Marco Antonio Casanova
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Arnaldo Vieira Moura
Instituto de Computação
UNICAMP

Princípios de Sistemas de Gerência de Bancos de Dados Distribuídos

Edição revisada – distribuição restrita
©1999, Marco Antonio Casanova e Arnaldo Vieira Moura

PREFÁCIO

Um banco de dados é um repositório integrado e compartilhado dos dados operacionais de um empreendimento. Um sistema de gerência de bancos de dados é uma ferramenta destinada a isolar os programas de aplicação dos detalhes de armazenamento do banco, controlando o acesso ao banco e evitando inconsistências ou acessos indevidos, entre outros requisitos funcionais.

Com tal, nos últimos 25 anos, sistemas de gerência de bancos de dados têm representado um papel fundamental para o desenvolvimento de sistemas de informação. Estes sistemas adotaram nos primeiros anos uma arquitetura centralizada, no sentido de que o banco de dados reside em um único processador central, principalmente por razões de economia de escala e controle dos dados. Por outro lado, o avanço das técnicas e o barateamento dos serviços de comunicação de dados tornaram possível e atrativo migrar para sistemas de informação com uma arquitetura distribuída. Esta evolução levou naturalmente à criação de bancos de dados também distribuídos, onde a responsabilidade de armazenamento, atualização e controle dos dados é dividida entre vários locais. A comunidade destes bancos de dados locais deve ser interligada e coordenada por um sistema de gerência de bancos de dados distribuídos (SGBDD) que permita acesso a dados remotos de forma transparente e confiável, mas mantenha a autonomia dos bancos locais.

O desenvolvimento de SGBDDs de uso geral lançou um novo desafio em várias áreas, como processamento de consultas, gerência de transações e, em especial, controle de integridade e controle de concorrência. O propósito deste livro é reunir as técnicas mais relevantes resultantes de pesquisa nestas áreas, de forma coerente e acessível a profissionais e estudantes com interesse em software de base para bancos de dados ou sistemas distribuídos em geral.

ORGANIZAÇÃO DO LIVRO

A construção de um SGBDD de uso geral é uma tarefa não trivial, comparável à criação de um sistema operacional sofisticado. O Capítulo 1 abre a discussão introduzindo uma arquitetura genérica para estes sistemas, que é acompanhada de uma lista de requisitos funcionais para SGBDDs, e uma caracterização das principais interfaces oferecidas por um SGBDD aos usuários. Após este trabalho preliminar, é possivel enunciar claramente quais são as principais funções de um SGBDD. Este primeiro capítulo age, desta forma, como um resumo conciso e estruturado do livro, servindo de guia para todo o material que se segue.

O segundo capítulo entra em mais detalhes sobre o que são bancos de dados distribuídos (BDDs). Este capítulo cobre resumidamente os aspectos relativos à descrição, projeto, e administração de BDDs, indicando em que pontos diferem do caso centralizado.

Os Capítulos 3 a 5 discutem em detalhe o processamento de comandos da linguagem de manipulação de dados (LMD), o primeiro dos grandes problemas discutidos no livro. O Capítulo 3 contém um resumo dos outros dois, além de definir a LMD e o subsistema de armazenamento local
que serão usados como exemplo no transcorrer do livro. Estes dois fatores definem a missão do processador de comandos da LMD: este deverá compilar (ou interpretar) os comandos da LMD em chamadas para o subsistema de armazenamento. O Capítulo 3 termina com um

exemplo detalhado ilustrando as diversas fases por que passa o processamento de uma consulta.

O Capítulo 4 discute exaustivamente o processamento de comandos da LMD no caso centralizado. A maior parte deste capítulo investiga como desmembrar uma consulta arbitrária em uma série equivalente de consultas mais simples que possam ser convertidas em chamadas para o subsistema de armazenamento. Como em geral há um enorme número de possíveis desmembramentos, o problema de escolher um desmembramento ótimo é também abordado em detalhe. O processamento de atualizações é deixado para o final, pois não apresenta problemas substancialmente mais difíceis do que aqueles encontrados ao se processar consultas.

O Capítulo 5 repete a mesma discussão para o caso distribuído. As várias opções abertas para processamento de consultas distribuídas são inicialmente enumeradas. Em seguida, três delas são discutidas em detalhe, incluindo uma extensão direta da técnica de desmembramento usada no caso centralizado. A maior ênfase recai, naturalmente, nos pontos em que o caso distribuído é mais complexo e exige soluções novas, quando comparado com o caso centralizado.

Os próximos quatro capítulos são dedicados à gerência de transações. O Capítulo 6 introduz, inicialmente, o conceito de transação. Em seguida, analisa os problemas que surgem quando várias transações são processadas simultaneamente em um ambiente distribuído, em presença de falhas de diversos tipos. A solução destes problemas é vital para assegurar o próprio princípio de atomicidade em relação a cada transação submetida ao SGBDD. Será necessário lançar-se mão de mecanismos especiais para controle de início, migração e término de transações distribuídas, controle de concorrência e controle de integridade. No Capítulo 6 são esboçadas soluções para cada um destes problemas. Os Capítulos 7 a 9 dedicam-se à tarefa de detalhar, em profundidade, os principais mecanismos envolvidos nas soluções propostas.

No Capítulo 7 são estudados que mecanismos devem ser acionados quando do início de uma transação, a expor técnicas para controlar a migração de transações e, finalmente, apresentar algoritmos ou protocolos que, ao término da transação, garantem sua atomicidade, esta última sendo de crucial importância para o correto funcionamento do banco de dados. Inicialmente, é revisto o ambiente distribuído onde a transação opera. Em seguida, são examinados em detalhe os mecanismos que atuam quando do ínicio, migração e término de transações.

O Capítulo 8 aborda o problema de controle de concorrência. As principais anomalias que surgem com execução simultânea de transações são ilustradas. Em seguida, os rudimentos da teoria de serialização são elaborados, indicando-se clara e precisamente quando um mecanismo de controle de concorrência deve ser considerado correto. Dois métodos para construção de mecanismos de controle de concorrência são então discutidos: bloqueio e pré-ordenação. Em primeiro lugar, o método de bloqueio é discutido detalhadamente para o caso centralizado. Isto feito, uma família de mecanismos usando técnicas de bloqueio distribuído é apresentada. Por fim, são definidos vários mecanismos que seguem a metodologia de pré-ordenação.

Finalmente, o Capítulo 9 descreve duas técnicas de controle de integridade. A primeira técnica baseia-se em um mecanismo de atas, armazenado todas as alterações feitas em um banco de dados local. Conquanto a idéia básica seja simples, devido ao fato de que operam em ambientes complexos, atas enfrentam problemas de sincronismo, de eficiência e de eficácia, que geram mecanismos de controle de integridade bastante sofisticados. A segunda

técnica utiliza a idéia de manter-se duas versões para cada página de dados: a versão certificada, criada pela última transação que terminou corretamente, e a versão corrente. A implementação desta técnica é apresentada de forma incremental até ser possível abranger também mecanismos de controle de concorrência. Ambas as técnicas implementam proteção contra determinados tipos de falhas que ocasionam a perda do conteúdo da memória principal, ou mesmo a perda de partes da memória secundária. Também permitem desfazer os efeitos de uma transação executada parcialmente.

O texto está organizado em ordem crescente de complexidade. Após o material introdutório dos Capítulos 1 e 2, o Capítulo 4 apresenta em uma visão do sistema onde há apenas um usuário, em um ambiente centralizado e em presenca de um meio perfeito e sem falhas. O Capítulo 5 expande esta visão para um ambiente distribuído. Em seguida, os Capítulos 7 e 8 introduzem múltiplos usuários. O cenário é completado com o Capítulo 9 quando são consideradas falhas. Os Capítulos 1, 2, 3 e 6 apresentam uma visão simplificada, porém completa, de todo este material.

POSSÍVEIS USOS PARA O LIVRO

O texto, como um todo, foi preparado tendo em vista um segundo curso na área de Banco de Dados, tanto a nível de graduação quanto de pós-graduação. A duração ótima do curso seria em torno de 55 horas. Quando usado a nível de graduação, as seções marcadas com asterisco poderão ser omitidas e os diversos métodos ilustrados através de exemplos, mais do que através de uma análise dos algoritmos. A nível de pós-graduação, o ritmo do curso pode ser acelerado, deixando uma margem de tempo para cobrir material adicional referenciado nas notas bibliográficas.

Para facilitar outros usos do livro, alguma redundância foi introduzida no texto. Os Capítulos 1, 2, 3 e 6 podem ser cobertos independentemente dos outros e formam a base de um curso para profissionais. Poderão ser cobertos em 12 a 16 horas, dependendo naturalmente do nível de detalhe e do desenrolar pretendido para o curso.

O material dos dois primeiros capítulos pode ser usado em um primeiro curso de banco de dados para cobrir os conceitos e facilidades básicas de sistemas de bancos de dados distribuídos. Dependendo do nível de detalhe, o material pode ser coberto em menos de 8 horas de aula regular (menos de 4 horas se transparências forem usadas).

O primeiro formato foi testado em um curso oferecido no Departamento de Matemática da UnB, no segundo semestre de 1983, como parte de um convênio firmado entre esta e o Centro Científico da IBM. Também foi parcialmente testado por um dos autores no Departamento de Informática da PUC/RJ, no segundo semestre de 1981, enquando professor daquele Instituição. Vários cursos com o segundo formato foram oferecidos no Departamento de Treinamento da Embratel, em 1981, e no Instituto Latino-Americano de Pesquisa em Sistemas da IBM, em 1984.

O texto não inclui exercícios pois o material não se presta bem a este tipo de avaliação. Sugere-se medir o desempenho dos alunos através de trabalhos de pesquisa, divididos conforme os tópicos cobertos em cada capítulo, e baseados na bibliografia fornecida.

AGRADECIMENTOS

A primeira versão deste texto foi preparada, como parte de um projeto interno dos autores, para a Quarta Escola de Computação, realizada no Instituto de Matemática e Estatística da USP de 12 a 20 de julho de 1984. Os autores gostariam de agradecer a gerência do Centro Científico da IBM, inicialmente na pessoa do Dr. Jean Paul Jacob e posteriormente na pessoa do Sr. José Roberto Giannini de Freitas, por terem acolhido entusiasticamente esta iniciativa. Os agradecimentos tambem são estendidos à comissão organizadora da Quarta Escola e, em especial, ao Prof. Routo Terada, coordenador do evento, pela oportunidade que nos foi dada para escrever este texto.

Várias pessoas contribuíram para que o texto pudesse ser escrito no curto espaço de tempo que tivemos. As nossas esposas, Silvia e Vera, não escaparam de revisar partes do texto. Vera, em especial, ajudou-nos a evitar anglicismos imperdoáveis (o que não implica que os que restaram sejam aceitáveis). Aos doutores Lineu C. Barbosa e Glen Langdon, Jr., do Laboratório de Pesquisas da IBM em San Jose, California, fica um agradecimento especial pela cooperação, de várias formas, prestada durante a estada de um dos autores naquele Laboratório quando o texto foi impresso. Este árduo passo de formatação e finalização não teria sido possível sem esta valiosa ajuda. A tarefa hercúlea de acentuar as centenas de páginas do livro não seria concluída se não fosse a ajuda de Denise Carvalho de Morais e Rosa Maria Correia Cabral. Denise, na verdade, acentuou quase metade do texto. Rosa também cuidou para que a impressão inicial fosse feita a tempo.

Por fim, o texto foi formatado utilizando-se GML como linguagem de formatação. A versão final do texto foi processada por JANUS e impressa diretamente em uma impressora IBM 6670/APA, de alta resolução. JANUS é um formatador experimental desenvolvido no Laboratório de Pesquisas da IBM em San Jose, California, por uma equipe liderada pelo Dr. Donald D. Chamberlin. Agradecemos especialmente a ajuda prestada pelo Dr. Bradford W. Wade na transformação da versão corrente de JANUS para que acomodasse as necessidades de acentuação da língua portuguesa. O uso de uma impressora de alta resolução economizou-nos a tediosa tarefa de datilografar o texto final, liberando-nos, pelo menos, mais 45 dias para terminar a versão preliminar. Por outro lado, a flexibilidade alcançada pelo uso de um editor de textos, levou-nos a cair na tentação de infinitas revisões.

Marco A. Casanova
Arnaldo V. Moura

Brasília, junho de 1984

ÍNDICE

Biografia dos Autores

MARCO ANTONIO CASANOVA é professor do Departamento de Informática da PUC-Rio. Formou-se em Engenharia Eletrônica pelo Instituto Militar de Engenharia em 1974, obteve o grau e Mestre em Ciências pela Pontifícia Universidade Católica do Rio de Janeiro em 1976, e os graus de Mestre em Ciências (1977) e Doutor em Filosofia (1979), ambos em Matemática Aplicada, pela Universidade de Harvard. Seus interesses acadêmicos incluem sistemas de gerencia de bancos de dados e sistemas de gerência de documentos hipermídia.

