possuem a mesma bolsa.

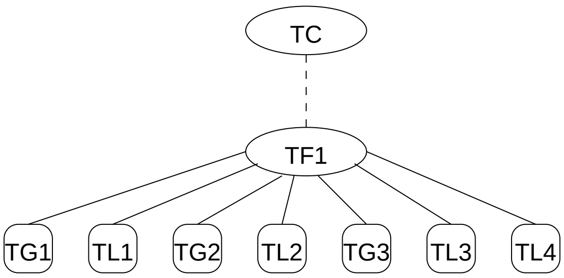


Figura 6.6: Uma transação canguru equivalente àquela da Figura 6.5

A Figura 6.6 mostra uma transação canguru com apenas uma transação filhote equivalente àquela da Figura 6.5 que possui quatro filhotes.

Capítulo 7

Modularização de Bases de Dados

7.1 Introdução

Uma maneira de permitir o desenvolvimento de sistemas flexíveis é a construção de sistemas modulares, onde cada unidade operacional de uma organização escolhe os módulos que lhe convém, podendo escolher diferentes versões de um módulo para cada função.

Considerando a fase de projeto de um sistema dividida em projeto funcional e projeto da base de dados, a modularidade, é tratada pelo projeto funcional do sistema, onde geram-se módulos compostos por transações que irão operar diretamente sobre a base de dados. Por sua vez, o projeto da base de dados segue considerando um esquema global de dados a ser implementado através da utilização de um repositório de dados comum a todos os módulos do sistema.

No caso de uma aplicação composta por vários subsistemas, a existência de um repositório de dados comum a todos esses subsistemas significa que, independente de partes da base de dados serem acessadas por apenas um subsistema, os dados estão armazenados de forma monolítica, acarretando a perda de autonomia por parte dos subsistemas envolvidos.

Para alcançar autonomia de gerenciamento de dados por parte dos subsistemas que compõem uma aplicação é necessário modularizar a base de dados, fazendo com que cada módulo gerado possa ter seu próprio repositório de dados que conterá apenas os dados relevantes as suas transações.

Este capítulo trata do problema de modularização de bases de dados, propondo diretrizes para a realização do projeto de modularização. De forma a contemplar a necessidade de interoperabilidade de informação, o projeto de modularização é tratado de forma top-down, garantindo a existência de um padrão mínimo de funcionalidade e estrutura da informação. A modularização é, então, incorporada ao processo genérico de projeto de base de dados encontrados em [CP84], [Nav92] e [EN00],através da inclusão de duas novas fases nesse processo.

- *A idéia básica da modularização de bases de dados é a decomposição do esquema global de dados da aplicação em subesquemas conforme [FB99] e [BF99]. A interseção de subesquemas caracteriza o compartilhamento de porções do esquema global de dados.*

7.2 Processo de Modularização de Bases de Dados

Um processo genérico de projeto de bases de dados segundo [SCW87], [EN00] inclui quatro fases, a serem realizadas na ordem apresentada:

- **Coleta e Análise de Requisitos:** consiste em identificar e entender os requisitos de dados e funcionais da aplicação, definindo, de forma não-ambígua, os elementos a serem considerados no projeto de dados e as transações a serem executadas nesses dados.

- **Projeto Conceitual:** envolve duas atividades paralelas. A primeira atividade consiste em examinar os requisitos de dados resultantes da fase anterior e produzir, ignorando detalhes de implementação, o esquema conceitual de dados. A segunda atividade consiste em examinar os requisitos funcionais obtidos na fase anterior e produzir especificações para as transações definidas, independente do SGBD a ser utilizado.

- **Projeto Lógico:** consiste em transformar o esquema conceitual de dados em um esquema específico para um dado tipo de SGBD, denominado esquema lógico de dados.

- **Projeto Físico:** consiste em escolher estruturas de armazenamento e acesso específicas para o SGBD a ser utilizado, visando atender os requisitos de desempenho dos sistemas que compõem a aplicação e que irão interagir com a base de dados.

A modularização de bases de dados será, então, contemplada com a inclusão de duas novas fases no processo de projeto de bases de dados, como mostrado na Figura 7.1:

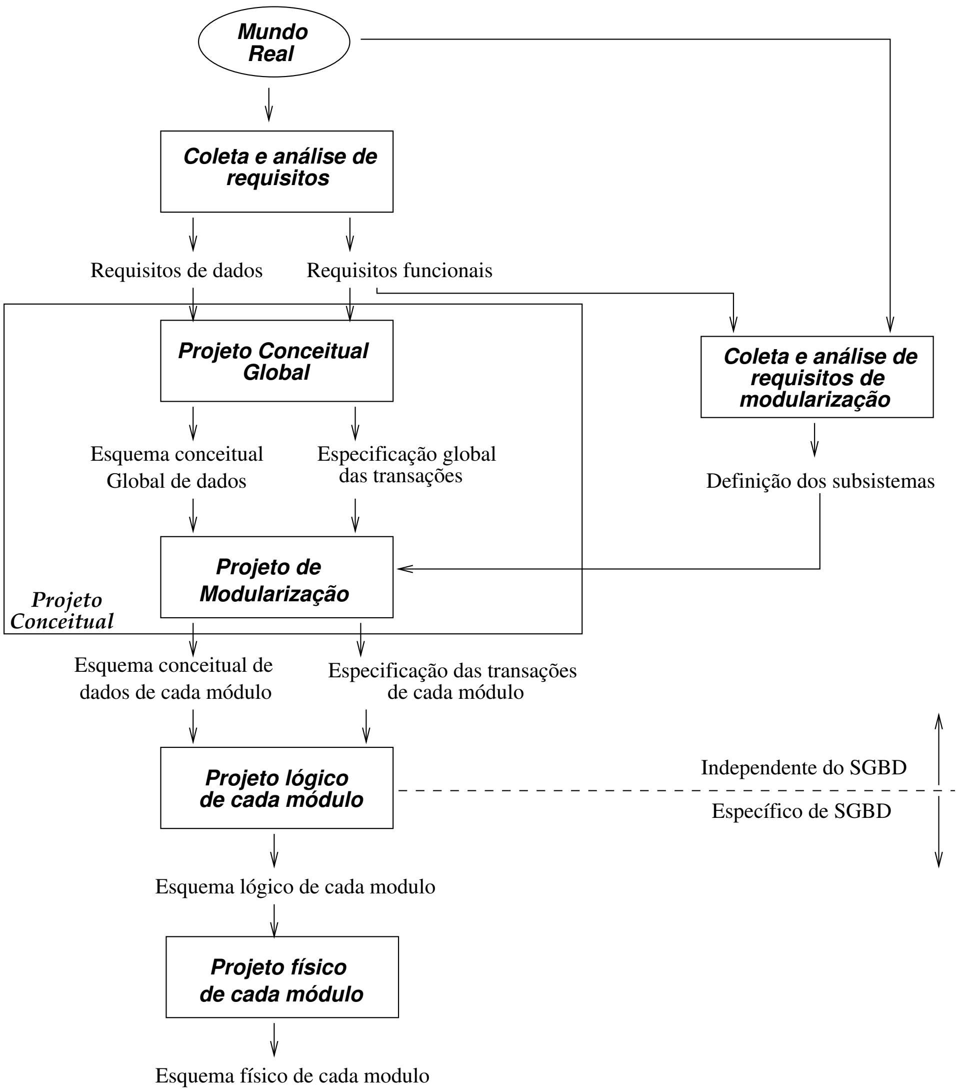


Figura 7.1: Fases do Projeto de Modularização de Dados

- **Coleta e Análise dos Requisitos de Modularização:** tem por objetivo identificar e entender os requisitos de modularização da aplicação, definindo os subsistemas que a compõem e associando a cada um deles, de acordo com as suas funções, as transações definidas na fase de Coleta e Análise de Requisitos.

- **Projeto de Modularização:** tem por objetivo modularizar a base de dados, associando os dados aos subsistemas que compõem a aplicação de acordo com as transações executadas por esses subsistemas, contemplando a necessidade de compartilhamento e interoperabilidade da informação. Para poderem interoperar, os subsistemas precisam atender a um padrão mínimo de operacionalidade e estrutura da informação, havendo a necessidade da construção de um esquema de dados de toda a aplicação. Desse modo, essa fase assume como entrada o esquema global de dados da aplicação, o qual será decomposto gerando subesquemas correspondentes às necessidades operacionais dos subsistemas. A interseção de dois ou mais subesquemas caracteriza o compartilhamento de porções do esquema conceitual global de dados, que deve ser contemplado no processo de geração dos módulos da base de dados. Considerando que para poder ser decomposto é necessário que o esquema de dados tenha sido construído segundo um modelo de dados que atribua semântica à modelagem, a fase de Projeto de Modularização deve ser realizada durante a fase de Projeto Conceitual, utilizando o esquema conceitual global de dados para realizar o particionamento. Como a fase de Projeto de Modularização vai gerar os esquemas conceituais de dados de cada módulo, a fase de Projeto Lógico deverá ser aplicada a cada um desses esquemas conceituais, gerando esquemas lógicos de dados para cada módulo. Por sua vez, a fase de Projeto Físico deverá ser aplicada a cada esquema lógico gerado pela fase de Projeto Lógico, gerando esquemas físicos de dados para cada módulo.

7.2.1 Coleta e Análise de Requisitos de Modularização

Durante a fase de Coleta e Análise de Requisitos de Modularização deve-se definir os subsistemas que compõem a aplicação e que serão utilizados por cada unidade operacional da empresa.

Assim, com base nas informações obtidas no próprio domínio da aplicação e nos requisitos funcionais especificados na fase de Coleta e Análise de Requisitos, identificam-se os subsistemas e associam-se a cada um deles as transações que atendam as suas necessidades operacionais.