ARNALDO VIERA MOURA é professor do Instituto de Computação da UNICAMP. Formou-se em Engenharia Eletrônica pelo Instituto Tecnológico de Aeronáutica em 1973, obteve o grau de Mestre em Ciências em Matemática Aplicada pelo mesmo Instituto em 1976, e o grau de Doutor em Filosofia, área de Ciência da Computação, pela Universidade da Califórnia, Berkeley, em 1980. Suas principais áreas de interesse incluem linguagens formais, complexidade de algoritmos e projeto e análise de algoritmos concorrentes.

CAPÍTULO 1. CONCEITOS BÁSICOS

Este capítulo apresenta os conceitos básicos e discute as principais funções de um sistema de gerência de bancos de dados distribuídos. Desenvolve ainda argumentos indicando quando bancos de dados distribuídos são uma alternativa interessante.

1.1 INTRODUÇÃO

1.1.1 O que é um Sistema de Gerência de Bancos de Dados Distribuídos

Sistemas de gerência de bancos de dados distribuídos (SGBDDs) estendem as facilidades usuais de gerência de dados de tal forma que o armazenamento de um banco de dados possa ser dividido ao longo dos nós de uma rede de comunicação de dados, sem que com isto os usuários percam uma visão integrada do banco.

A criação de SGBDDs contribui de forma significativa para o aumento da produtividade em desenvolvimento de aplicações, um fator importante desde longa data. De fato, tais sistemas simplificam a tarefa de se definir aplicações que requerem o compartilhamento de informação entre usuários, programas ou organizações onde os usuários da informação, ou mesmo as fontes de informação, estão geograficamente dispersas. Aplicações com estas características incluem, por exemplo, sistemas de controle de inventário, contabilidade ou pessoal de grandes empresas, sistemas de consulta a saldos bancários, outros sistemas voltados para clientes e bancos de dados censitários.

A idéia de SGBDDs é atrativa sob muitos aspectos. Sob ponto de vista administrativo, tais sistemas permitem que cada setor de uma organização geograficamente dispersa mantenha controle de seus próprios dados, mesmo oferecendo compartilhamento a nível global no uso destes dados. Do ponto de vista econômico, SGBDDs podem diminuir os custos de comunicações, que hoje em dia tendem a ser maiores do que o próprio custo de equipamento, com o tradicional declínio dos custos de "hardware". Finalmente, SGBDDs também são atrativos de um ponto de vista técnico pois facilitam o crescimento modular do sistema (em contraste principalmente com um sistema centralizado de grande porte), aumentam a confiabilidade através da replicação das partes críticas do banco em mais de um nó, e podem aumentar a eficiência através de um critério judicioso de particionamento e replicação que coloque os dados próximos do local onde são mais freqüentemente usados (em contraste com acesso remoto a um banco de dados centralizado).

Por outro lado, não é difícil argumentar que sistemas com esta arquitetura levantam problemas de implementação sérios, têm um custo de desenvolvimento elevado, consomem recursos e podem ter uma performance duvidosa.

Em todo o livro usaremos as seguintes abreviações:

BDD - banco de dados distribuído;
SGBD - sistema de gerência de bancos de dados (centralizado ou distribuído);
SGBDD - sistema de gerência de bancos de dados distribuídos.

1.1.2 Por Que Bancos de Dados Distribuídos?

A arquitetura de sistemas utilizando banco de dados tem sido tradicionalmente centralizada. Ou seja, o banco de dados reside em um único computador onde são executados todos os programas acessando o banco. Os usuários podem, por sua vez, estar ligados diretamente, ou através de uma rede, ao computador onde o banco reside. Sistemas com esta arquitetura amadureceram durante a década de 70.

Usualmente, alinham-se a favor de sistemas centralizados argumentos que incluem fatores de economia de escala - o custo por unidade de trabalho decresce à medida que o tamanho do processador cresce (Lei de Grosch) - facilidade de controle de segurança, integridade e implantação de padrões, além de disponibilidade de dados para a gerência.

Estes argumentos precisam ser reavaliados, no entanto. Em primeiro lugar, a centralização dos dados e de responsabilidades choca-se com o objetivo de tornar os dados mais facilmente disponíveis ao usuário final em aplicações geograficamente dispersas. A relação entre os custos de processamento e de comunicações alteraram-se com a queda acentuada do custo de processadores, mas não do custo de transmissão de dados. Ou seja, a estratégia de trazer os dados a um processador central pode ser mais cara do que trazer capacidade computacional ao local de geração ou uso dos dados. Finalmente, sistemas centralizados apresentam vulnerabilidade maior a falhas e nem sempre permitem um crescimento gradativo da capacidade computacional instalada de forma simples e adequada.

Invertendo-se a discussão acima obtém-se argumentos a favor de bancos de dados distribuídos (BDDs).

Em detalhe, os argumentos que tornam BDDs atrativos podem ser postos da seguinte forma. BDDs podem refletir a estrutura organizacional ou geográfica do empreendimento dando maior autonomia e responsabilidade local ao usuário, mas preservando uma visão unificada dos dados. Do lado tecnológico, o desenvolvimento de redes de comunicação de dados permitiu a interligação de um grande número de processadores independentes de forma confiável e com custo previsível. Do ponto de vista puramente econômico, o preço/performance de equipamentos de menor porte tem melhorado substancialmente, obliterando o argumento a favor de equipamentos de grande porte. Além disto, BDDs podem diminuir os custos de comunicação se a maior parte dos acessos gerados em um nó puderem ser resolvidos localmente, sem acesso a dados armazenados em nós remotos. Finalmente, BDDs podem ser projetados de tal forma a melhorar a disponibilidade e confiabilidade do sistema através da replicação de dados, além de permitirem um crescimento modular da aplicação simplesmente acrescentando-se novos processadores e novos módulos do banco ao sistema.

Em contrapartida, o desenvolvimento de SGBDDs de uso genérico não é um problema simples. Em um SGBDD, o conhecimento do estado global do sistema é necessário para se processar consultas e para controle de concorrência, enquanto que não só os dados mas também o controle e informação sobre o estado do sistema estão distribuídos. Portanto, SGBDDs diferem significantemente de SGBDs centralizados do ponto de vista técnico, e um SGBDD não pode ser entendido como a simples replicação de SGBDs centralizados em vários nós.

1.1.3 Arquitetura Genérica para SGBDs Distribuídos

De um ponto de vista bem geral, um SGBD distribuído pode ser visto como uma federação de SGBDs centralizados, autônomos, chamados de *SGBDs locais*, que são interligados por uma camada de software chamada de *SGBD da rede* ou *SGBD global* ("network data base management system").

Um SGBD local é, para todos os efeitos, um SGBD centralizado gerenciando de forma autônoma o *banco de dados local,* exceto que poderá receber comandos tanto de usuários locais quanto da cópia local do SGBD global. O SGBD local faz uso do sistema operacional local que provê as seguintes facilidades básicas: métodos de acesso, gerência de processos e gerência de memória.

A coletividade dos bancos locais constitui, então, uma implementação do banco distribuído.

O SGBD global roda como uma aplicação sob o sistema operacional da rede de comunicação de dados e, portanto, pertence à camada de aplicação na nomenclatura do modelo de referência da ISO. Isto significa que todos os problemas de comunicação de dados e distribuição de recursos é transparente ao SGBD global.

1.1.4 Tipos de SGBDs Distribuídos

SGBDs distribuídos podem ser classificados em dois grandes grupos. Um SGBD distribuído será hamado de homogêneo (em "software") se os SGBDs locais são semelhantes, caso contrário será chamado de heterogêneo. (Uma classificação semelhante pode ser feita do ponto de vista de “hardware", mas não é importante no contexto deste livro). Mais precisamente, um SGBD distribuído é homogêneo se todos os seus SGBDs locais:

- oferecem interfaces idênticas ou, pelo menos, da mesma família;
- fornecem os mesmos serviços aos usuários em diferentes nós.

SGBDs distribuídos homogêneos aparecem com mais freqüência quando a aplicação a que se destinam não existia antes. Conversamente, SGBDs distribuídos heterogêneos surgem usualmente quando há necessidade de integrar sistemas já existentes. A escolha entre uma arquitetura ou outra é influenciada pelo aproveitamento de "hardware" e "software" já existentes e pelo próprio hábito e grau de cooperação esperado dos usuários em caso de uma mudança para um sistema diferente. A alternativa óbvia seria adotar uma arquitetura híbrida.

1.1.5 Classes de Usuários de um SGBD Distribuído

Para apresentação de certas características de SGBDs distribuídos, convém classificar os seus usuários da seguinte forma. Por um lado, os usuários de um SGBD distribuído podem ser grupados em:

- usuários globais, que observam o banco de dados distribuído como um todo e acessam os dados através das interfaces do SGBD global;
- usuários locais que têm contato apenas com o banco de dados local ao nó onde residem e interagem apenas com o SGBD local.

Ortogonalmente, os usuários de um SGBD (centralizado ou distribuído) podem ser classificados em quatro grandes grupos:

- o *administrador do banco,* ("database administrator") responsável pela definição e manutenção do banco (que naturalmente pode ser uma equipe; por tradição mantivemos aqui o singular);
- analistas e programadores de aplicação, responsáveis pelo desenvolvimento de aplicações sobre o banco de dados;
- usuários casuais, como gerentes, que usualmente não são programadores treinados e fazem uso do banco irregularmente;
- usuários paramétricos, como caixas de banco, que fazem uso do banco de dados através de transações paramétricas pré-programadas.

1.2 REQUISITOS FUNCIONAIS DE UM SGBD DISTRIBUíDO

Do ponto de vista do usuário, um SGBD é uma ferramenta de "software" para armazenamento e acesso aos dados operacionais de um empreendimento. Um SGBD distribuído é utilizado quando a forma mais adequada de armazenamento dos dados é ao longo dos nós de uma rede. Um SGBD distribuído deverá facilitar a gerência dos dados em si, o desenvolvimento de aplicações relativas ao empreendimento, além de facilitar a utilização dos dados para fins de planejamento gerencial.

Nesta seção serão apresentadas as características funcionais de um SGBD distribuído necessárias para alcançar estes objetivos gerais. Os requisitos funcionais aqui apresentados terão grande influência no desenrolar dos capítulos seguintes, aplicando-se tanto a SGBDs centralizados, quanto a SGBDs distribuídos, exceto em certos casos. Muitos dos requisitos selecionados, embora originalmente introduzidos para SGBDs seguindo o modelo relacional de dados, refletem preocupações mais gerais e independentes do modelo usado.

1.2.1 Independência Física de Dados

A forma como o banco de dados está armazenado não deve ser visível aos programadores e analistas de aplicação, muito menos aos usuários casuais, sendo responsabilidade exclusiva do administrador do banco definí-la. Em outros termos, os detalhes de armazenamento do banco devem ser transparentes (ou mesmo irrelevantes) ao desenvolvimento de programas de aplicação e ao uso casual do banco, já que a este nível apenas a forma com que os dados estão logicamente estruturados importa. Além disto, espera-se que um bom sistema de gerência de banco de dados permita mudar a forma de armazenar o banco sem alterar os programas de aplicação. Ou seja, deve ser possível alcançar o que se convencionou chamar de *independência física de dados.*

1.2.2 Independência de Localização e Replicação

Em um SGBD distribuído, independência física de dados adquire um significado especial. Este requisito exige que o fato do banco ser distribuído seja um problema de implementação e, portanto, transparente aos usuários (exceto, é claro, na variação do tempo de acesso). Isto significa que devem ser transparentes aos usuários tanto a localização das várias partes do banco de dados ("location transparency"), quanto ao fato destas partes estarem replicadas ou

não ("replication transparency"). O sistema deve, então, ser responsável por localizar os dados e atualizar todas as cópias. Além disto, se os arquivos forem movidos de um nó para outro, ou divididos, os usuários não devem tomar conhecimento do fato (estas são formas de reestruturar um banco de dados distribuído).

Em resumo, os *usuários globais deverão ver o banco de dados distribuído como se fosse centralizado.* Chamaremos estas características de *independência de localização e replicação*.

1.2.3 Autonomia Local

Este requisito está intrinsecamente ligado à estruturação de um SGBD distribuído em uma federação de SGBDs locais autônomos interligados pelo SGBD global. Mais precisamente, na arquitetura genérica adotada exige-se que cada SGBD local mantenha sua autonomia, no seguinte sentido:

- cada SGBD local deve manter controle sobre seus próprios dados, pois uma das motivações para banco de dados distribuídos era justamente a distribuição da responsabilidade dos dados para os próprios usuários locais;
- programas que acessem dados locais devem ser executados localmente, sem que seja necessário consultar outros nós. Não deve haver, portanto, um controle central do banco, nem os dados necessários ao funcionamento do sistema devem ser centralizados (como em um diretório único centralizado).

Como consequência deste requisito, *um usuário local deverá acessar os dados locais como se constituíssem um banco de dados centralizado independente*.

1.2.4 Interfaces de Muito Alto Nível

A linguagem para acesso aos dados armazenados no banco deve ser de muito alto nível, ou seja, com as seguintes características:

- a linguagem deve ser *não-procedimental* no sentido do usuário especificar que dados devem ser acessados e não como eles devem ser acessados (isto é problema do sistema);
- os comandos de acesso ao banco, oferecidos pela linguagem, devem manipular conjuntos de objetos e não apenas um objeto de cada vez;
- os comandos devem ser completamente independentes dos detalhes de armazenamento do banco e da existência de caminhos de acesso pré-definidos.

Estas características são evidentemente importantes para usuários casuais, com pouco treinamento em processamento de dados. Mas há duas outras razões de peso para se exigir interfaces de alto nível. Primeiro, a produtividade de analistas e programadores de aplicação aumentará, pois poderão se concentrar primordialmente na aplicação em si e não na sua implementação (a situação é a mesma quando linguagens de programação de alto nível começaram a aparecer). Segundo, linguagens de muito alto nível podem potencialmente aumentar a eficiência do sistema, no seguinte sentido. Cada comando para acesso ao banco exige a intervenção do SGBD, o que gera um custo adicional considerável. Se o comando manipula conjuntos de objetos de cada vez, este custo adicional é, portanto, diluído em um volume maior de trabalho útil. Este argumento é especialmente importante quando o comando gera acessos a dados remotos exigindo o envio de mensagens através da rede.

1.2.5 Otimização Automática

O uso de interfaces de alto nível perderia o impacto se o processamento de comandos para acesso aos dados fosse ineficiente. O SGBD deve, portanto, conter um otimizador para selecionar os caminhos de menor custo para acessar os dados.

A construção de otimizadores eficientes foi, de fato, um dos principais problemas enfrentados no projeto de SGBDs recentes, especialmente aqueles seguindo o modelo relacional. As soluções apresentadas provaram, no entanto, serem bastante satisfatórias, viabilizando assim o uso de interfaces de alto nível.

1.2.6 Reestruturação Lógica do Banco e Suporte a Visões

Os requisitos de independência física de dados e independência de localização e replicação implicam em que a forma de armazenamento do banco pode ser modificada sem que seja necessário alterar os programas de aplicação. No entanto, reestruturações deste tipo são necessárias para otimizar a forma de armazenamento quando o perfil de utilização do banco muda.

Por outro lado, modificações nas estruturas lógicas do banco (ou seja, na forma como os usuários vêem a estruturação dos dados) são necessárias quando a aplicação muda conceitualmente. O SGBD deve, então, fornecer meios para modificar a estrutura lógica de um banco já existente e criar a nova versão dos dados a partir da antiga.

Reestruturações deste tipo podem causar impacto nos programas de aplicação. Uma estratégia para minorar o impacto de tais mudanças seria criar *visões* através das quais os programas de aplicação acessam o banco. Assim, se a estrutura lógica do banco mudar, em certos casos basta adaptar a definição das visões, sem que com isto os programas de aplicação recebam o impacto das mudanças. É interessante então que o SGBD suporte o mecanismo de visões em complemento a reestruturações no banco (ver Capítulo 2 também sobre os conceitos de estruturas lógicas e visões).

1.2.7 Segurança dos Dados

Uma aplicação baseada em um banco de dados facilita enormemente o acesso aos dados operacionais do empreendimento, o que traz o efeito adverso de facilitar acessos não autorizados a dados classificados. O SGBD deverá, necessariamente, prover meios para definir critérios de autorização para acesso aos dados e meios para assegurar que as regras de acesso serão cumpridas.

1.2.8 Suporte à Administração dos Dados

Um banco de dados é, em geral, uma estrutura complexa com centenas de tipos de objetos diferentes, armazenados de diversas formas. A tarefa de administrar um banco, especialmente se é distribuído, exige ferramentas especiais para ser efetivamente executada. O SGBD deve, então, fornecer um dicionário ou diretório, onde é armazenada a descrição do banco, ferramentas para acesso a este dicionário, além de utilitários para manutenção do banco.

Em especial, o acesso ao dicionário não deverá ser exclusividade do administrador do banco, já que é importante que os usuários casuais, programadores e analistas de aplicação conheçam a definição dos tipos de objetos armazenados no banco.

1.3 ESPECIFICAÇÃO DAS INTERFACES DE UM SGBD DISTRIBUÍDO

Nesta seção serão estudadas as características das interfaces oferecidas tanto a usuários locais quanto a usuários globais, e os efeitos sobre as interfaces resultantes do SGBD distribuído ser homogêneo ou heterogêneo.

1.3.1 Interfaces Globais e Locais

Conforme visto, um SGBD distribuído é constituído de uma coleção de SGBDs locais interligados por um SGBD global. Em cada nó, os usuários locais são servidos pelo SGBD local implementado naquele nó, e os usuários globais (residentes naquele nó) são servidos pela cópia local do SGBD global. Há, portanto, duas classes de interfaces em um SGBD distribuído:

- as *interfaces globais*, oferecidas pelo SGBD global aos usuários globais;
- as *interfaces locais*, oferecidas pelos SGBDs locais aos usuários locais.

Como conseqüência de dois dentre os requisitos básicos que SGBDs distribuídos devem satisfazer, a especificação destas duas classes de interfaces se confunde, no entanto, com a descrição das interfaces de um SGBD centralizado. De fato, lembremos que o requisito de Independência de Localização e Replicação implica em que os usuários globais de um SGBD distribuído deverão ver o banco de dados distribuído como se fosse centralizado. Logo, o SGBD global deverá se comportar como um SGBD centralizado perante estes usuários. Já o requisito de Autonomia Local implica em que um usuário local deverá acessar os dados locais como se constituíssem um banco de dados centralizado independente. Ou seja, o SGBD local é, para efeito dos usuários locais, um SGBD centralizado autônomo.

Assim, para especificar as características das interfaces oferecidas tanto a usuários locais quanto a usuários globais, basta estudar os tipos de interfaces comumente oferecidas por SGBDs centralizados. Este será o assunto do parágrafo seguinte.

1.3.2 Especificação das Interfaces

Recordemos que os usuários locais ou globais podem ser classificados em quatro grandes grupos: o administrador do banco, analistas e programadores de aplicação, usuários casuais, e usuários paramétricos. Para satisfazer as necessidades destas classes de usuários, tradicionalmente um SGBD centralizado oferece:

- uma *linguagem de definição de dados* (LDD) usada para definir novos bancos de dados;
- uma ou mais *linguagens de manipulação de dados* (LMDs) usadas para recuperar e modificar os dados armazenados no banco;
- opcionalmente, uma *linguagem de geração de relatórios* (LGR) que, como o nome indica, é apropriada para extrair relatórios do banco de dados;
- utilitários para manutenção do banco.

A LDD conterá comandos para definir as estruturas lógicas do banco e indicar como estas deverão ser armazenadas fisicamente. A LDD poderá também conter outros tipos de comandos como, por exemplo, comandos para definir critérios de autorização de acesso aos dados. A LDD e os utilitários são as ferramentas de que dispõe o administrador do banco para executar as suas tarefas.

Uma LMD pode ser oferecida como uma linguagem independente para acesso ao banco através de terminais, ou ser oferecida como uma extensão de uma linguagem de programação já existente, chamada de *linguagem hospedeira*. A primeira versão é apropriada à formulação de acessos não antecipados em geral, típicos de usuários casuais. A segunda versão é utilizada no desenvolvimento de programas de aplicação que implementam transações repetitivas, definidas a priori (como desconto de cheques, transferência de fundos, ou consulta a saldo).

Uma LMD, tipicamente, conterá comandos para recuperar, inserir, remover e atualizar dados armazenados no banco. Outros comandos de controle também existirão indicando ao sistema quando começa e termina uma transação, quando os dados de uma transação podem se tornar visíveis a outros usuários, etc... A LDD e as LMDs oferecidas por um SGBD são baseadas no mesmo *modelo de dados*, que fixa os tipos de estruturas lógicas, como árvores, tabelas, etc... que serão usados para organizar o banco de dados do ponto de vista conceitual.

Um usuário paramétrico não utiliza diretamente as interfaces do SGBD, mas sim transações paramétricas pertencentes a uma aplicação desenvolvida sobre o banco de dados. Uma transação é por sua vez definida por um programa de aplicação (um procedimento com parâmetros de entrada) escrito em uma determinada linguagem hospedeira e contendo comandos da LMD. Em geral o termo "transação" tem o sentido mais preciso de uma coleção de comandos que deve ser processada pelo SGBD como se fosse atômica, ou seja, o banco deve refletir o resultado de todos os seus comandos ou de nenhum deles.

Isto conclui a discussão sobre as interfaces de um SGBD.

1.3.3 Influência do Tipo de SGBD Distribuído sobre as Interfaces

Como em um SGBD distribuído homogêneo todos os SGBDs locais oferecem interfaces idênticas, estes últimos usam, então, o mesmo modelo de dados, a mesma LDD e as mesmas LMDs. Logo, uma vez fixadas as interfaces locais, é natural que o SGBD global também ofereça estas mesmas interfaces. Assim, qualquer usuário, local ou global, poderá acessar tanto dados locais quanto dados remotos através da mesma linguagem de manipulação de dados.

Este não é o caso, porém, para sistemas heterogêneos pois SGBDs locais potencialmente usam modelos de dados e LMDs diferentes. Uma opção seria o SGBD da rede oferecer ao usuário global, residente em um dado nó, uma visão do banco de dados distribuído no mesmo modelo de dados que o banco local, e permitir que este usuário acesse dados definidos nesta visão através da própria LMD local. Esta opção é interessante pois não é necessário ensinar uma nova LMD aos usuários residentes em um determinado nó para que possam acessar dados remotos.

Nesta opção, o SGBD global possui, na verdade, uma interface diferente para cada nó. Isto não quer dizer que o SGBD global não suporte uma LMD independente das LMDs oferecidas pelos SGBDs locais, chamada *LMD pivot*, quer seja pela existência de uma nova classe de usuários globais, quer seja para simplificar a estruturação do sistema.

1.4 PRINCIPAIS FUNÇÕES DE UM SGBD DISTRIBUÍDO

As principais funções de um SGBD (centralizado ou distribuído) podem ser grupadas em sete grandes módulos:

- Armazenamento do Diretório de Dados
- Armazenamento do Banco de Dados
- Processamento de Comandos da Linguagem de Manipulação de Dados
- Gerência de Transações
- Controle de Integridade
- Controle de Concorrência
- Controle de Acesso ao Banco

Este agrupamento de funções corresponde diretamente à organização do livro, sendo cada função discutida em um capítulo separado.

Nesta seção, cada uma destas funções será brevemente abordada, criando-se assim um guia para leitura dos outros capítulos. Como a discussão é bastante genérica, aplica-se tanto a SGBDs centralizados quanto distribuídos, exceto em certos casos. Antes, porém, um refinamento da arquitetura de SGBDs distribuídos descrita na Seção 1.1.3, será apresentado, bem como um esquema simplificado do ciclo de acesso a um banco de dados distribuído.

1.4.1 Refinamento da Arquitetura de um SGBD Distribuído

De acordo com a arquitetura genérica adotada, um SGBD distribuído consiste de uma coleção de SGBDs locais interligados pelo SGBD global. O SGBD global pode, por sua vez, ser refinado em três grandes componentes:

1. diretório de dados global (DDG): contém a descrição do banco de dados distribuído. O critério usado para sua distribuição/duplicação é crucial para a performance do sistema;
2. gerente de transações (GT): interpreta e controla o processamento de consultas e transações acessando o BDD;
3. gerente de dados (GD): interface com o SGBD local, fazendo as traduções necessárias no caso de sistemas heterogêneos.

Cada SGBD local possui também um diretório de dados local descrevendo o banco de dados local.

Cada nó da rede conterá então um SGBD local e uma cópia do SGBD global, caso armazene parte do banco, ou apenas o gerente de transações, caso não armazene parte do banco. Um nó poderá conter ou não parte do diretório global, já que a estratégia de alocação deste último não é fixada "a priori" neste nosso cenário.