7.2.2 Projeto de Modularização

Durante a fase de Projeto de Modularização deve-se identificar os módulos de dados a serem acessados pelos subsistemas que compõem a aplicação e gerar os esquemas conceituais de dados de cada um desses módulos, contemplando o compartilhamento e a interoperabilidade da informação. A fase de Projeto de Modularização é composta por quatro etapas a serem cumpridas na ordem apresentada:

Etapa 1 - Particionamento do esquema conceitual global de dados

Essa etapa consiste em decompor o esquema conceitual global de dados da aplicação construído na fase de Projeto Conceitual, com base nas especificações dos subsistemas e das transações que executam, obtidas nas fases de Coleta e Análise de Requisitos de Modularização e Projeto Conceitual, respectivamente. Durante o processo de particionamento, os elementos do esquema conceitual global de dados devem ser agrupados de acordo com as necessidades operacionais de cada subsistema que compõe a aplicação, através de uma análise cíclica entre os dados e as transações que os operam, obtendo-se subesquemas correspondentes aos subsistemas.

É importante observar que para um esquema conceitual de dados poder ser decomposto é essencial que a informação modelada esteja representada da forma mais atômica possível. No caso da utilização de mecanismos de abstração no processo de modelagem, essa atomicidade de um esquema de dados é obtida no nível mais baixo da hierarquia de abstrações, onde a informação se encontra em sua forma mais detalhada. Formalmente, a decomposição do esquema pode ser representa por:

$$DER = \bigcup_{i=1\ldots n} = DER_i$$

Etapa 2 - Tratamento do compartilhamento da informação

Essa etapa consiste em tratar da necessidade de compartilhamento de porções do esquema conceitual global de dados que surge devido à sua decomposição. Esse compartilhamento é caracterizado pela interseção dos subesquemas identificados na etapa 1, ou seja, existem elementos do esquema conceitual global de dados que pertencem a mais de um subesquema e, portanto, seus dados são necessários a mais de um subsistema.

Um subesquema que não possui nenhum elemento compartilhado dá origem a um módulo da base de dados com total autonomia em relação ao gerenciamento de seus dados, não havendo necessidade de nenhum tratamento especial para seus elementos. Nos casos em que há compartilhamento, deve-se analisar separadamente cada um dos elementos dos subesquemas que se sobrepõem. Primeiramente, determinam-se as operações executadas em cada elemento por parte dos subsistemas, as quais podem ser classificadas em: operações de leitura, ou seja, apenas consulta ao elemento, e operações de escrita, ou seja, criação, atualização e remoção do elemento. Em seguida, de acordo com o número de subesquemas aos quais cada elemento pertence e com as operações executadas em cada um, atribui-se a cada elemento o seu tipo de compartilhamento. Assim, quanto ao tipo de compartilhamento pode-se ter:

- não-compartilhado: o elemento pertence a um único subesquema e, portanto, um único subsistema executa operações tanto de leitura quanto de escrita no elemento;
- compartilhamento unidirecional: o elemento pertence a dois ou mais subesquemas e um único subsistema executa operações de escrita no elemento, sendo que os outros executam apenas operações de leitura;
- compartilhamento multidirecional: o elemento pertence a dois ou mais subesquemas e dois ou mais subsistemas executam operações de escrita no elemento.

Formalmente, para cada subsistema S_i tem-se a utilização de um subesquema DER_i. Assim:

$$\begin{gathered}
S_1 \text{ utiliza } DER_1 = \{E_1, A_1, R_1, HG_1, AG_1, C_1\} \\
S_2 \text{ utiliza } DER_2 = \{E_2, A_2, R_2, HG_2, AG_2, C_2\} \\
\vdots \\
S_n \text{ utiliza } DER_n = \{E_n, A_n, R_n, HG_n, AG_n, C_n\}
\end{gathered}$$

Definem-se G_{Sn} como grupos de elementos dos respectivos subesquemas e operações para sintetizar cada um dos DER_n utilizados por S_n.

Etapa 3 - Definição dos módulos da base de dados

Essa etapa consiste em definir os módulos da base de dados a partir das informações sobre compartilhamento obtidas na etapa 2, gerando os esquemas conceituais de cada módulo e determinando um ou mais subsistemas responsáveis pela manutenção de seus elementos.

A manutenção de um elemento está relacionada às operações de escrita executadas no elemento, isto é, os subsistemas que mantêm um elemento são aqueles que executam operações de escrita no elemento. Então, para a definição dos módulos deve-se, primeiramente, agrupar os elementos de acordo com os subsistemas que os mantêm, resultando em um grupo de elementos para cada subsistema. Em seguida, deve-se fazer todas as combinações de interseção entre grupos G_{Sn} obtidos, de modo que:

- grupos cuja interseção com todos os outros é vazia origina diretamente um módulo. Assim, para cada combinação de interseção, se:

  $G_{S1} \cap G_{S2} = \emptyset$ e

  $G_{S1} \cap G_{S3} = \emptyset$ e

  $\vdots$

  $G_{S1} \cap G_{S2} \cap \ldots G_{Sn} = \emptyset$, então cria-se um módulo M_i para cada G_{Si}

- grupos cuja interseção não é vazia devem ter seus elementos unidos, a união gerando um único módulo. Assim, para cada combinação de interseção, se:

  $G_{S1} \cap G_{S2} \neq \emptyset$, então cria-se um módulo $M_i = G_{S1} \cup G_{S2}$ ou

  $G_{S1} \cap G_{S3} \neq \emptyset$, então cria-se um módulo $M_i = G_{S1} \cup G_{S3}$ ou

  $\vdots$

  $G_{S1} \cap G_{S2} \cap \ldots G_{Sn} \neq \emptyset$, então cria-se um módulo $M_i = G_{S1} \cup G_{S2} \cup \ldots G_{Sn}$

Quando a maioria dos elementos de uma interseção não pertence a nenhuma hierarquia de generalização, abstração de agregação ou a qualquer outra hierarquia complexa, pode-se criar até três módulos. Por exemplo:

se $G_{S1} \cap G_{S2} = G_x$, então
$G_{S1} - G_x \Rightarrow M_1$
$G_{S2} - G_x \Rightarrow M_2$
$G_x \Rightarrow M_3$

M_1 é acessado somente por S_1, M_2 é acessado somente por S_2 e M_3 é acessado por S_1 e S_2.

Etapa 4 - Definição da interface dos módulos da base de dados

Essa etapa consiste em definir uma interface de acesso aos dados para cada módulo da base de dados especificado na etapa 3. Para tanto, deve-se criar procedimentos que encapsulem o acesso aos dados de cada módulo da base de dados.

Os procedimentos devem ser criados de acordo com as necessidades operacionais que cada subsistema tem em relação aos dados de cada módulo. Assim, cada subsistema que utiliza dados de um determinado módulo deve fazê-lo somente através dos procedimentos daquele módulo.

Pode-se ainda classificar os procedimentos de cada módulo como privados e públicos. Os procedimentos privados são os responsáveis pela execução de operações de escrita nos dados, sendo, portanto, disponibilizados apenas para os subsistemas responsáveis pela manutenção dos dados do módulo. Enquanto que os procedimentos públicos realizam apenas operações de leitura nos dados, podendo ser disponibilizados para todos os subsistemas que acessam o módulo.

Assim, o esquema de dados de cada módulo deve ter a representação de seus elementos e a especificação de seus procedimentos privados e públicos, como mostrado na Figura 7.2

Para eliminar possíveis confusões existentes em projetos de bases de dados corporativas, foram estabelecidas diretrizes baseadas nas duas hipóteses seguintes:

a) para determinar o grau de compartilhamento dos módulos das bases de dados é necessária uma análise cíclica entre os dados e as transações envolvidas na construção dos sistemas de informação;

b) para a decomposição do esquema conceitual global do esquema de dados é necessária a utilização de um grau detalhado das abstrações de dados.

Com a adição de mais duas fases no processo de projeto de bases de dados e a criação dos critérios para geração dos esquemas conceituais de dados de cada módulo, é possível construir sistemas de informação para atender necessidades as específicas de cada unidade operacional de grandes corporações.

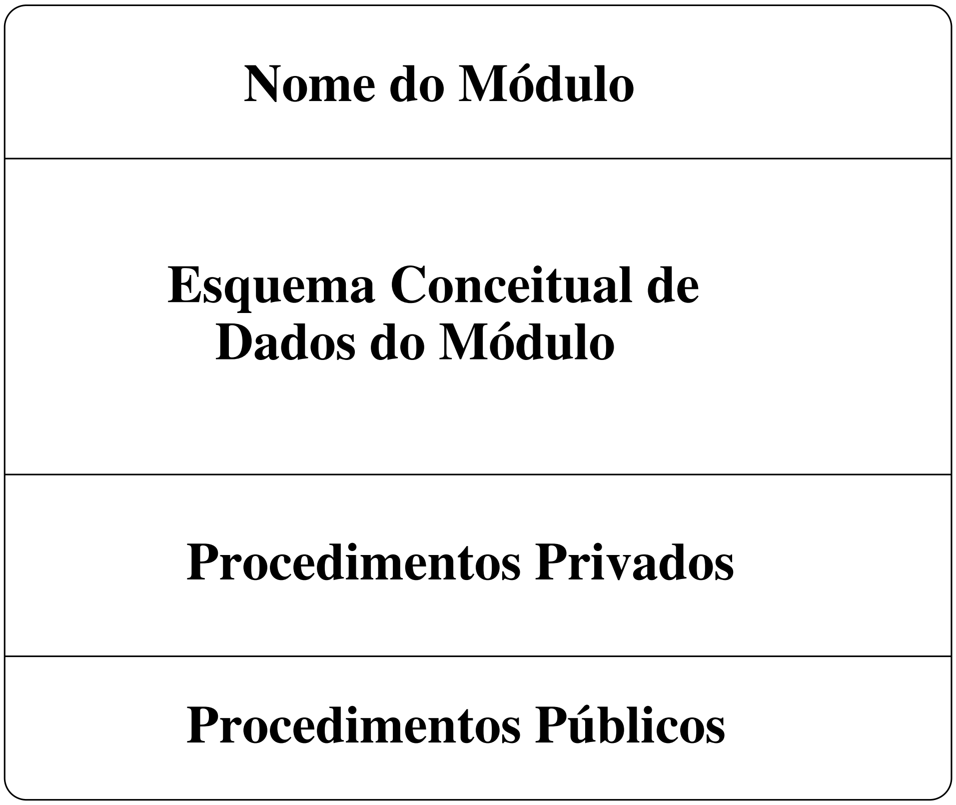


Figura 7.2: Representação de um módulo de dados

7.3 Modularização através de um estudo de caso

O cadastro de serviços médicos da associação de cooperativas médicas em questão é acessado por quatro subsistemas, como descritos a seguir:

- **subsistema** S_1**:** responsável pelo gerenciamento dos contratos que podem ser feitos entre uma determinada cooperativa médica e uma pessoa física ou jurídica, suportando planos de saúde e atendendo a beneficiários;
- **subsistema** S_2**:** responsável pelo gerenciamento dos diversos planos de saúde mantidos pelas cooperativas médicas, bem como das coberturas providenciadas por cada um deles em relação aos vários serviços médicos;
- **subsistema** S_3**:** responsável pelo gerenciamento dos dados dos diversos prestadores de serviços médicos vinculados a cooperativas médicas;
- **subsistema** S_4**:** responsável pelo gerenciamento dos dados das pessoas físicas e jurídicas que mantêm contratos com as cooperativas médicas, das cooperativas médicas propriamente ditas e dos beneficiários vinculados a cada contrato;

O esquema conceitual global de dados do cadastro em questão pode ser encontrado na 7.3. Deve-se ressaltar, entretanto, que esse esquema de dados foi simplificado e as especificações das transações realizadas pelos subsistemas que compõem a aplicação foram omitidas, pois o objetivo aqui é exemplificar a modularização da base de dados e não detalhar a aplicação. Assim, de posse da definição dos subsistemas que compõem a aplicação e do esquema conceitual global de dados, parte-se para a fase de projeto de modularização, onde deve-se cumprir as seguintes etapas especificadas na seção 7.2.2.

Etapa 1 - Particionamento do esquema conceitual global de dados: com base na funcionalidade de cada subsistema que compõe a aplicação, realiza-se uma análise cíclica entre os dados e as transações que os operam e agrupam-se os elementos do esquema conceitual global de dados, obtendo-se os subesquemas E1, E2, E3, E4, que correspondem aos acessos realizados aos dados pelas transações dos subsistemas S1, S2, S3, S4, respectivamente, como mostrado na Figura 7.3.

Etapa 2 - Tratamento do compartilhamento da informação: pode-se observar na Figura fig3 que existem interseções de subesquemas,

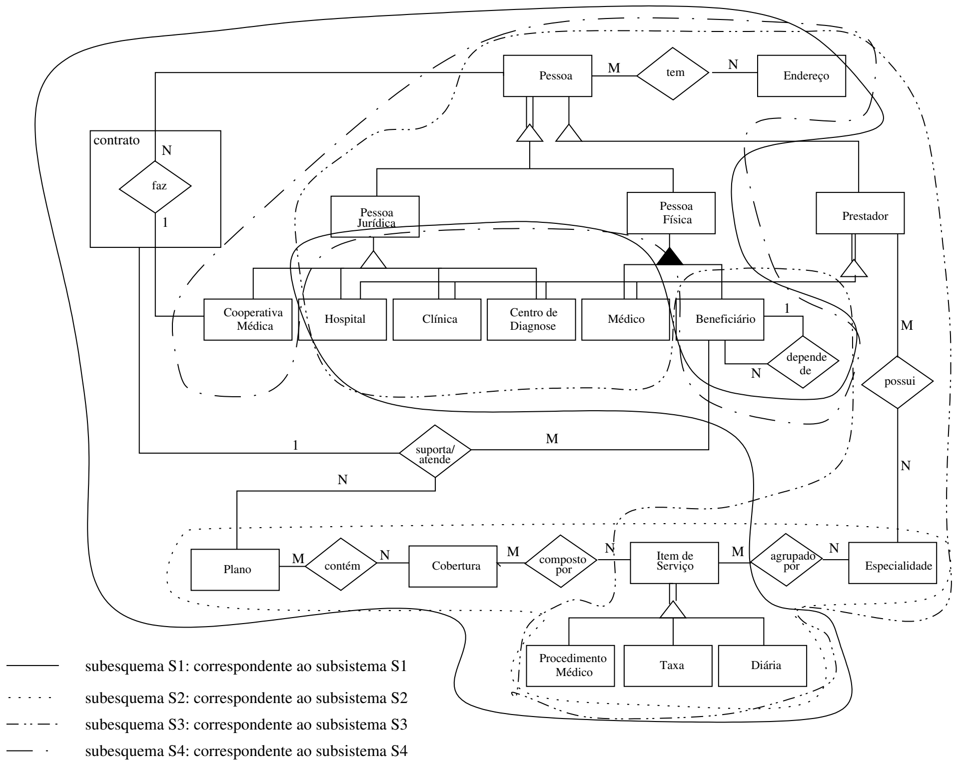


Figura 7.3: Esquema conceitual global de dados, mostrando os subesquemas

o que caracteriza porções do esquema global compartilhadas por subsistemas. Deve-se, então, classificar cada elemento do esquema global de dados quanto ao compartilhamento como mostrado na Tabela 7.1.

Etapa 3 - Definição dos módulos da base de dados: os módulos da base de dados são definidos a partir da análise da Tabela 7.1. O primeiro passo é agrupar os elementos da Tabela 7.1 de acordo com os subsistemas que os mantêm (subsistemas que executam operações de escrita), obtendo-se:

$G_{S1} = \{Contrato, Faz, Suporta/Atende\}$
$G_{S2} = \{$*Plano, Contém, Cobertura, Composto por, Item de Serviço, Procedimento Médico, Taxa, Diária, Agrupado por, Especialidade*$\}$
$G_{S3} = \{$*Pessoa, Pessoa Jurídica, Pessoa Física, Prestador, Hospital, Clínica, Centro de Diagnose, Médico, Possui, Endereço, Tem*$\}$
$G_{S4} = \{$*Pessoa, Pessoa Jurídica, Pessoa Física, Cooperativa Médica, Beneficiário, Depende de, Endereço, Tem*$\}$

O próximo passo é fazer as interseções entre os grupos, obtendo-se:

$G_{S1} \cap G_{S2} = \emptyset$
$G_{S1} \cap G_{S3} = \emptyset$
$G_{S1} \cap G_{S4} = \emptyset$
$G_{S2} \cap G_{S3} = \emptyset$
$G_{S2} \cap G_{S4} = \emptyset$
$G_{S3} \cap G_{S4} = \{$*Pessoa, Pessoa Jurídica, Pessoa Física, Endereço, Tem*$\}$
Os resultados são:

- G_{S1} e G_{S2} têm interseção vazia com todos os outros grupos e, portanto, originam diretamente os módulos M_1 e M_2, respectivamente;

- G_{S3} e G_{S4} têm interseção entre si e a maioria de seus elementos pertencem a uma hierarquia de generalização. Portanto, GS3ÈGS4 origina o módulo M_3.

Assim, definem-se os módulos M_1, M_2 e M_3 como mostrado na Tabela 7.3.

Etapa 4 - Definição da interface dos módulos da base de dados: nas Figuras 7.4, 7.5 e 7.6, encontram-se o esquema conceitual de dados de cada um dos módulos M_1, M_2 e M_3, respectivamente. Os conjuntos de entidades que aparecem pontilhados no esquema de dados de um módulo não

Tabela 7.1: Tipo de compartilhamento de cada elemento do esquema conceitual global de dados:

Elemento	Pertence ao(s) subesquema(s)				Tipo de Operação L-leitura/E-escrita				T*
	E_1	E_2	E_3	E_4	S_1	S_2	S_3	S_4	
Contrato	x				L/E				1
Faz	x				L/E				1
Suporta/Atende	x				L/E				1
Plano	x	x			L	L/E			2
Contém	x	x			L	L/E			2
Cobertura	x	x			L	L/E			2
Composto por	x	x			L	L/E			2
Item de serviço	x	x	x		L	L/E	L		2
Procedimento médico	x	x	x		L	L/E	L		2
Taxa	x	x	x		L	L/E	L		2
Diária	x	x	x		L	L/E	L		2
Agrupado por		x	x			L/E	L		2
Especialidade		x	x			L/E	L		2
Pessoa	x		x	x	L		L/E	L/E	3
Pessoa jurídica	x		x	x	L		L/E	L/E	3
Pessoa física	x		x	x	L		L/E	L/E	3
Prestado			x				L/E		1
Cooperativa médica	x			x	L			L/E	2
Hospital			x				L/E		1
Clínica			x				L/E		1
Centro de Diagnose			x				L/E		1
Médico			x				L/E		1
Beneficiário	x			x	L			L/E	2
Depende de	x			x	L			L/E	2
Possui			x				L/E		1
Endereço	x		x	x	L		L/E	L/E	3
Tem	x		x	x	L		L/E	L/E	3

T* Tipos de Compartilhamento
1 não compartilhado
2 unidirecional
3 multidirecional

existem realmente no módulo, apenas representam a existência de relacionamentos com esses conjuntos de entidades cujos dados estão armazenados em outro módulo. De acordo com a seção 5.3, esses relacionamentos são inter-relacionamentos de módulos, indicando a necessidade de manutenção de integridade referencial entre módulos no processo de integração dos módulos da base de dados (seção 6.3).