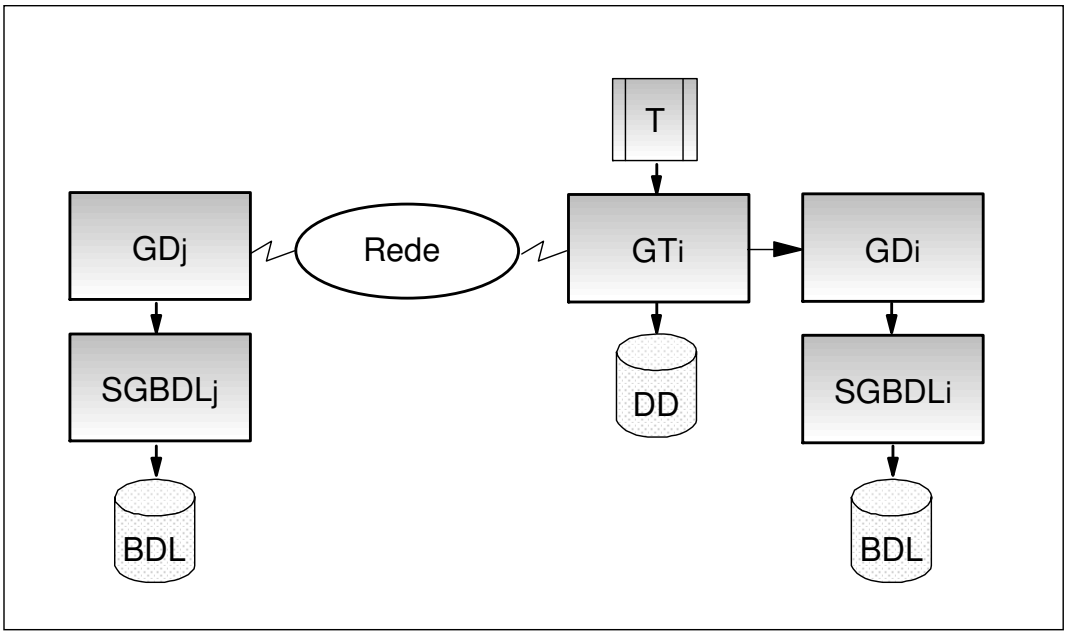

Figura 1.1 - Ciclo de um Comando Distribuído

1.4.2 Ciclo de Processamento em um SGBD Distribuído

Um ciclo típico de acesso ao banco de dados distribuído na arquitetura descrita na seção anterior seria (ver Figura 1.1):

1. uma transação *T* operando no nó *i* (ou usuário acessando o banco através do nó *i*) executa um comando para acessar o banco;
2. o gerente de transações do nó *i* intercepta o comando, acessa o diretório global (que pode estar em outro nó) e cria um plano de acesso ao BDD para obter os dados necessários, ou seja, cria uma seqüência de comandos a serem enviados aos outros nós e para o próprio banco local;
3. gerente de transações do nó *i* envia comandos aos nós envolvidos e coordena a sua execução;
4. o gerente de dados de um nó *j* envolvido no processamento recebe comandos para o banco local e se encarrega de chamar o SGBD local para executá-los (se for necessário, o gerente de dados traduz os comandos para a linguagem de manipulação de dados local);
5. gerente de dados do nó *j* devolve os dados pedidos ao gerente de transações do nó *i*;
6. o gerente de transações do nó *i* completa o processamento do comando submetido, passando os dados para a transação (ou para o usuário).

O acesso ao banco de dados local seguiria o seguinte ciclo esquematizado:

1. uma transação local executa um comando para acessar o banco local, ou o gerente de dados repassa um comando remoto ao SGBD local;
2. SGBD local, por intermédio do sistema operacional, acessa o diretório de dados local;
3. SGBD local analisa o comando;
4. SGBD local chama o sistema operacional para transferir dados do banco de dados local para memória principal;
5. SGBD local extrai dos dados transferidos aqueles necessários para responder ao comando.

O resto desta seção discute quais são as principais funções de um SGBD distribuído e como elas são implementadas.

1.4.3 Armazenamento do Diretório de Dados

O diretório de dados, na sua forma mais simples, contém toda informação sobre o sistema e sobre os bancos de dados e transações já definidos. Esta informação é armazenada sob forma interna para ser usada pelos outros componentes do SGBD e, naturalmente, é fundamental para o funcionamento do sistema.

Este papel básico do diretório pode ser expandido com outras facilidades para que se torne a principal ferramenta de administração do banco e a principal fonte de informação sobre o significado dos dados para os usuários.

Em um SGBD distribuído haverá um diretório de dados global, que descreve o banco de dados distribuído e é usado pelas cópias do SGBD global, e um diretório local para cada SGBD local, descrevendo o banco local. Independentemente do fato de ser global ou local, as seguintes observações podem ser feitas acerca do diretório.

A linguagem de definição de dados (LDD) implementada pelo SGBD constitui-se na principal interface com o dicionário de dados, no seguinte sentido. O diretório contém, inicialmente, apenas informação sobre as suas próprias estruturas e outras transações e usuários especiais. A descrição de novos bancos de dados é então acrescentada ao diretório como resultado do processamento de comandos da LDD; novos usuários são catalogados através de comandos de autorização da LDD; e novas transações são acrescentadas também através de comandos especiais de catalogação.

Há duas formas de implementar o diretório. A rigor, o diretório nada mais é do que um banco de dados e um conjunto de transações e, portanto, pode ser implementado como um banco de dados usando os mesmos mecanismos que os bancos comuns. Ganha-se com isto em uniformidade de interfaces e em compartilhamento de código, já que a própria linguagem de manipulação de dados do SGBD pode ser utilizada para manter o diretório. Por outro lado, perde-se em performance.

De fato, o diretório, embora sendo um banco de dados, é acessado com extrema freqüência. Portanto merece técnicas de armazenamento e acesso especiais. Com isto ganha-se em performance, mas perde-se em complexidade. Estas últimas observações são especialmente importantes no que se refere ao diretório global. Como qualquer outro banco, ele poderá ser distribuído com replicação ou particionamento. No entanto, como acessos ao diretório são necessários a cada consulta, a sua implementação deve ser cuidadosa para minimizar o tráfego adicional de mensagens gerado para acessá-lo.

1.4.4 Armazenamento do Banco de Dados

A função de armazenamento do banco de dados se refere às estruturas de arquivo, métodos de acesso, técnicas de compressão, etc., oferecidas como opções para organização física dos dados em memória secundária. Esta é uma função dos SGBDs locais já que, uma vez definidos os critérios de distribuição do banco, o problema de armazenamento físico é puramente local.

As opções de armazenamento oferecidas pelo SGBD tornam-se aparentes sob forma de comandos especiais da LDD usados para descrição da organização física do banco de dados. Esta parte da linguagem é, às vezes, denominada de *linguagem de definição interna de dados*.

A implementação do subsistema de armazenamento interage fortemente com o subsistema responsável pelo processamento de comandos, considerando-se que o problema básico de um SGBD local consiste em receber um comando do usuário (local ou remoto) e traduzí-lo em acessos físicos a registros armazenados em mémoria secundária. Este problema é exacerbado quando a linguagem de manipulação de dados usada é de muito alto nível, conforme um dos requisitos funcionais impostos a SGBDs distribuídos.

Neste contexto, a arquitetura do subsistema de armazenamento pode ser dividida em dois níveis. O nível mais baixo, que chamaremos de nível físico, consiste de uma coleção de métodos de acesso primários cujas principais funções são:

- esconder dos subsistemas superiores todos os detalhes de armazenamento referentes aos periféricos;
- oferecer alternativas básicas para armazenamento e recuperação de registros físicos, através da definição de estruturas de dados e algoritmos para manipulação destas estruturas.

Este primeiro nível, na verdade, não costuma ser parte do SGDB. Ou seja, o SGBD aproveita os próprios métodos de acesso primários do sistema operacional subjacente, talvez complementando-os com outros mais sofisticados.

O segundo nível do subsistema de armazenamento, que chamaremos de nível interno ou lógico, pode ser encarado como um SGBD independente, definido com os seguintes objetivos:

- oferecer aos subsistemas superiores uma interface melhor do que simplesmente chamadas para as rotinas dos métodos de acesso, mas bem mais simples do que a LMD do sistema completo;
- implementar a base de outras funções do sistema como, por exemplo, controle de integridade.

Este segundo nível oferece então um passo intermediário entre os comandos da LMD do sistema e as rotinas dos métodos de acesso.

1.4.5 Processamento de Comandos da Linguagem de Manipulação de Dados

Por processamento de comandos da linguagem de manipulação de dados (LMD) entenderemos o problema de processar comandos tanto para recuperação de dados quanto para atualização do banco, formulados por um usuário de forma interativa, ou resultantes da execução de transações. Mas excluiremos o problema de implementar o conceito de transação como coleção atômica de comandos, que é discutido na seção seguinte.

Independentemente do sistema ser centralizado ou distribuído, o processamento de comandos poderá ser feito de forma interpretativa, ou através de compilação. A estratégia de compilação provou ser bastante mais eficiente, mas cria certas dificuldades adicionais. De fato, considere o seguinte cenário: um comando é compilado, assumindo uma dada organização física do banco, e o código objeto armazenado para posterior execução; o banco de dados tem sua organização alterada; o código objeto é executado, resultando em erro (pois a organização do banco mudou). Para evitar este problema e manter a capacidade de

reestruturar dinamicamente o banco é necessário, então, um mecanismo de validação em tempo de execução que recompile automaticamente o comando caso necessário. Note que, em uma estratégia interpretativa, este problema não ocorre.

Em um SGBD distribuído, o processamento de comandos é subdividido nas seguintes fases:

1. análise sintática: inclui a tradução dos nomes das estruturas do banco de dados para uma forma interna com auxílio do diretório de dados;
2. otimização: seleciona um plano de execução para o comando;
3. distribuição do plano de execução;
4. execução dos subcomandos pelos SGBDs locais e criação do resultado final.

Há três tarefas particularmente difíceis. A primeira, otimização, requer escolher que cópias dos dados deverão ser usadas, fragmentar o comando original em subcomandos e definir a estratégia de movimentação dos resultados parciais (um extremo seria, por exemplo, mover o resultado de todos os subcomandos para um dado nó, que não precisa ser o nó em que o usuário está, e aí resolver a consulta).

A segunda tarefa, execução dos subcomandos por cada SGBD local, coincide com o problema de processar comandos em um SGBD centralizado. Como esta é uma tarefa complexa, o subsistema correspondente costuma ser estruturado em várias camadas, ou máquinas virtuais, cada uma implementando uma interface que se assemelha a uma LMD de mais alto nível que a anterior. Conjugando-se esta arquitetura com a discussão sobre armazenamento de bancos, a máquina mais interna corresponderia então ao nível físico do subsistema de armazenamento, a máquina imediatamente superior corresponderia ao nível interno do subsistema de armazenamento, e assim por diante até a última máquina, que ofereceria como interface a própria LMD do sistema.

A terceira tarefa que merece comentários nesta fase introdutória surge quando o SGBD distribuído é heterogêneo e, em cada nó, o SGBD global oferece a própria LMD local como interface. Isto significa que comandos para acessar o banco de dados distribuído poderão ser formulados em várias LMDs diferentes. Cada um destes comandos deverá, então, ser fragmentado em subcomandos que serão, por sua vez, traduzidos para a LMD do SGBD local em que serão processados.

Esta tarefa de tradução é função do gerente de dados local. Supondo-se que o banco esteja armazenado em n nós, serão necessários $n(n-1)$ tradutores, já que deverá haver um tradutor para cada par de LMDs locais diferentes.

Esta situação pode ser melhorada instituindo-se um modelo de dados e uma LMD pivots, intermediários e invisíveis, em princípio, aos usuários. Um comando formulado contra o banco distribuído seria então traduzido para a LMD pivot e fragmentado em subcomandos ainda na LMD pivot. Assim, seria necessário a construção de apenas n tradutores.

Finalmente, note que quando atualizações estão envolvidas, o problema recai na tradução de atualizações em visões, que é extremamente difícil.

1.4.6 Gerência de Transações

As funções de gerência de transações são:

- implementar o conceito de transação;
- gerenciar as atividades e recursos do sistema;
- monitorizar o início e término das atividades do sistema.

Destas atividades, a mais importante é a implementação do conceito de atomicidade. Esta propriedade é caracterizada por uma seqüência de comandos bem delimitada de tal sorte que o sistema deve garantir que, ou todos os comandos na seqüência sejam completamente executados, ou o banco não reflete o resultado da execução de nenhum deles, mesmo que o sistema falhe antes da completa execução de todos os comandos da seqüência. O gerente de transações, portanto, transforma o produto final do processador de consultas em uma unidade atômica de trabalho, implementando os comandos de iniciar, migrar, terminar, cancelar, e reiniciar transações.