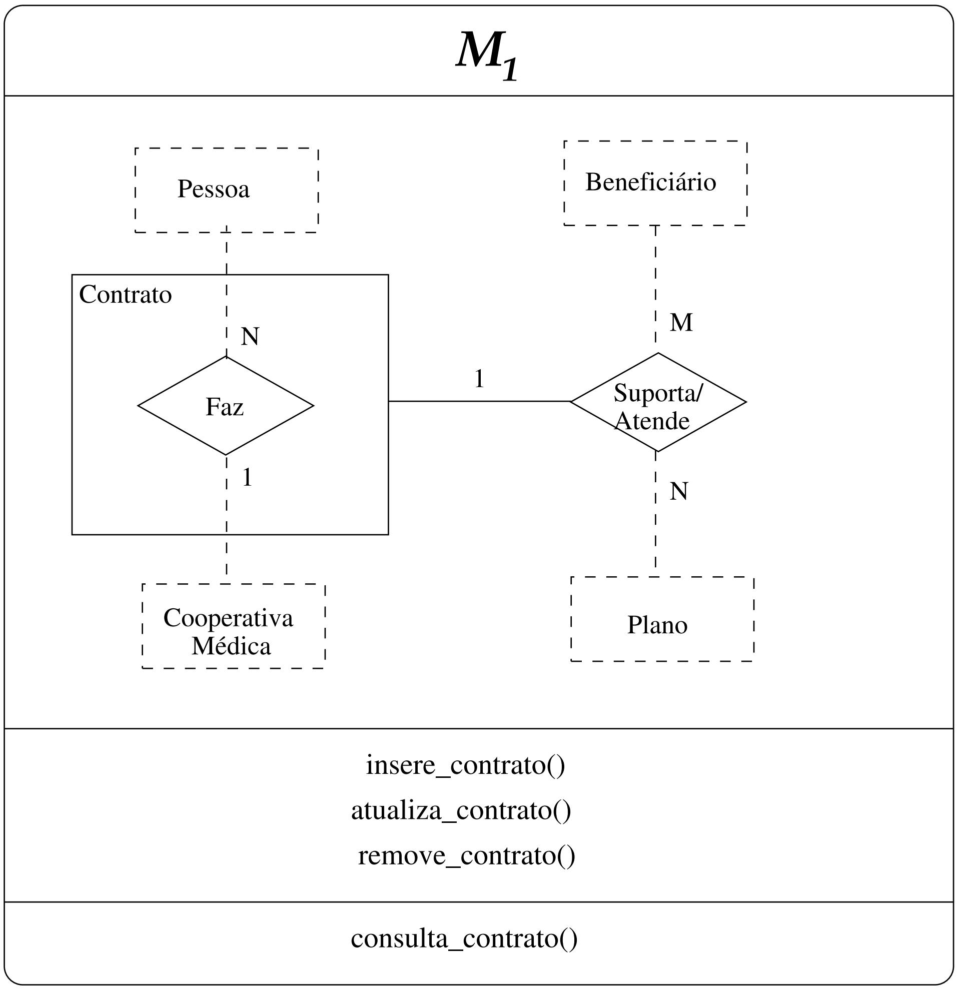


Figura 7.4: Esquema de dados do módulo M_1

Tabela 7.2: Definição dos módulos da base de dados

Módulo	Elemento	É mantido por			
		S_1	S_2	S_3	S_4
M_1	Contrato	x			
	Faz	x			
	Suporta/Atende	x			
M_2	Plano		x		
	Contém		x		
	Cobertura		x		
	Composto por		x		
	Item de serviço		x		
	Procedimento médico		x		
	Taxa		x		
	Diária		x		
	Agrupado por		x		
	Especialidade		x		
M_3	Pessoa			x	x
	Pessoa física			x	x
	Pessoa jurídica			x	x
	Prestador			x	
	Cooperativa médica				x
	Hospital			x	
	Clínica			x	
	Centro de Diagnose			x	
	Médico			x	
	Beneficiário				x
	Dependente de				x
	Possui			x	
	Endereço			x	
	Tem			x	x

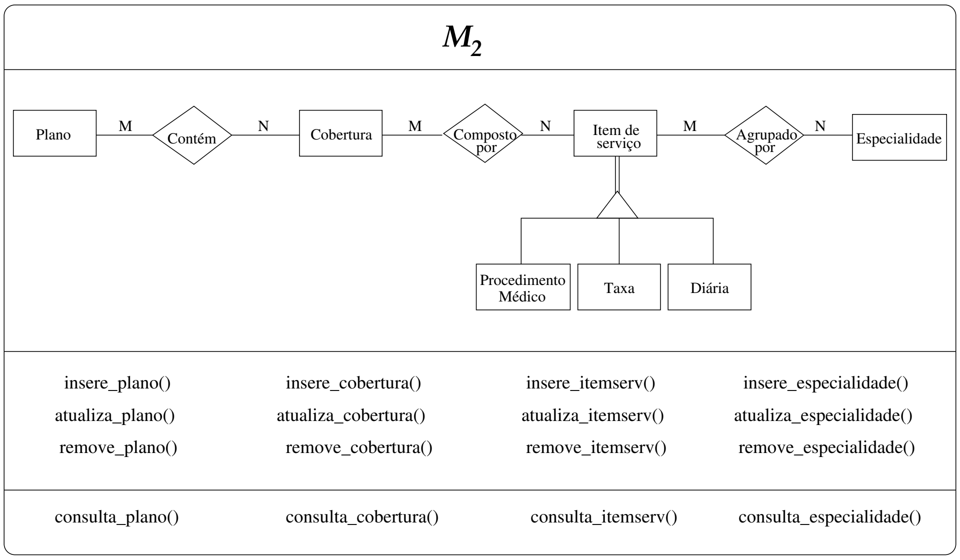


Figura 7.5: Esquema de dados do módulo M_2

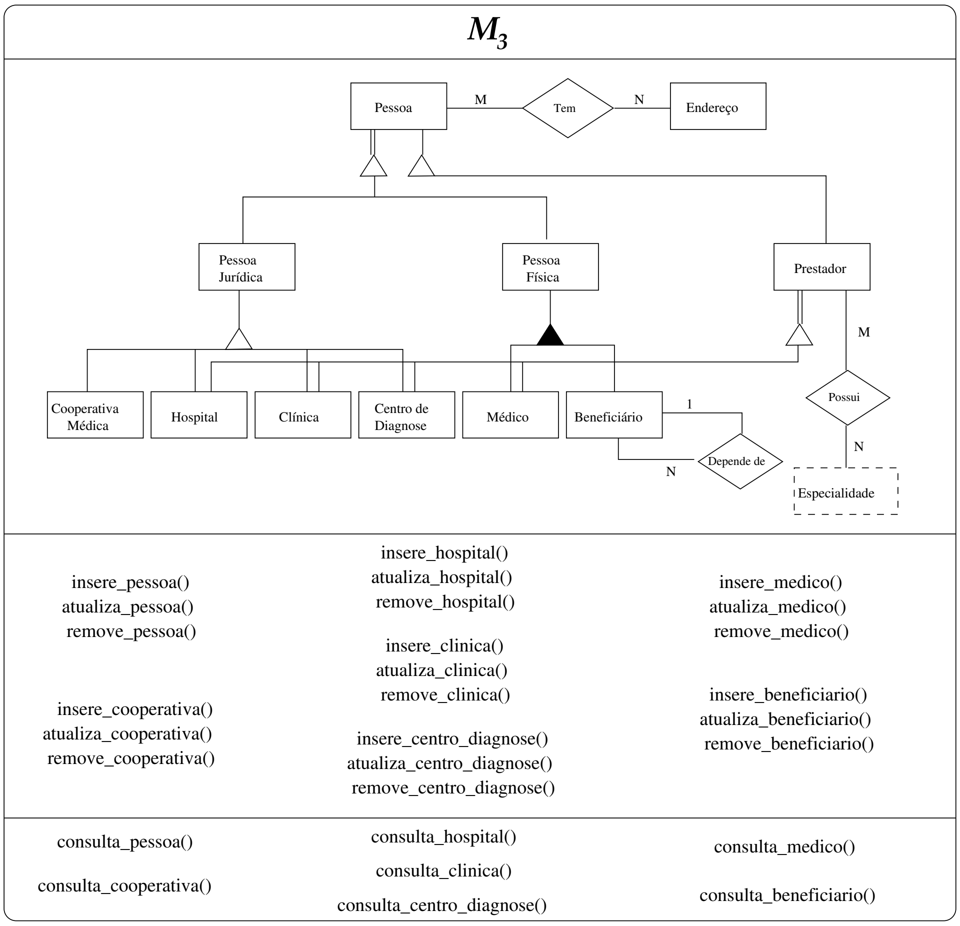


Figura 7.6: Esquema de dados do módulo M_3

Capítulo 8

Bancos de Dados com Dados Móveis

8.1 Introdução

As plataformas que suportam grupos de trabalhos e as redes locais têm atendido com eficiência as várias necessidades de suporte aos usuários, em particular ao desenvolvimento de projetos. Nesses ambientes um importante uso das estações de trabalho são as aplicações que utilizam os dados locais (base de dados local) e eventualmente se conectam ao sistema servidor para completar as informações não disponíveis no local em questão. Este modo de operação das plataformas, aliada à necessidade de uma maior capacidade semântica de representação de dados apontam para a necessidade de novas abordagens para distribuição de dados.

Alguns trabalhos como [Kim91a] apresentam especificações de necessidades e soluções para a distribuição de dados em base de dados orientadas a objetos. Esses trabalhos ressaltam a necessidade de uma arquitetura de base de dados na qual exista um determinado número de base de dados privadas (locais) para atendimento do trabalho local e bases dados remotas públicas, compartilhadas pelas bases de dados locais.

De maneira geral, uma base de dados privada(local) pode ser criada de duas maneiras. A primeira através da criação de dados dos próprios usuários locais. A outra pela cópia, através de processos de separação, dos objetos pertencentes a bases de dados públicas disponíveis nos servidores. Quando os objetos são criados na base de dados local, estes não estão necessariamente disponíveis a outras bases pertencentes a rede. Para a replicação ou particionamento de objetos utiliza-se o conceito de objeto complexo [LP83].

Portanto a unidade de cópia pode ser um simples objeto ou então uma coleção de objetos.

Após a fase de separação, as alterações ocorridas nos objetos são repassadas para a base pública, se durante tais alterações, os objetos na base pública permanecerem bloqueados. Os objetos posteriormente podem retornar a base pública ou serem copiados. Em síntese tal arquitetura de bases de dados permite a migração de objetos. Para isso são definidos tipos de autorização sobre os objetos, tipos de usuários, processos de separação e integração de dados.

As soluções apresentadas em [Kim91a] embora atendam às necessidades caracterizadas na introdução desse capítulo, estas não abordam os problemas decorrentes da composição de objetos para compartilhamento de dados e nem dos relacionamentos entre os objetos compostos. A fase de separação está limitada a um conjunto muito reduzido de tipos de utilizações, tais como: pública e privada, que os objetos sofrerão. As próximas seções apresentam alternativas para aumentar o significado semântico do modelo de autorização e o processo de separação e integração de dados apoiando-se na necessidade de mobilidade de dados conforme destaca o título deste capítulo.

8.2 Modelo de autorização sobre objetos

O mecanismo de autorização é um dos importantes controles de sistemas de bases de dados convencionais. Os modelos de autorização que suportam as bases de dados existentes convencionais estão intimamente ligados aos modelos relacional, hierárquico ou de redes. Estes modelos assumem que a unidade de autorização uma relação (tabela), ou um atributo da relação (campo). Isto é insuficiente quando necessita-se controlar as abstrações dos conceitos orientados a objetos, tais como hierarquia de classes, métodos e a composição de objetos.

O modelo de autorização proposto por Rabitti [FRK88], tem como pressuposto o conceito de autorização implícita já apresentada e formalizada por Fernandez [EBFL75], para os modelos de dados convencionais. A concepção convencional de autorização implícita é baseada em deduções e combinações dos poderes dos usuários para agirem sobre uma determinada base de dados, a partir das definições iniciais de tipos de usuários e unidades de base ou sub-bases de dados acessíveis. A proposta de Rabitti extende o conceito de autorização implícita para os conceitos de modelagem de dados orientada a objetos introduzindo três níveis para autorização: usuários (um usuário ou grupo de usuários); tipo de autorização (escrita, leitura, criação) e objetos (objetos simples, grupos de objetos ou bases de dados). Dessa maneira o conceito de autorização é analisado em três dimensões: autorização para usuários, tipos de autorizações e autorizações para objetos. Naturalmente, como essa extensão pressupõe uma base de dados com um poder semântico maior, a autorização implícita torna-se mais complexa.

Para controlar de maneira mais precisa as autorizações implícitas decorrentes da hierarquia de composição de objetos em uma base de dados orientadas a objetos, Won Kim [Kim91b] apresenta os mecanismos de autorização. Entre outros aspectos importantes, pode-se destacar a substituição do conceito de tipos de autorização pelos conceitos de autorização forte e fraca e autorização positiva e negativa. Os principais conceitos do modelo de autorização para a base de dados e seus respectivos formalismos, proposto por Won Kim, são apresentados a seguir.

8.2.1 Definição dos Conceitos para o Modelo de Autorização

Autorização Explícita

A autorização explícita é definida como uma tupla, autorização $< u, o, a >$ onde:

$u \in U$, onde $U = \{U_1, U_2, \ldots, U_n\}$ é a coleção de usuários do sistema;

$o \in O$, onde $O = \{O_1, O_2, \ldots, O_n\}$ é a coleção de objetos do sistema;

$a \in A$, onde $A = \{A_1, A_2, \ldots, A_n\}$ é a coleção de tipos de autorização.

A função $\mathcal{F}$ é definida para determinar se a autorização $< u, o, a >$ é verdadeira ou falsa:

$$\mathcal{F} : U \times O \times A \Rightarrow \{True, False\}$$

Dada uma determinada tupla $< u, o, a >$, se $\mathcal{F}(u, o, a) = True$ então:
o usuário **u** tem autorização do tipo **a** sobre o objeto **o**

Autorização Implícita

O conceito de autorização implícita é fundamental no modelo de autorização para uma base de dados orientadas a objetos. Na medida em que a autorização ¡u,o,a¿ está explicitamente definida em vários nós da hierarquia de classe, então pode-se utilizar o conceito de autorização implícita para controlar as operações em nós dos níveis hierárquicos inferiores. Se em um determinado nó α do nível hierárquivo for definido $\mathcal{F}(u_1, o_1, a_1)$, em outro nó β for definido $\mathcal{F}(u_2, o_2, a_2)$ e ainda, se α e β mantém um vínculo entre si na hierarquia de composição da base de dados então, pode-se baseado em regras derivação controlar a propagação de autorizações não conflitantes nos mais variados níveis da hierarquia. A Figura 8.1 ilustra o conceito de autorização explícita e as suas respectivas autorizações implícitas decorrentes.

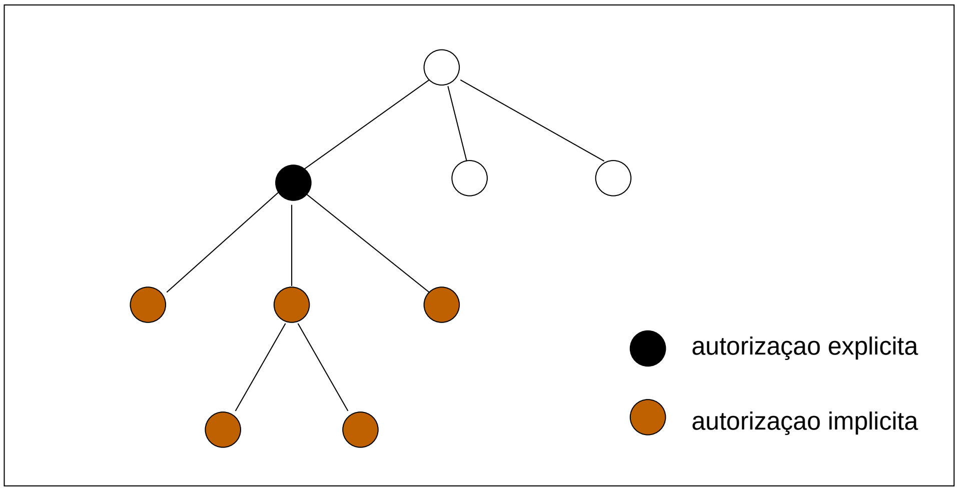


Figura 8.1: Tipos de autorização explícita e implícita

Autorização Positiva e Negativa

O mecanismo de autorização assume um valor positivo quando eventualmente um usuário pode acessar um determinado objeto. O conceito de autorização negativa é complementar a autorização positiva. Quando um determinado usuário u_1 tem acesso negado para um determinado objeto o_1, pelo fato de outro usuário u_2 ter autorização sobre o objeto o_1, então diz-se que o usuário u_1 tem autorização negativa sobre o objeto o_1.

Autorização Forte e Fraca

O conceito de autorização forte significa que num contexto de uma rede hierárquica, nenhuma outra autorização pode sobrepor uma autorização forte já definida e suas autorizações implícitas decorrentes. A autorização fraca, ao contrário da forte, pode ser substituída por uma outra autorização fraca ou uma autorização forte. Isto permite afirmar que uma autorização implícita ou explícita fraca pode ser substituída por uma autorização explícita forte.

A combinação das autroizações explícitas/implícitas, positiva/negativa e forte/fraca permite a obtenção de uma semântica de acesso no contexto de uma rede hierárquica. Cabe observar que quanto mais autorizações fracas houverem na hierarquia, maior o poder de alteração do sistema do controle de autorização. Quanto maior o número de autorizações fortes, menor será o poder de alteração do sistema de controle de autorização, pois uma autorização forte, não pode ser substituída nem por outra autorização forte. Para sobrepor uma autorização forte é necessário primeiro removê-la. A Figura 8.2 ilustra uma hierarquia com as combinações das autorizações sendo utilizadas.

8.2.2 Regras de Derivação

Para cada elemento a ser gerenciado pela autorização (usuário, objeto e tipo de autorização) existe uma semântica de derivação que pode ser traduzida num grafo acíclico direcionado, de acordo com o poder de atuação de cada nó. No caso de usuários, um grafo pode ser construído ligando-se um nó origem, no qual representa-se o poder de atuação de um tipo de usuário (papel) com outro nó destino, no qual o tipo de usuário (papel) possui um poder de atuação hierarquicamente inferior. A técnica de utilização de grafos acíclicos também é devidamente extendida para a hierarquia de objetos de uma base de dados (super-classes, classes, sub-classes, atributos) e para hierarquia do tipo de autorização (leitura, escrita e criação). Este modelo permite a criação dos nós que comporão cada elemento do controle de autorização: usuários, objetos e tipos de autorização estabelecendo uma hierarquia de poder de acesso do usuário. Apoiando-se nestas considerações, é possível mostrar algumas regras de derivação imediatas para o grafo acíclico. As figuras a,b,c da Figura 8.3 ilustram os grafos acíclicos para usuários, tipos de autorizações e objetos respectivamente.

Definição 1: A notação $u_i > u_j$ onde u_i pertence a U e u_j pertence a U significa que existe uma ligação direcionada de u_i para u_j no grafo acíclico, de modo que se $u_i \geq u_j$ implica que $u_i = u_j$ ou $u_i > u_j$ ou então existe $u_1, \ldots, u_n$ em U de modo que $u_i > u_1 > \ldots > u_n > u_j$.