Em um SGBD distribuído, a implementação destes comandos é consideravelmente mais difícil pois uma transação pode modificar dados armazenados em mais de um nó, o que significa que vários nós têm que cooperar no processo. Por outro lado, por requisitos de projeto, toda informação e mesmo o próprio controle do processo não devem estar centralizados em um único nó.

Com relação aos outros componentes do SGBD, o gerente de transações usa os subsistemas de controle de concorrência e de controle de integridade para executar as suas tarefas.

1.4.7 Controle de Concorrência

Controle de concorrência visa a garantir que, em toda execução simultânea de um grupo de transações, cada uma seja executada como se fosse a única do sistema. Isto significa que, em uma execução concorrente, transações não devem gerar interferências que levem a anomalias de sincronização. As anomalias mais comumente encontradas são perda de atualizações, perda de consistência do banco e acesso a dados inconsistentes.

Mais precisamente, uma técnica de controle de concorrência deve garantir que toda execução concorrente de um conjunto de transações seja *serializável*, ou seja, equivalente a alguma execução das transações em que cada transação é completamente processada antes da próxima começar (i.e., a execução é *serial*).

Há três classes básicas de técnicas de controle de concorrência: técnicas de bloqueio, pré-ordenação ou mistas. As técnicas de bloqueio exigem que um dado seja bloqueado pela transação antes de ser lido ou modificado (embora isto não seja suficiente, conforme veremos). Estas técnicas em geral criam problemas de bloqueio mútuo ("deadlock") que são especialmente difíceis de resolver em um ambiente distribuído.

Nas técnicas de pré-ordenação, a ordem das transações é escolhida "a priori" e as transações são executadas concorrentemente como se fossem processadas serialmente na ordem escolhida. Em geral estas técnicas evitam problemas de bloqueio mútuo, mas criam problemas de reinício cíclico de transações e postergação indefinida de transações ("cyclic restart" e "indefinite postponement").

As técnicas mistas, como o nome indica, tentam combinar as vantagens de bloqueio e pré-ordenação.

1.4.8 Controle de Integridade

Controle de integridade endereça os seguintes problemas:

- implementar o cancelamento, recuperação e término de transações;
- trazer o banco de dados, em caso de falhas, a um estado consistente que reflita apenas o efeito de todas as transações já concluídas;
- recuperar regiões danificadas do banco de dados.

As funções de controle de integridade estão implementadas tanto como parte do SGBD global, quanto como parte dos SGBDs locais. Por exemplo, a recuperação de regiões do banco danificadas por falhas nos periféricos é uma função do SGBD local. Já as funções relacionadas a transações e a manter a consistência do banco é uma função do SGBD global, pois envolve o banco de dados distribuído como um todo.

Os tipos de falhas podem ser classificados em: falhas (de "hardware" ou "software") no processador local, falhas nos periféricos que armazenam o banco e falhas na rede de comunicação de dados.

Não inclui a manutenção automática dos critérios de consistência lógica do banco que porventura tenham sido definidos para o banco (critérios tais como "todo salário deve ser maior do que o mínimo").

1.4.9 Controle de Acesso ao Banco

O objetivo da função de controle de acesso é implementar mecanismos que garantam a segurança dos dados armazenados no banco, permitindo que informação seja lida ou modificada apenas por usuários autorizados.

O controle de acesso pode usar um mecanismo de definição de visões, restringindo a que partes do banco de dados cada grupo de usuários pode ter acesso. Um usuário, ou transação, acessaria o banco de dados através da sua visão. O sistema seria, então, encarregado de traduzir acessos à visão em acessos aos objetos realmente armazenados pelo banco.

Independentemente do mecanismo de visões, o sistema deve oferecer um mecanismo de autorização de privilégios. Como parte da LDD, existiriam comandos de autorização indicando para cada usuário que privilégios (leitura, modificação, inserção, etc.) possui e como estes privilégios se relacionam com as estruturas lógicas do banco. Ao entrar no sistema (ou começar a executar, no caso de transações), haveria um mecanismo de autenticação do usuário (ou transação) estabelecendo a sua identidade perante o sistema. Finalmente, a cada acesso ao banco (ou outra unidade de trabalho mais conveniente, principalmente no caso de transações), o sistema verificaria se o usuário possui os privilégios necessários à execução do acesso.

NOTAS BIBLIOGRÁFICAS

O relatório do CODASYL Systems Commitee [1982] contém uma análise detalhada das vantagens e desvantagens de BDDs. Gray [1979] também discute BDDs de um ponto de vista bem geral. Rothnie e Goodman [1977] contém uma das primeiras discussões genéricas sobre SGBDDs. Mohan [1980] discute brevemente vários problemas levantados pelo projeto de SGBDDs. Draffan e Poole (eds.) [1980] forma uma boa coletânea de artigos. Procuram cobrir todos os aspectos relativos a banco de dados distribuídos, desde a fase de projeto lógico até os mecanismos mais internos de controle de integridade. A única desvantagem é ter sido escrito por um grupo de pessoas, com notação e modelos algo desencontrados. Date [1983] contém também um capítulo específico sobre SGBDDs e discute vários dos problemas abordados neste livro, porém com bem menos detalhes. Draffan e Poole (eds.) [1980] discutem os problemas de sistemas homogêneos e heterogêneos, sendo estes últimos analisados em detalhe em Spaccapietra [1980]. Lindsay e Selinger [1980] introduzem o conceito de autonomia local. Vários protótipos de SGBDDs estão operacionais hoje em dia. Rothnie e Goodman [1977] e Rothnie et all. [1980] descrevem a arquitetura do SDD-1, desenvolvido na "Computer Corporation of America", que muito influenciou a discussão deste capítulo. Williams et all. [1981] dão uma visão geral do Sistema R*, desenvolvido no Laboratório de Pesquisas da IBM em San Jose, California, como continuação do projeto do Sistema R (veja Astrahan et all. [1976] e Blasgen et all. [1979] para uma descrição do Sistema R). Stonebraker e Neuhold [1977] esboçam o que viria a ser o INGRES distribuído. Adiba et all. [1980] descrevem o sistema POLIPHEME e Neuhold e Walter [1982] o sistema POREL, ambos europeus. Smith et all. [1981] descrevem um sistema heterogêneo chamado MULTIBASE, também desenvolvido pela "Computer Corporation of America".

CAPÍTULO 2. BANCOS DE DADOS DISTRIBUÍDOS

Este capítulo inicia com uma proposta para estruturação da descrição de bancos de dados distribuídos, que estende aquela sugerida pela ANSI/SPARC os centralizados. Em seguida, os problemas de projeto e administração de um banco de dados distribuído são discutidos.

2.1 ESPECIFICAÇÃO DE BANCOS DE DADOS DISTRIBUÍDOS

A forma usual adotada para descrição de bancos de dados centralizados é inicialmente apresentada e, em seguida, estendida para bancos distribuídos. Um pequeno exemplo da descrição de um banco de dados distribuído completa a seção.

2.1.1 Descrição de Bancos de Dados Centralizados

A descrição de um banco de dados centralizado usualmente está dividida em três níveis: *conceitual* ou *lógico*, *interno* ou *físico* e *externo* (vide Figura 2.1).

A descrição a nível lógico forma o *esquema conceitual* do banco de dados. O esquema conceitual deve apresentar uma visão de alto nível do banco, independente da forma de armazenamento refletindo apenas a semântica do empreendimento que está sendo modelado. O esquema conceitual consiste de um conjunto de estruturas de dados descrevendo como os dados estão organizados do ponto de vista lógico, além de um conjunto de restrições de integridade indicando que conjuntos de dados corretamente refletem situações do mundo real. A classe de estruturas de dados e restrições de integridade permitidas são determinadas pelo modelo de dados escolhido.

Ao nível interno ou físico, obtém-se uma representação eficiente do esquema conceitual em termos dos métodos de acesso e estruturas de arquivos oferecidas pelo sistema de gerência de banco de dados. O resultado é chamado de *esquema interno* do banco. A existência de um esquema interno separado do esquema conceitual é bastante importante pois os detalhes de armazenamento do banco devem ser transparentes (ou mesmo irrelevantes) ao desenvolvimento de programas de aplicação. O esquema interno não deve ser visível aos usuários ou analistas de aplicação, sendo responsabilidade do administrador do banco definí-lo. Além disto, espera-se de um bom sistema de gerência de banco de dados que permita mudar o esquema interno do banco sem alterar os programas de aplicação. Ou seja, o SGBD deve permitir alcançar o que chamamos de independência física de dados.

Finalmente, pode-se criar uma visão especializada do banco para cada grupo de usuários, ainda do ponto de vista lógico, através da definição de um *esquema externo* para cada grupo. Esquemas externos facilitam o desenvolvimento de aplicações já que focalizam apenas a parte do banco que interessa à aplicação, escondendo parte da complexidade do banco. Esquemas externos também são úteis como uma forma de restringir o acesso a dados classificados por parte de grupos de usuários.

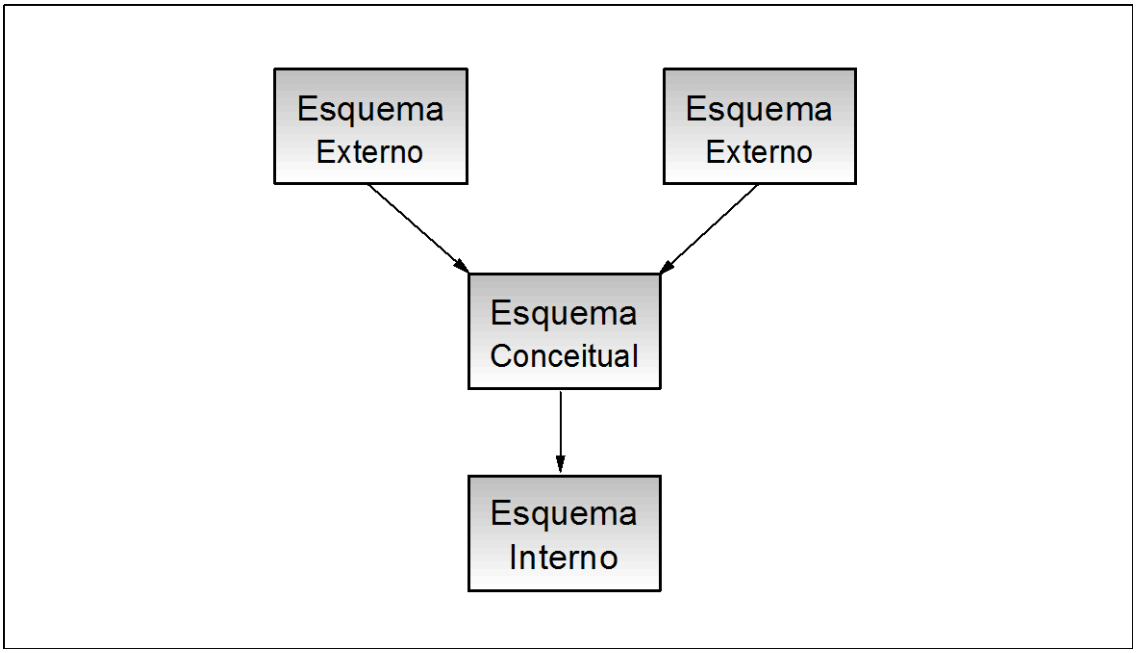

Figura 2.1 - Um Banco de Dados Centralizado

2.1.2 Descrição de Bancos de Dados Distribuídos

A descrição de um banco de dados distribuído, refletindo os requisitos de que a localização e replicação dos dados deve ser transparente aos usuários do BDD e de que o sistemas locais devem manter sua autonomia, é apresentada a seguir (vide Figura 2.2).

O requisito postulando que a distribuição do BDD deve ser transparente ao usuário pode ser entendido como indicando que, a nível lógico, um BDD deve ser visto como se fosse um banco de dados centralizado. Desta forma, deve existir

- um *esquema conceitual global* descrevendo o BDD a nível lógico e ignorando o fato deste ser distribuído e
- vários *esquemas externos globais* descrevendo visões do BDD para grupos de usuários.

Estes dois primeiros níveis são, portanto, idênticos para bancos de dados centralizados e distribuídos.