Regra 1: $\forall o \in O$ e $a_n \in A$, se $u_i \geq u_j$, então

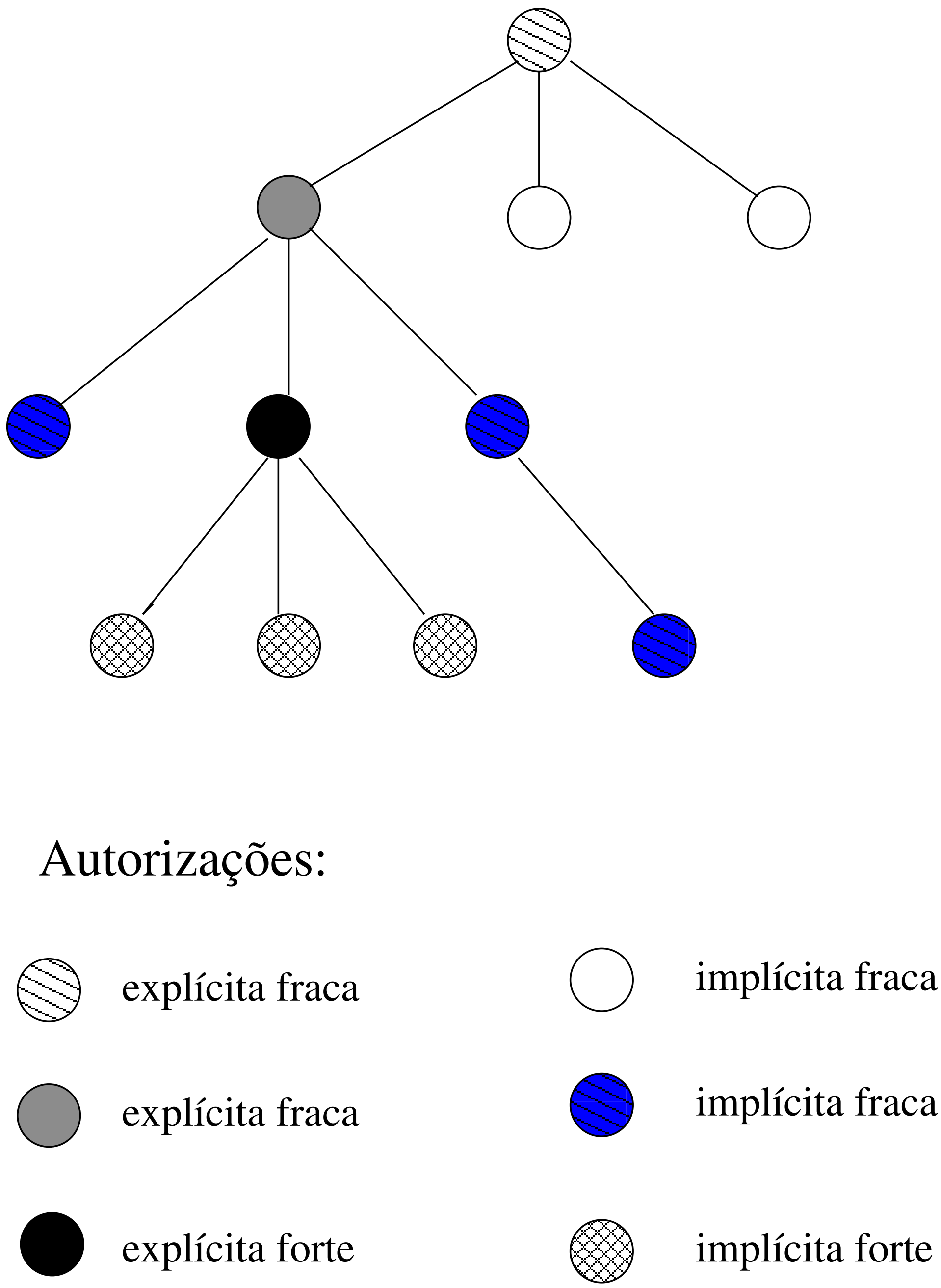


Figura 8.2: Tipos de autorizações combinadas

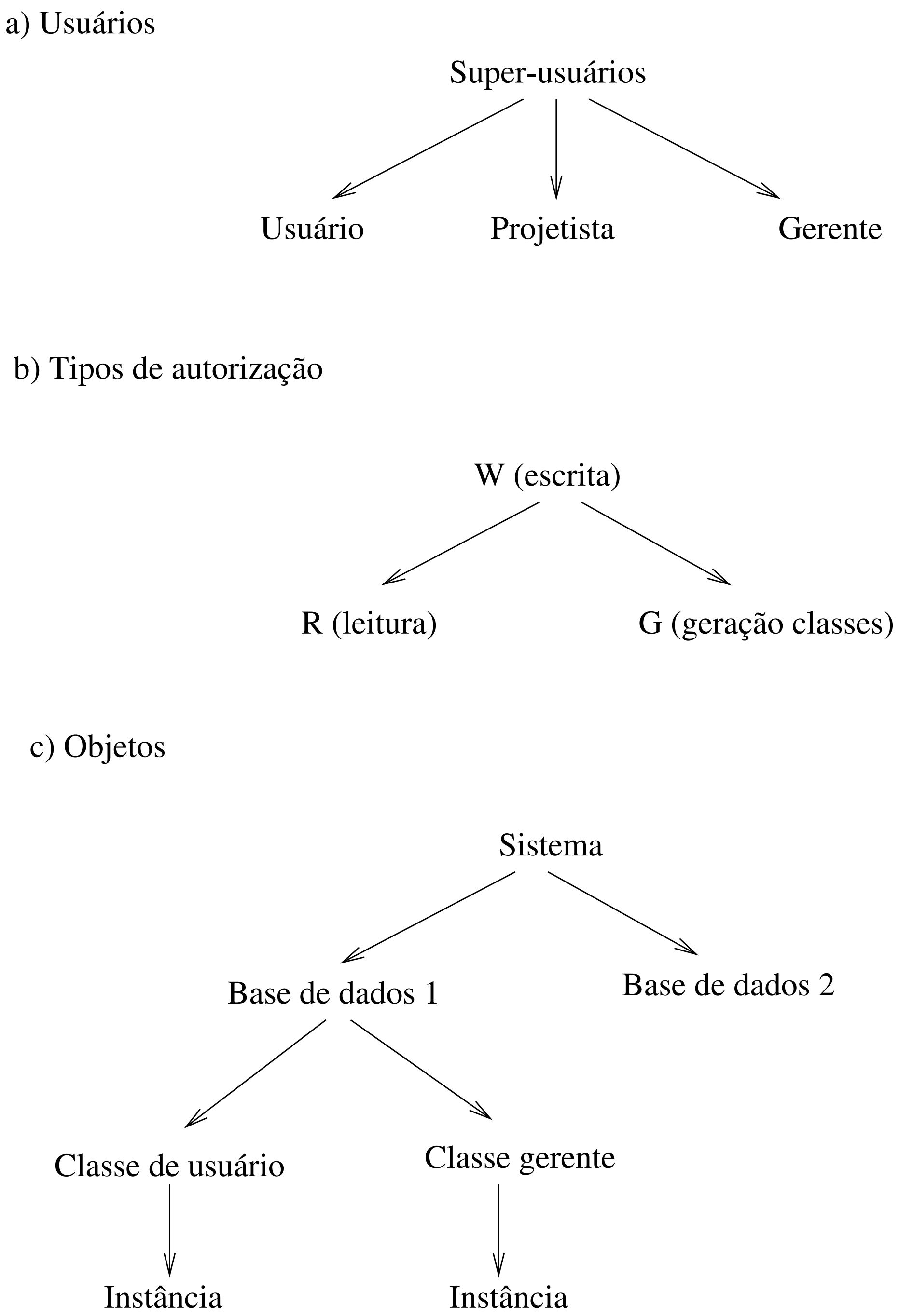


Figura 8.3: Grafos acíclicos de usuários, tipos de autorizações e objetos

$$(u_j, o, a_n) \rightarrow (u_j, o, a_n).$$

Isto significa que o poder de acesso (papel) de um usuário u_j, sobre um objeto o, está contido no poder de acesso do usuário u_i sobre o objeto o.

Definição 2: A notação $a_i > a_j$ onde a_i pertence a A e a_j pertence a A significa que existe uma ligação direcionada de a_i para a_j no grafo acíclico, de modo que $a_i > a_1 > \ldots > a_n > a_j$.

Regra 2: Se para qualquer u pertencente a U e o pertencente a O, se $a_i > a_j$, então
$$(u, o, a_i) \rightarrow (u, o, a_j).$$

Isto significa que o poder de acesso do usuário u sobre o objeto o é maior na autorização a_i comparada à autorização a_j.

Analogamente às definições 1 e 2, pode-se apresentar a definição 3 e a regra 3 para objetos pertencentes a um grafo acíclico. Para esta definição, é necessário incluir o conceito de propagação ascendente e descendente das autorizações. No caso da autorização de leitura sobre uma classe de objetos, a qual inclui autorização de leitura das propriedades e instâncias da classe, significa que a autorização de leitura também estará permitida para todas as instâncias da classe em questão. Neste caso, tem-se a autorização com propagação descendente ("A. down"). Inversamente, se for dada autorização de leitura para instâncias de objetos, ler a instância implica em ler a definição do objeto. Neste caso, tem-se a autorização com propagação ascendente ("A. up"). Dessa maneira, tem-se a coleção de tipos de autorização $A = A.down \bigcup A.up$

Definição 3: A notação $o_i > o_j$, onde $o_i, o_j \in O$, significa que existe uma ligação dirigida de o_i para o_j no grafo acíclico, de modo que, se $o_i > o_j$, implica que $o_i = o_j$, ou $o_i > o_j$, ou então existe $o_1, \ldots, o_n$ em O de modo que $o_i > o_1 > \ldots > o_n > o_j$.