O esquema conceitual global não é mapeado diretamente em esquemas internos nos diversos nós onde residirá o banco. Esta alternativa aglutinaria em um único passo os problemas de se definir tanto o critério de distribuição do banco como também a estratégia de armazenamento do banco em cada nó. Para evitar este inconveniente, introduz-se para cada nó onde uma parte do banco estará armazenada um *esquema conceitual local* descrevendo o banco de dados local. O mapeamento do esquema conceitual global para os vários esquemas conceituais locais define, então, o critério de distribuição usado.

A estratégia de armazenamento de cada banco de dados local é definida mapeando-se o esquema conceitual local que o define em um *esquema interno local*. Cada nó possui, portando, uma descrição completa, a nível lógico e físico, do banco ali armazenado.

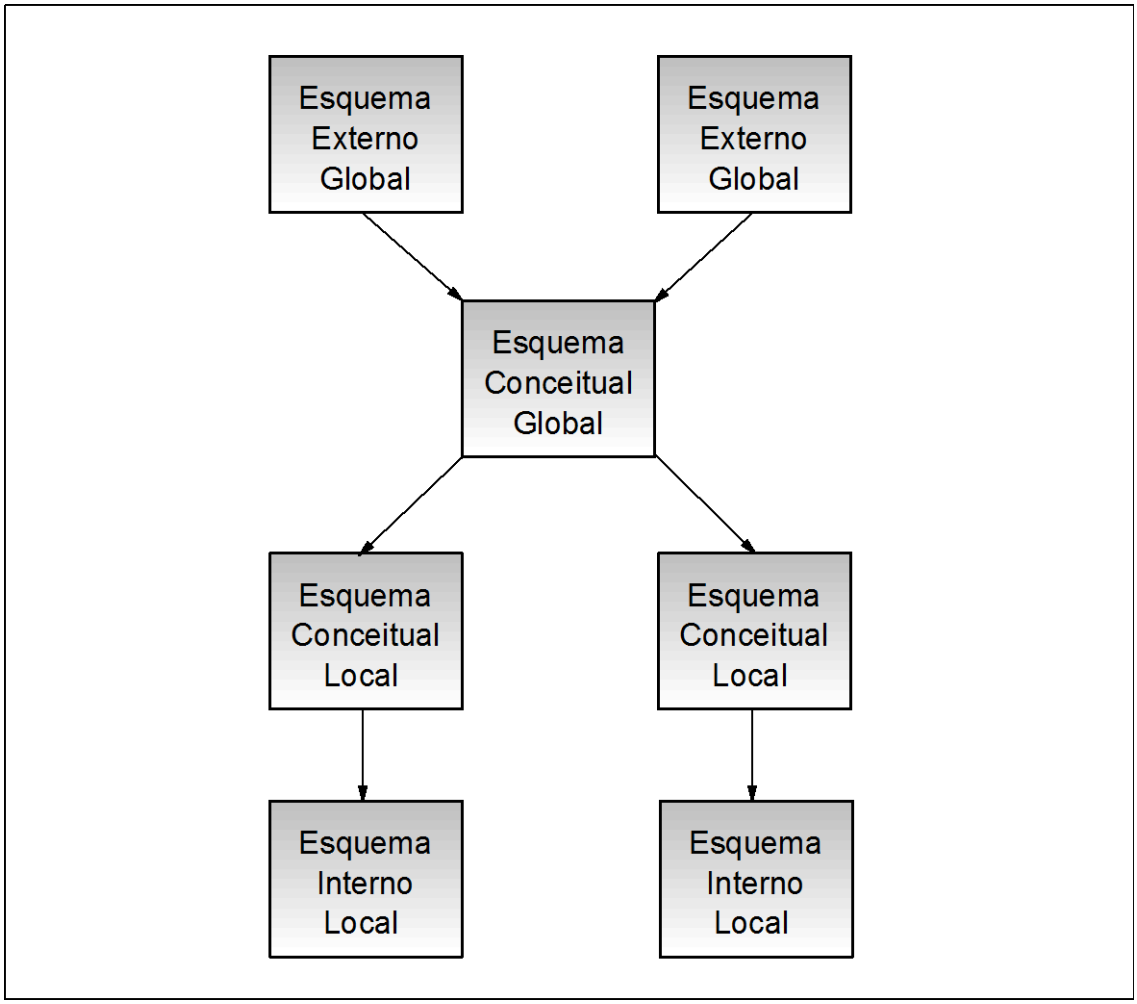

Figura 2.2 - Um Banco de Dados Distribuído

O conceito de um banco de dados local é mais facilmente justificado frente ao requisito indicando que sistemas locais devem manter sua autonomia. Assim, faz sentido introduzir também *esquemas externos locais* em cada nó descrevendo visões do banco de dados local para cada grupo de usuários locais (ver Figura 2.3).
A descrição de um banco de dados distribuído é afetada pelo tipo de sistema da seguinte forma. Se o SGBD for homogêneo, todos os esquemas a nível lógico utilizarão o mesmo modelo de dados. Já no caso de sistemas heterogêneos, teremos a seguinte situação:

esquema conceitual global	no modelo de dados pivot;
esquemas externos globais	podem ser tanto no modelo de dados pivot, para usuários globais, ou em um modelo de dados local, no caso de se desejar oferecer a um usuário local uma visão do BDD no modelo que ele está acostumado;
esquemas conceituais locais	no modelo de dados local;
esquemas externos locais	no modelo de dados local.

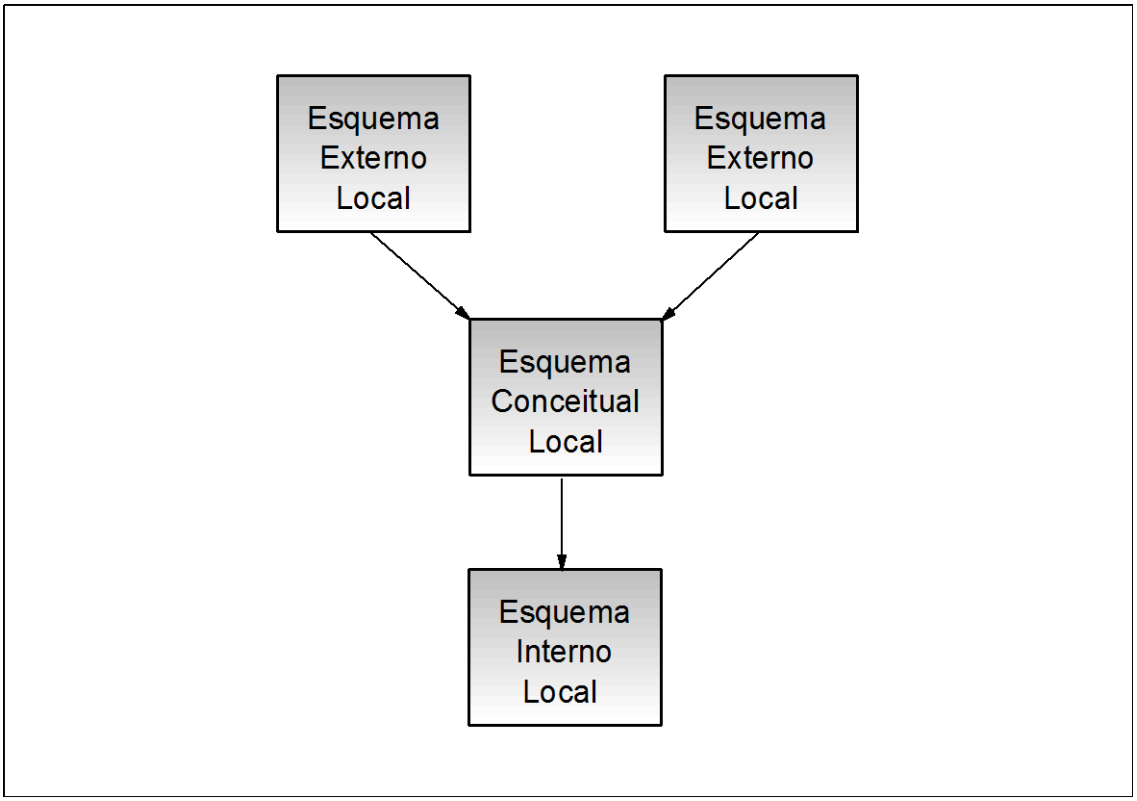

Figura 2.3 - Um Banco de Dados Local

2.1.3 Um Exemplo da Descrição de Bancos de Dados Distribuídos

O banco de dados usado como exemplo refere-se a fornecedores, com número, nome e cidade-sede; peças, com código e nome; e fornecimentos relacionando um fornecedor a cada peça que fornece e indicando a quantidade fornecida. Assumindo que o banco é descrito no modelo relacional, o esquema conceitual seria:

Esquema Conceitual Global

FORNECEDORES [NUMERO,NOME,SEDE]
PECAS [CODIGO,NOME,COR,PESO]
FORNECIMENTO [NUMERO,CODIGO,QUANTIDADE]

Poderíamos definir dois esquemas externos globais da seguinte forma:

Esquema Externo Global A:

Esquema de relação:

FORN_PECA [NUMERO,CODIGO,NOME]

Definição:

FORN_PECA = (FORNECIMENTO * PECAS) [NUMERO,CODIGO,NOME]

Esquema Externo Global B:

Esquema de relação:

FORN_PECA [NUMERO,CODIGO]

Definição:

FORN_PECA = FORNECIMENTO [NUMERO,CODIGO]

Nota: Álgebra relacional será usada para indicar mapeamentos neste exemplo; a operação de junção natural será indicada pelo símbolo '*' e as operações de projeção e seleção serão denotadas da forma usual, ou seja, R[X] indicará a projeção de R na lista de atributos X e R[B], onde B é uma qualificação, indicará uma seleção das tuplas de R que satisfazem B.

Assumindo que o sistema é homogêneo e distribuído em apenas dois nós, os esquemas conceituais locais e a distribuição do esquema conceitual global seriam então descritos da forma abaixo (o primeiro nó conterá todos os fornecedores com sede em Passa Três, todos os fornecimentos em que estão envolvidos e o nome e código das peças; o segundo nó conterá o resto dos fornecedores e seus fornecimentos, além do código, cor e peso das peças.

Esquemas Conceituais Locais:

Primeiro Nó:

FORNECEDORES1 [NUMERO,NOME,SEDE]
PECAS1 [CODIGO,NOME]
FORNECIMENTO1 [NUMERO,CODIGO,QUANTIDADE]

Segundo Nó:

FORNECEDORES2 [NUMERO,NOME,SEDE]
PECAS2 [CODIGO,COR,PESO]
FORNECIMENTO2 [NUMERO,CODIGO,QUANTIDADE]

Mapeamentos Definindo o Critério de Distribuição:

Primeiro Nó:

FORNECEDORES1 = FORNECEDORES [SEDE='PASSA TRES']
PECAS1 = PECAS [CODIGO,NOME]
FORNECIMENTO1 = FORNECIMENTO * (FORNECEDORES1 [NUMERO]

Segundo Nó:

FORNECEDORES2 = FORNECEDORES [SEDE ≠ 'PASSA TRES']
FORNECIMENTO2 = FORNECIMENTO * (FORNECEDORES2 [NUMERO]
PECAS2 = PECAS [CODIGO,COR,PESO]

Como os esquemas internos locais dependem do SGBD local em questão, não faz sentido apresentá-los aqui. Esquemas externos locais também são omitidos.

2.2 PROJETO DE BANCOS DE DADOS DISTRIBUÍDOS

A breve discussão sobre o projeto de bancos de dados distribuídos apresentada nesta seção baseia-se em certas observações simples. Em primeiro lugar, o projeto do esquema conceitual global e o dos esquemas externos globais é inteiramente semelhante ao caso centralizado, já que o banco de dados distribuído deverá se comportar como centralizado perante os usuários globais. Além disto, o projeto dos esquemas internos locais é também idêntico ao de bancos centralizados, exceto que a carga imposta por acessos remotos aos dados locais também deve ser levada em consideração. Portanto, o problema básico de projeto de bancos de dados distribuídos reside no projeto dos esquemas conceituais locais, pois estes refletem a *estratégia de distribuição* do banco.

As estratégias de distribuição são classificadas em *particionamento* e *replicação*. Seja *D* uma estrutura (lógica) de dados do esquema conceitual global. Dizemos que *D* é *particionada verticalmente* (ou *estruturalmente*) quando *D* é mapeada em duas ou mais estruturas (lógicas) de dados que não são idênticas a *D* e que pertencem a diferentes esquemas conceituais locais. Dizemos que *D* é particionada *horizontalmente* (ou *particionada por ocorrência*) quando *D* é mapeada em estruturas idênticas a *D* e pertencentes a dois ou mais esquemas conceituais locais de tal forma que o mapeamento define um particionamento (no sentido matemático) do conjunto de dados associado a *D*.

No exemplo da seção anterior, FORNECEDORES foi particionada horizontalmente em FORNECEDORES1 e FORNECEDORES2, o mesmo acontecendo com FORNECIMENTO. Já PECAS foi particionada verticalmente em PECAS1 e PECAS2.

Dizemos que *D* é *replicada* quando *D* é mapeada em duas ou mais estruturas (lógicas) de dados idênticas a *D* e pertencentes a diferentes esquemas conceituais locais de tal forma que o apeamento de *D* em cada uma destas estruturas é sempre a identidade. Ou seja, existirão cópias idênticas do conjunto de dados associado a *D* armazenadas em dois ou mais nós. A replicação é *total* quando cada banco de dados local contém uma cópia completa do banco. Caso contrário a replicação é *parcial*.

A escolha da estratégia de distribuição do banco exige cuidados especiais pois se vier a gerar um tráfego de dados exagerado entre os vários nós, o custo de comunicação tornará o projeto anti-econômico.

Inicialmente, deve-se verificar se a solução distribuída é de fato uma opção viável. Isto significa, essencialmente, detetar se o banco é fortemente integrado, ou se pode ser dividido em partes mais ou menos independentes; se este for o caso, deve-se então determinar qual a vantagem de descentralizar o banco. Um estudo do perfil da população de transações existentes no sistema centralizado em uso (se este for o caso) deverá ser feito, tentando determinar se é possível dividir o sistema - banco de dados e transações - em subsistemas mais ou menos independentes. Se este for o caso, o custo de transmissão de dados deverá ser reduzido, descentralizando-se o banco e suas funções. Acessos que cortem fronteiras geográficas ainda serão suportados, desde que não sejam muito freqüentes.

Uma vez identificado que a solução distribuída é viável, deve-se escolher a técnica de distribuição, levando-se em conta os seguintes fatores:

- um BDD particionado não fica limitado à memória secundária disponível localmente e, comparativamente a uma solução centralizada (com acesso distribuído), aumenta a confiabilidade e eficiência do sistema, se há um alto grau de localidade de referência;

- um BDD replicado aumenta a confiabilidade, disponibilidade e pode aumentar a rapidez do sistema, mas por outro lado cria problemas de propagação de atualizações nos dados e exige mais memória secundária local.

Naturalmente, o grau de replicação do BDD traduz um compromisso entre o custo de acesso a dados remotos e o custo de atualizar cópias múltiplas.

Um resumo desta discussão encontra-se na tabela abaixo:

RESUMO DAS ESTRATÉGIAS DE DISTRIBUIÇÃO

% de Exceções	Tamanho do Arquivo	Método de Distribuição
--	pequeno	replicação
pequena	grande	particionamento
alta	grande	centralizado

Nota: percentagem de exceções refere-se à freqüência com que uma transação necessita de dados que não estão armazenados localmente.

Por fim, considerações envolvendo "hardware" devem ser mencionadas com relação ao projeto de BDDs. A análise do equipamento necessário deverá responder, pelo menos, às seguintes perguntas: Que processadores existem na organização (ou precisam ser adquiridos)? Qual a configuração mínima dos processadores para suportar o SGBDD? Que periféricos são necessários? Que equipamentos de comunicação de dados são necessário para interligar os processadores?

Isto encerra a nossa breve discussão sobre projeto de BDDs.

2.3 ADMINISTRAÇÃO DE BANCOS DE DADOS DISTRIBUÍDOS

Nesta seção são abordados os problemas, as tarefas e a organização da equipe de administração de um banco de dadosdistribuído.

2.3.1 Organização e Tarefas da Equipe de Administração

A organização da equipe de administração, no caso distribuído, deve acompanhar a própria estratégia de descentralização. Controle centralizado de um banco distribuído não faz sentido. Uma organização plausível seria criar uma equipe local para cada nó onde o banco reside com autoridade para propor e implementar mudanças em detalhe no banco local e nos esquemas externos locais. Haveria ainda uma equipe central com autoridade para coordenar e vetar, se necessário, mudanças no sistema (a serem implementadas pelas equipes locais).

As tarefas tradicionais da equipe de administração de um banco de dados (centralizado) incluem o projeto lógico e físico do banco e sua documentação, definição dos vários esquemas externos em consulta com os analistas de aplicação, definição dos critérios de autorização, criação de rotinas de recuperação do banco, monitoração da utilização do banco e reestruturação do banco. No caso de bancos de dados distribuídos, deve-se acrescentar ainda a tarefa mais geral de garantir a cooperação entre os usuários em prol de uma compartilhação efetiva dos dados.

Três facetas da administração de um BDD merecem especial atenção: documentação do banco, administração dos recursos locais de cada sistema e monitoração do sistema. (A tarefa básica de projeto lógico e físico do banco já foi brevemente abordada na seção anterior).

A documentação do BDD deve tornar claro a todos os usuários o significado dos ítens de dados armazenados pelo banco. Isto requer regras para sistematizar a nomenclatura e a descrição informal dos ítens de dados, definição dos tipos de cada ítem de dados e regras para traduzir um tipo utilizado em uma máquina para o tipo equivalente de outra (e.g., representação e precisão de reais em máquinas diferentes).

A administração da carga imposta a cada sistema que compõe o BDD exige, antes de mais nada, a definição de critérios de medição. Feito isto, é necessário criar regras que assegurem a usuários remotos acesso a recursos locais e que atinjam um balanceamento entre a carga local e a imposta por acessos remotos. Administração da carga inclui, também, definir como será cobrado aos usuários locais e remotos a utilização do sistema.

Finalmente, uma vez estabelecidas regras para administração do banco, a equipe deverá auditar periodicamente o sistema para assegurar a aderência a tais regras. A carga, tempo de resposta e utilização do sistema deverá ser constantemente monitorada, prevendo-se reestruturação do banco ou mudanças nas regras de administração para corrigir desequilíbrios.

2.3.2 Problemas que Afetam a Administração

Os problemas a serem enfrentados pela equipe de administração para atingir os seus objetivos podem ser compreendidos considerando-se três cenários básicos para um banco de dados distribuído.

Se o BDD resultou da interligação de sistemas existentes então certamente aparecerão problemas devidos a: heterogeneidade do sistema global, introdução de padrões globais sem que seja comprometida a autonomia local, critérios de alocação de custos tendo em vista acessos locais e remotos, além do balanceamento do tempo de resposta de acessos locais e remotos.

Se o cenário admite a criação de novos bancos locais de forma semi-autônoma, aparecerão problemas relativos a: definição de regras e responsabilidades locais, descrição da semântica dos dados definidos localmente, grau de cooperação entre os núcleos locais, principalmente no que se refere à alocação de recursos para processamento de acessos remotos.

Finalmente, em um cenário onde o BDD foi criado pela distribuição em nós homogêneos de um sistema centralizado, haverá o problema fundamental de definir uma estratégia de distribuição que otimize o tempo de resposta global, sem penalizar demasiadamente grupos de usuários.

NOTAS BIBLIOGRÁFICAS

O relatório do CODASYL Systems Commitee [1982] discute extensivamente as principais características, formas de organização, problemas de administração, etc. de BDDs. Gross et all. [1980] contém uma discussão interessante dos problemas de administração. Champine [1978] apresenta vários exemplos de BDDs. Shertock [1982] descreve o esforço da Philips do Brasil na área de sistemas distribuídos. Schreiber, Baldissera e Ceri [1980] aborda, de um ponto de vista genérico, aplicações de BDDs. O problema de alocação de arquivos em um

sistema distribuído, assunto abordado muito superficialmente neste capítulo, é de primordial importância no projeto de BDDs. Este problema é bastante semelhante ao problema clássico de transporte em Pesquisa Operacional. Downy e Foster [1982] resume os principais modelos e soluções desenvolvidos para o problema de alocação de arquivos e Wah [1984] contém uma introdução bastante interessante sobre o problema. Apers [1983] contém uma discussão longa sobre o problema de alocação de arquivos em bancos de dados distribuídos.

CAPÍTULO 3. INTRODUÇÃO AO PROCESSAMENTO DE CONSULTAS

O propósito deste capítulo é posicionar o leitor quanto ao problema de se processar consultas e atualizações em bancos de dados centralizados ou distribuídos. Inicialmente, uma visão geral do problema é apresentada, indicando em linhas gerais todo o processo. Em seguida, a LMD tomada como exemplo neste texto, SQL, é definida. Por fim, o subsistema responsável pelo armazenamento interno do banco de dados é descrito. Assim, o problema em questão fica fixado em como traduzir comandos de SQL para operações do subsistema de armazenamento. Em toda discussão será suposto que o SGBDD é homogêneo.

3.1 ETAPAS DO PROCESSAMENTO DE COMANDOS DA LMD

Processamento de consultas e atualizações em um banco de dados distribuído corresponde à tradução de pedidos, formulados em uma linguagem de alto nível, para seqüências de ações elementares sobre os dados armazenados nos vários bancos de dados locais. Mesmo abstraindo os problemas de falhas no sistema e acessos concorrentes aos dados, este é um problema difícil e a LMD é de alto nível e não-procedimental.

O propósito desta seção é indicar em linhas gerais como este problema pode ser subdividido em problemas mais simples. O resultado será uma arquitetura para o processador de comandos da LMD. Todos os problemas referentes a falhas e acesso concorrente aos dados são supostos resolvidos por camadas inferiores do SGBDD. Ou seja, para o processador de comandos tudo se passa como se o sistema fosse perfeitamente confiável e monoprogramado.

A estrutura do processador de comandos da LMD é induzida pela organização imposta à descrição de bancos de dados distribuídos. Lembremos que a descrição é dividida em quatro níveis: externo, conceitual global, conceitual local e interno local. A nível externo, os diversos grupos de usuários vêem os dados que lhes interessam através de esquemas externos EE_1 , ... , EE_m . Cada esquema externo EE_j é mapeado no esquema conceitual global, ECG, que define a totalidade do banco a nível conceitual global. A distribuição do banco é definida em duas etapas. Inicialmente, a distribuição é definida a nível lógico mapeando-se o esquema conceitual global em uma coleção ECL_1 , ... , ECL_n de esquemas conceituais locais, um para cada nó onde o banco será armazenado, onde o esquema ECL_i descreve o banco de dados local do nó i a nível lógico. Como último passo do processo, a estrutura interna de cada banco de dados local é definida por um esquema interno EI_i (o esquema conceitual local ECL_i, naturalmente, deverá ser mapeado no esquema interno local EI_i).

Assim, um comando sobre um esquema externo sofrerá as seguintes transformações (ver Figura 3.1):

1. tradução para o esquema conceitual global;
2. tradução para os esquemas conceituais locais;
3. processamento local e transferência de dados; e
4. pós-processamento global.