Regra 3:

a) Se, para qualquer u pertencente a U e a_n pertencente a $A.down$, se $o_i > o_j$, e ainda as autorizações a_n sobre os objetos o_i e o_j estão definidas, então:

$$(u, o_i, a_n) \rightarrow (u, o_j, a_n).$$

b) Se, para qualquer u pertencente a U e a_n pertencente a $A.up$, se $o_i \geq o_j$, e ainda as autorizações a_n sobre os objetos o_i e o_j estão definidas, então:

$$(u, o_j, a_n) \rightarrow (u, o_i, a_n).$$

8.3 Modelo de Compartilhamento de dados

Em bases de dados não convencionais, tais como aquelas que atendem às aplicações de computação móvel e aos ambientes de desenvolvimento de projetos, as necessidades de distribuição tem características distintas das de uma base de dados convencional. Embora possam haver situações em que o sistema deva incorporar o tratamento de concorrência dos dados como apresentado nos capítulos 2 e 3, o desenvolvimento e utilização desse recurso de concorrência de dados passa a ser apenas mais um, face a outras necessidades de distribuição para os novos ambientes citados. Nesses ambientes, a distribuição dos dados envolve, além do tratamento de um novo paradigma para a concorrência dos dados, o controle de versão e partição de dados nas mais variadas formas de cópias.

As principais necessidades de distribuição de uma base de dados em ambientes de computação móvel e de desenvolvimentos de projetos são:

1. atender às transações longas;
2. permitir que parte da base de dados seja de uso exclusivo de cada usuário e outras partes compartilhadas por outros usuários;
3. permitir um grau controlado de conflitos de transações que são postergadas;
4. permitir que o usuário faça transações de forma isolada nos dados sem comunicação com o servidor de dados;
5. permitir o processo de integração de dados aos servidores de dados após o fim da utilização dos mesmos pelos usuários.
6. permitir a utilização de estratégias e algoritmos diferenciados para o controle consistente de réplica e particionamento de dados.

Dado que as necessidades de distribuição nesses ambientes são mais amplas que as tradicionamente atendidas em ambientes relacionais, será utilizado aqui o termo *"compartilhamento"* de dados , considerando-se que a distribuição de dados é um dos casos particulares das formas de compartilhamento de dados.

Segue nas próximas seções a descrição de um modelo de compartilhamento de dados capaz de estabelecer fragmentos em uma base de dados, formas de vínculo para a evolução das partes, grau de compartilhamento e formas para possíveis integrações desses fragmentos, de acordo com as características das aplicações.

8.3.1 Objeto Fragmento para Compartilhamento de Dados

Para tratar a base de dados como uma "massa" de dados que pode ser dividida é necessário também um outro conceito que defina como agrupar os dados envolvendo-os em unidades referenciáveis, que são denominados aqui de *objeto fragmento*. Os *objetos fragmentos* são os elementos que o sistema de controle de réplica e particionamento pode utilizar para armazenar os tipos vínculos e as estratégias de integração de dados entre as várias cópias ou partições num ambiente compartilhado de dados.

No contexto de compartilhamento de dados, agrupar significa envolver ou colecionar um conjunto de dados. Assim cada *objeto fragmento* é capaz de encapsular um conjunto de dados. Esse conjunto de dados pode ser apenas uma instância de um determinado dado, uma coleção de tuplas ou colunas de uma relação(tabela), ou um objeto complexo no paradigma de orientação a objetos.

Independentemente do significado semântico do objeto fragmento, ele é o primeiro elemento a ser definido no processo de compartilhamento de dados. Sempre que ocorre alguma divisão da base de dados um objeto fragmento é criado. Existe assim um processo de "extração" ou *separação* de dados da base de dados que chamamos nesse texto de *base de dados origem*. Esses dados são encapsulados pelo *objeto fragmento*. Os dados encapsulados por tal objeto poderão ser replicados ou alocados sem réplica em vários outros nós da rede ou de computadores móveis.

Para melhor caracterizar tal fase de separação é preciso atribuir mais informações ao objeto fragmento. Os dados *separados* da base origem devem *mover-se*, replicados ou não, para outras aplicações em outros ambientes. O local destino para o qual tal movimentação ocorreu, denominamos aqui de *base destino*. Cabe observar que os dados movidos para base destino não necessariamente estão sob o controle de um gerenciador de banco de dados, podendo ter variações comuns de arquivos de dados.

De modo a delimitar e definir um conjunto de permissões associadas a cada operação de escrita e leitura dos dados que se moveram para a base destino em relação a base origem, estabelecemos o conceito de *tipo de vínculo* entre os dados separados na base origem e os movidos para a base destino.

Considerando as várias possibilidades de vínculos entre as replicações ou partições de dados entre a base de origem e destino, no que se refere ao poder das operações de escrita e/ou leitura sobre os dados encapsulados pelo objeto fragmento temos os seguintes tipos de vínculos mais significativos:

1. *apenas leitura*, as operações permitidas restringem-se apenas a leitura tanto na base origem como na base destino;

2. *flagrante*, na base origem apenas as operações de leitura são permitidas enquanto que na base destino são possíveis as operações de leitura e escrita;

3. *isolado*, apenas a base origem possui permissão de escrita e leitura, restando a base destino apenas as operações de leitura;

4. *mutuamente exclusivo*, operações de escrita e leitura são permitidas em ambas as bases, entretanto apenas a base original ou a base destino será tomada como referência para o processo de re-integração de dados, que será visto com mais detalhe na próxima seção. Nesse tipo de vínculo, apenas uma base será considerada sendo a outra descartada pelo usuário no instante da re-integração.

5. *simultâneo*, tanto a base origem como a base destino podem ter as operações de escrita e leitura autorizadas. Entretanto qualquer alteração em uma das bases de dados esta deve imediatamente ser repassada para outras bases. Nesse tipo de vínculo toda a teoria de controle de concorrência e os algoritmos vistos no ambiente distribuído nos Capítulos 2 e 3 são utilizados.

6. *independente*, tanto a base origem como a destino(replicada ou não) podem ter operações de escrita e leitura de forma assíncrona. Os conflitos que eventuamente ocorram poderão ser minimizados e resolvidos de acordo com o significado de cada processo de re-integração. Na seção 8.4 será abordado uma estratégia de integração baseado no conceito de multiversão de dados.

Não iremos entrar em detalhes de como controlar de forma heterogênea o acesso aos dados encapsulados pelo *objeto fragmento*, pois não é objetivo desse texto. Assumiremos nesse texto, que todos os dados encapsulados pelo objeto fragmento tem o mesmo tipo de operações de escrita e leitura definindo uma granularidade fixa para o controle de acesso aos dados em questão. Entretanto é possível variar a granularidade do controle de acesso aos itens de dados do *objeto fragmento* através da utilização dos conceitos vistos na seção 8.2. Detalhes podem ser encontrados em [FT96] e [TFB97].

Para cada tipo de vínculo escolhido, deve-se associar um processo para a re-integração de dados. Esses processos normalmente estão associados a estrutura de *log* ou marcas de tempo para solucionar as possíveis transações conflitantes sobre os dados que se movimentaram para outros locais.

8.3.2 A Operação de Compartilhamento de Dados

A Operação de Compartilhamento é constituída de três fases: *separação, evolução e re-integração.* A fase de separação consiste na extração de dados de uma base de dados como visto na seção anterior.

Na fase de evolução, as operações de escrita e leitura serão executadas normalmente nos dados encapsulados pelo *objeto fragmento*, respeitando as restrições estabelecidas no *tipo de vínculo.* Esta fase se prolongará até que o usuário decida a realização da re-integração dos dados. Em alguns casos extremos pode-se até anular o vínculo de compartilhamento tornando a base de dados autônoma.

Os processos pertencentes a fase de re-integração dependem dos dados gerados durante a fase de separação, que estão localizados tanto na base origem como destino. Para cada tipo de vínculo estabelecido e armazenado no *objeto fragmento* entre a base origem e destino, deverá haver um *método* específico que seja capaz de integrar a base destino com a base origem, levando-se em consideração as alterações ocorridas em cada base.

Assim é definido para cada tipo de vínculo um algoritmo para a re-integração das bases de dados, implementados através dos métodos disponíveis no *objeto fragmento.* Existe uma correspondência direta entre o tipo de vínculo escolhido e o método responsável pela integração.

Os quatro primeiros tipos de vínculos definidos na subseção 8.3.1 tem métodos de integração que são facilmente implementados considerando que os conflitos das transações não existem. As possibilidades de conflitos foram eliminados pela restrição das operações de escrita e leitura em cada base origem e destino.

No quinto tipo de vínculo(simultâneo), a re-integração se restringe a eliminação do objeto fragmento e suas réplicas ou partições. O método para manter a consistência dos dados é síncrono e utiliza os algoritmos mencionados nos Capítulos 2 e 3. Os conflitos entre as transações foram todos solucionados pelos escalonadores de transações intrínsecos ao método.

No último tipo de vínculo(independente), há necessidade de reconciliar as transações sobre os dados na medida em que o controle de concorrência é assíncrono. Na fase de re-integração pode-se considerar algumas reconciliações simples como média aritmética entre as réplicas, o maior ou menor valor da instância da réplica. Entretanto a complexidade aumenta quando queremos reconciliar os valores de cada réplica e armazenar os valores parciais que traduzem fielmente as transações locais que operaram em cada uma das réplicas.

As alternativas para a solução se baseiam na reconciliação de arquivos de *logs* de transações de cada uma das réplicas ou no controle de multiversão de dados.

8.4 Formas de Re-Integração Dados Independentes

Recentemente muitas pesquisas tem direcionado a solução da re-integração de fragmentos de dados a esquemas de replicação otimistas e base de dados móveis com processos de reconcialiação. Alguns trabalhos iniciais podem ser encontrados em [AK93] e [Dav84]. Mais recentemente Gray em [Gra96], apresenta uma arquitetura de sistema e um modelo de replicação para base de dados móveis. A base de dados é tratada como uma coleção de réplicas de objetos com cópias primárias controladas através de nós coordenadores conhecidos como objetos mestres. O modelo em questão utiliza duas camadas para o esquema de replicação. Uma camada para os nós móveis (desconectados) e outra para os nós da base de dados (conectados). Todas as transações operam primeiramente sobre a primeira camada, controladas pelo objeto mestre e depois dependendo de alguns critérios de aceitação, são repassados para a base de dados quando os nós móveis se reconectam.

Em Bayou [JGS94], as réplicas são cópias locais *completas* do repositório de dados. Este pressuposto requer aplicações específicas para detecção e resolução de conflitos para a atualização de dados. As atualizações são reconciliadas quando dois repositórios de dados são conectados. A reconciliação é feita através da reprodução e junção de todas as atualizações realizadas em cada servidor local, através da utilização de arquivos de *logs* locais. Tal reconciliação gera um conjunto de operações, incluindo a detecção e resolução de conflitos para sincronizar as dados de cada repositório. O processo de reconciliação de todas as cópias é feito duas a duas. Note que o repositório de dados nessa arquitetura não é necessariamente uma base de dados.

Uma outra abordagem encontrada em [WC95] para o problema de reconciliação é aliviar o número total de transações das aplicações com a fragmentação de objetos. Nessa abordagem, o projetista do fragmento tira proveito do conhecimento semântico da aplicação para a quebra de grandes e complexos objetos para pequenos fragmentos. O computador móvel ou o nó desconectado da rede pode processar as transações de modo independente.

Alguns dos trabalhos citados acima, principalemente as abordagens otimistas, não estão preocupados com o problema da resolução de conflitos. Por outro lado, as propostas de reconciliação baseadas em arquivos de *log* ou a de fragmentação de objetos, requerem um conhecimento especializado do sistema de informação, para todas as transações em especial para o modelo de controle de concorrência.

8.4.1 Versão de Instâncias da Base de Dados

O modelo de reconcialiação baseado em instâncias proposto em [PB99], considera a base de dados como uma coleção de itens de dados, tuplas ou objetos, com suas respectivas versões. O item de dado está vinculado a um determinado universo X. A base de dados em qualquer determinado instante, existe como um estado particular. Tal estado reflete as atualizações realizadas para todas as confirmações das transações para um determinado instante. A cada transação confirmada sobre o item de dado é gerado uma marca de tempo. As marcas de tempo estabelecem a ordem das transações confirmadas. O número de versões de dados depende do número de marcas de tempo das transações confirmadas para cada item de dado. A cada transação confirmada, o conjunto de estados do banco de dados D expande para incluir as atualizações realizadas pelas transações. Detalhes desse mecanismo podem ser encontrado em [PB99].

O conceito de compartilhamento de dados visto na seção 8.3 adquire uma conotação diferente do modelo de distribuição convencional de dados. Os tipos de vínculos entre as bases origem e destino estabelecem uma semântica de compartihamento de dados que atende de forma mais completa ao ambiente de desenvolvimento de projetos e computação móvel. Os dados extraídos e encapsulados pelo *objeto fragmento* juntamente com métodos de re-integração permitem uma transparência de replicação e controle de concorrência, tanto dos métodos síncronos como assíncronos. Com a possibilidade de pré-estabelecer os métodos de re-integração dos dados, o modelo apresentado permite ainda uma flexilidade na escolha da estratégia de reconciliação de dados no modo independente. Os métodos podem ser especializados em algoritmos baseados em arquivos de *log* ou em multiversão de dados, de acordo com as características de cada aplicação.

Referências Bibliográficas

[AK93] R. Alonso and H. Korth. Database system issues in nomadic computing. In *Proceedings of the ACM SIGMOD*, pages 388–392, 1993.

[BBG+83] C. Beeri, P. A. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transactions. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 45–62, 1983.

[BF99] G. Busichia and J. E. Ferreira. Compartilhamento de modulos de bases de dados heterogeneas atraves de objetos integradores. In *Proceedings of XIV Simposio Brasileiro de Banco de Dados*, pages 395–410, 1999.

[BH73] P. Brinch-Hansen. *Operating System Principles.* Prentice-Hall, 1973.

[BHG87] P. A. Berstein, V. Hadzilacos, and N. Goodman. *Concurrency control and Recovery in Database Systems.* Addison Wesley, 1987.

[BSW79] P. Bernstein, D. Shipman, and W. Wong. Formal aspects of serializability theory in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[CM84] M. A. Casanova and A. V. Moura. *Principios de Sistemas de Gerencia de Banco de Dados Distribuidos.* IV Escola de Computacao - Sociedade Brasileira de Computacao, 1984.

[CP84] S. Ceri and G. Pelagatti. *Distributed Database: Principles and Systems.* MacGraw Hill, 1984.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database system. In *ACM Transactions on Database Systems*, pages 456–481, 1984.

[DHB97] Margaret H. Dunham, Abdelsalam Helal, and Santosh Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *ACM/Baltzer Journal on Special Topics in Mobile Networks and Applications (MONET)*, 2:149–162, 1997.

[EBFL75] R. C. Summers E. B. Fernandez and T. Lang. Definition and evaluation of access rules in data management systems. In *Proceedings of the International Conference on Very Large Databases*, pages 129–148, 1975.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, 1976.

[EN00] R. Elmasri and S. B. Navathe. *Fundamental of Database Systems, 3.ed.* Addison-Wesley, 2000.

[FB99] J. E. Ferreira and G. Busichia. Database modularization design for the construction of flexible information system. In *Proceedings of IEEE International Database Engineering and Application Symposium*, pages 442–452, 1999.

[Fer96] J. E. Ferreira. *Compartilhamento de objetos compostos entre bases de dados orientadas a objetos.* PhD thesis, IF, Universidade de Sao Paulo, Campus de Sao Carlos, 1996. 127p, Tese doutorado, IF-USP S. Carlos.

[FLMW87] Alan Fekete, Nancy Linch, Michael Merritt, and William Weihl. Nested transactions and read/write locking. Technical report, Massachussets Institute of Technology, MIT/LCS/TM-324, 1987.

[FLMW88] Alan Fekete, Nancy Linch, Michael Merritt, and William Weihl. commutativity-based locking for nested transactions. Technical report, Massachussets Institute of Technology, 1988.

[FRK88] D. Woelk F. Rabitti and W. Kim. A model of authorization for object-oriented and semantic database. In *International Conference on Extending Database Technology*, pages 85–127, 1988.

[FT96] J. E. Ferreira and C. TrainaJr. Controle de compartilhamento e acesso em gbdoo baseado em composicao de objetos. In *Proceedings of 11o. Simposio Brasileiro de Banco de Dados*, pages 143–157, 1996.

[GLPT75] J. Gray, R. Lorie, G. Putzulo, and I. Traiger. Granularity of locks in shared databases. In *Proceeding of the 1str International Conference on Very Large Databases (VLDB)*, pages 428–451, 1975.

[Gra78] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, LNCS 60*, pages 393–481, 1978.

[Gra81] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference of Very Large Data Bases*, pages 144–154, 1981.

[Gra96] J. N. Gray. The dangers of replication and solution. In *Proceedings of ACM SIGMOD*, pages 173–182, 1996.

[Had83] V. Hadzilacos. An operational model for database system reliability. In *Proceedings of the 2nd ACM Symposiun on Principles of Database Systems (PODS)*, pages 244–256, 1983.

[Had88] V. Hadzilacos. A theory of reliability for database systems. *Journal of the ACM*, 35(1):121–145, 1988.

[Hol72] R. Holt. Deadlock properties in computer science. *ACM Computer Surveys*, 4(3):179–196, 1972.

[JGS94] P. E. O'Neil J. Gray, P. Helland and D. Shasha. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Aplications*, pages 2–7, 1994.

[KaF88] H. F. Korth and W. Kim amd F.Bancilhon. On long duration CAD transactions. *Information Sciences*, October 1988.

[Kim91a] W. Kim. A distributed object-oriented database system supporting shared and private databases. In *ACM Transactions on Informations Systems, V9*, pages 31–51, 1991.

[Kim91b] W. Kim. A model of authorization for next-generation database system. In *ACM Transactions on Informations Systems, V16*, pages 88–131, 1991.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases*, pages 95–106, Brisbane, Queensland, Australia, August 13-16 1990. Morgan Kaufmann. Also appearing as [KLS98].

[KLS98] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In Vijay Kumar and Meichun Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 15, pages 444–465. Prentice Hall, 1998.

[LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Proceedings of the Database for Engineering Aplications*, pages 115–121, 1983.

[LS76] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, Computer Science Laboratory, Xerox, Palo Alto Research Center, 1976.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachussets Institute of Technology, 1981. Tech Rep MIT/LCS/TR-260.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[Nav92] S. Navathe. Evolution of data modeling for databases. In *Communications of the ACM, vol 35, no.9*, pages 112–123, 1992.

[NCW]

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[Pap79] C. Papadimitriou. Serializability of concurrency database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[Pap86] C. Papadimitriou. *The Theory of Concurrency Control*. Computer Science Press, 1986.

[PB99] S. H. Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. In *Proceedings of the 15th Internation Conference on Data Engineering*, pages 582–589, 1999.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th VLDB Conference*, 1988.

[SCW87] B. Pernici S. Ceri and G. Wiederhold. Distributed database design methodologies. In *Proceedings of the IEEE, vol 75, no.5*, pages 533–546, 1987.

[Ske82] D. Skeen. Nonblocking commit protocols. In *Proceeding of the ACM SIGMOD Conference on Management of Data*, pages 133–147, 1982.

[SLR76] R. Stearns, P. Lewis, and D. Rosenkrantz. Concurrency control for database systems. In *Proceedings of the 17th Symposium on Foundations of Computer Science*, pages 19–32. IEEE, 1976.

[TFB97] C. TrainaJr, J. E. Ferreira, and M. Biajiz. Use of a semantically grained database system for distribution and control within design environments. In *Proceedings on Third Conference International EuroPAR'97*, page 11301135, 1997.

[WC95] G. Walborn and P. Chrysanthis. Supporting semantics-based transaction processing in mobile database systems. In *Proceedings of the 14th Symposium on Reliable Database Systems*, pages 81–94, 1995.