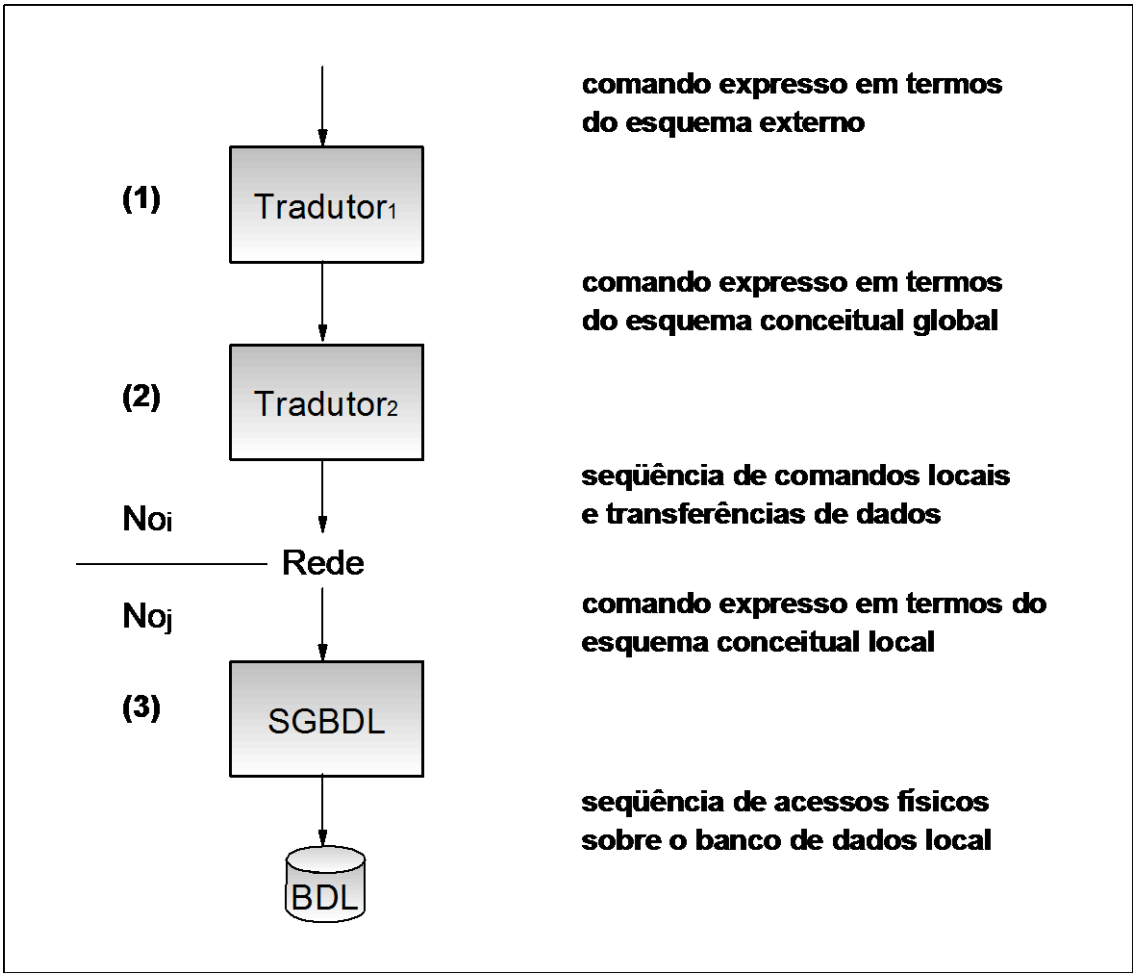

Figura 3.1 -Processamento de Comandos Distribuídos

Consideremos inicialmente o caso mais simples de um sistema homogêneo. A etapa inicial, tradução para o esquema conceitual global, transforma o comando submetido pelo usuário, como formulado em termos do esquema externo a que ele tem acesso, em um comando equivalente, mas formulado em termos do esquema conceitual global. A etapa seguinte, tradução para os esquemas conceituais locais, consiste da tradução do comando, formulado agora em termos do esquema conceitual global, para uma seqüência de comandos locais e transferências de dados. Esta etapa é inteiramente diferente do processamento de comandos em um banco de dados centralizado e deverá resolver os problemas inerentes à distribuição do banco. A fase de otimização nesta etapa é a parte crucial do processo. A terceira etapa, processamento local e transferência de dados, consiste em resolver comandos locais através do SGBD local. A resolução de um comando local poderá, no entanto, envolver transferência prévia de dados.

O processamento de comandos locais é idêntico ao caso centralizado. Novamente, em termos dos esquemas que compõem a descrição do banco de dados local, podemos distinguir duas etapas neste caso (ver Figura 3.2):

1. Tradução para o Esquema Interno.
2. Tradução para Acessos Físicos.

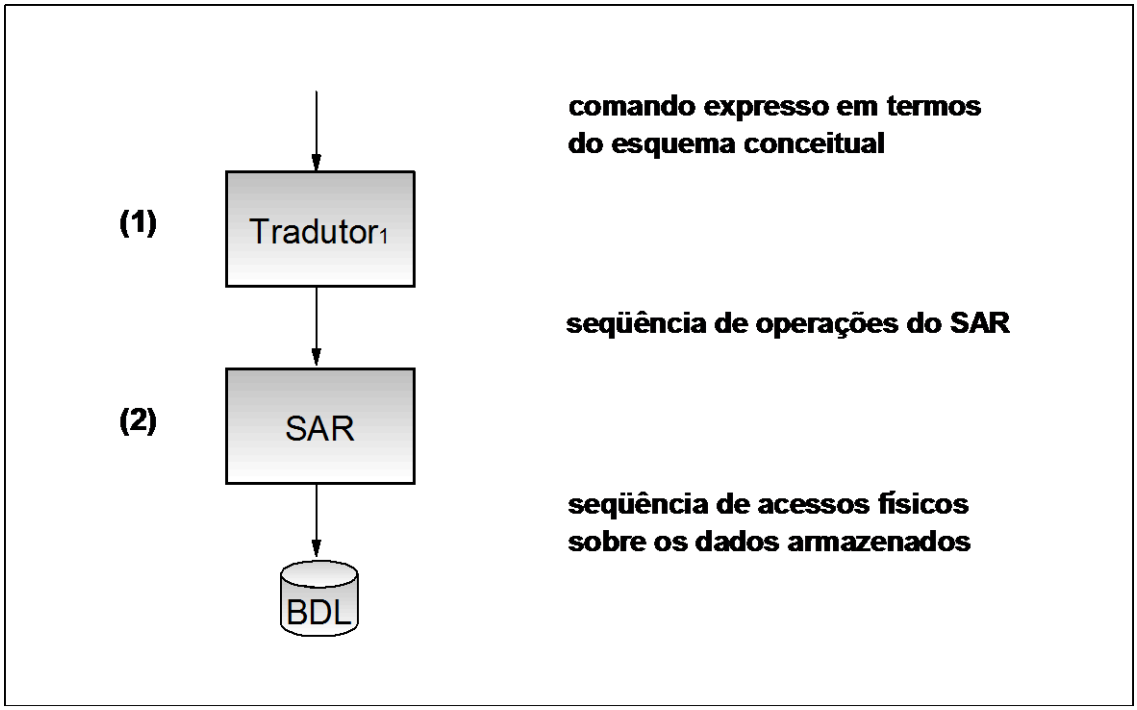

Figura 3.2 - Fases do Processamento de Comandos Locais

A etapa de tradução para acessos físicos será estudada formulando-se uma máquina abstrata, ou subsistema do SGBD, responsável pelo armazenamento físico do banco de dados. Chamaremos esta máquina de *subsistema de armazenamento* (SAR). Este subsistema está dividido em dois níveis:

nível interno: define a interface oferecida por este subsistema ao processador de comandos. A complexidade e eficiência do SGBD dependem em grande parte da variedade e sofisticação das operações oferecidas pelo SAR a este nível.

nível físico: define as estruturas físicas finais e os respectivos acessos físicos a que estão sujeitas.

A etapa de tradução para o esquema interno mapeia um comando formulado em termos do esquema conceitual local para uma seqüência de operações do SAR. Esta é a etapa principal do processo e, como em uma compilação tradicional, possui quatro fases distintas: análise sintática, otimização, geração de código e execução.

Finalmente, a etapa de pós-processamento é necessária pois o resultado de um comando poderá ser deixado sob forma de uma relação distribuída pelas etapas anteriores. Logo, é necessário reunir os fragmentos do resultado em um único nó e entregá-lo ao usuário. Isto conclui a discussão sobre SGBDDs homogêneos. O caso heterogêneo é mais complicado na medida em que os esquemas externos e os esquemas conceituais locais não necessariamente estão no mesmo modelo de dados do esquema conceitual global. Isto torna a primeira e a última etapas mais complexas e está fora do escopo deste texto.

O resto deste capítulo define a LMD e o SAR a serem usados como exemplo no texto. Note que ambos definem, respectivamente, a linguagem-fonte e a linguagem-objeto do processador de comandos.

3.2 UMA LINGUAGEM DE DEFINIÇÃO E MANIPULAÇÃO DE DADOS RELACIONAL

A base para discussão do problema de processamento de consultas e atualizações será uma linguagem de manipulação de dados (LMD) relacional chamada SQL (ou SEQUEL, para "Structured English Query Language"). A escolha é justificada sob vários aspectos. SQL é uma linguagem não-procedimental, da família do Cálculo Relacional, de nível bem mais alto que as LMDs oferecidas pela maioria dos sistemas tradicionais. Portanto, um espectro bem maior de problemas é coberto ao se estudar como processar SQL do que seria possível atendo-se apenas a LMDs tradicionais. Por outro lado, SQL é bastante representativa de uma nova geração de LMDs, sendo adotada por vários sistemas recentes. Finalmente, a escolha de uma linguagem específica para servir de exemplo facilita a apresentação dos algoritmos de otimização.

SQL inclui, ainda, uma linguagem de definição de dados (LDD) que permite a descrição de esquemas conceituais e aspectos relativos a esquemas internos. Apenas a parte relativa ao esquema conceitual será abordada nesta seção.

3.2.1 Definição de Esquemas de Relação em SQL

Para definição de um esquema conceitual (global ou local) SQL oferece um comando para descrição de esquemas de relação. Considere, por exemplo, um esquema conceitual (global ou local) contendo os seguintes esquemas de relação:

FORNECEDOR [NUMERO,NOME,SEDE]
PRODUTO [CODIGO,NOME,MELHOR_FORN]
FORNECIMENTO [NUMERO,CODIGO,QUANTIDADE,LOCAL]
REGIAO [NOME,ESTADO]

Estes esquemas serão descritos em SQL da seguinte forma (já incluindo o tipo de cada atributo):

```
create table FORNECEDOR   ( NUMERO          (integer),
                            NOME            (char(20)),
                            SEDE            (char(5)) )

create table PRODUTO      (  CODIGO         (integer),
                             NOME           (char(10)),
                             MELHOR_FORN    (integer) )

create table FORNECIMENTO(NUMERO            (integer),
                             CODIGO         (integer),
                             QUANTIDADE     (integer),
                             LOCAL          (char(5)) )

create table REGIAO       (  NOME           (char(5)),
                             ESTADO         (char(2)) )
```

Em geral, um esquema conceitual é definido através de uma série de comandos CREATE da forma:

create table <nome de relação> <lista de atributos>

Embora não sejam descritos aqui, convém mencionar que, além deste comando, SQL oferece comandos para adicionar novos atributos a um esquema de relação e para retirar esquemas de relação de um esquema relacional.

3.2.2 Consultas em SQL

As principais características de consultas em SQL serão apresentadas através de exemplos, usando-se para tal o banco de dados relacional cujo esquema conceitual serviu de exemplo na seção anterior. O estado deste banco de dados, apresentado abaixo, servirá de base para compreender o significado das consultas.

FORNECEDOR	Número	Nome	Sede
	10.329	Kopenhagen	SP
	22.345	Garoto	RJ
	41.738	Nestle	SP
	5.938	Praline	DF

PRODUTO	Código	Nome	Melhor_Forn
	342	Balas	Garoto
	2.580	Caramelos	Nestlé
	34	Bombons	Praline

FORNECIMENTO	Número	Código	Quantidade	Local
	10.329	324	10.000	SP
	10.329	34	60.000	SP
	22.345	34	5.000	RJ
	41.738	342	15.000	DF
	41.738	2.580	3.000	RJ
	41.738	34	50.000	MA
	5.938	34	1.000	RJ

REGIÃO	Nome	Estado
	Centro-Sul	SP
	Centro-Sul	RJ

Uma consulta em SQL tem a forma genérica "SELECT-FROM-WHERE" indicando que campos devem ser recuperados (SELECT) das tuplas daquelas relações (FROM) que satisfazem a uma determinada qualificação (WHERE). Por exemplo, a consulta "Obtenha o nome dos fornecedores sediados em SP" seria formulada em SQL como:

```
select   NOME
  from   FORNECEDOR
where    SEDE = 'SP'
```

O resultado seria:

Nome
Kopenhagen
Nestlé

Uma consulta envolvendo duas relações seria "Obtenha o nome dos Fornecedores e a quantidade fornecida relativos, ao produto 34", que em SQL ficaria na seguinte forma:

```
select   F.NOME, FN.QUANTIDADE
  from   FORNECEDOR F, FORNECIMENTO FN
    where        F.NUMERO = FN.NUMERO
   and   FN.CODIGO = '34'
```

Neste exemplo, F e FN são *variáveis da consulta* varrendo as relações denotadas por FORNECEDOR e FORNECIMENTO. Os atributos de cada relação são, então, qualificados por estas variáveis. O resultado desta consulta seria:

Nome	Quantidade
Kopenhagen	60.000
Garoto	5.000
Nestlé	50.000
Praline	1.000

Duas ou mais variáveis podem percorrer a mesma relação, como na consulta "Quais pares de fornecedores têm sede no mesmo estado?":

```
select   F1.NUMERO, F2.NUMERO
  from   FORNECEDOR F1, FORNECEDOR F2
where    F1.SEDE = F2.SEDE
         and      F1.NUMERO < F2.NUMERO
```

A segunda cláusula da qualificação foi adicionada para evitar que o mesmo par fosse recuperado duas vezes apenas com a ordem invertida. O resultado da consulta seria:

Número	Número
10.329	41.738

A qualificação de uma consulta é, em geral, uma expressão booleana formada usando os conectivos 'and', 'or' e 'not', aplicados a comparações entre campos de tuplas ou entre um campo de uma tupla e uma constante. Por exemplo, a consulta "Qual o nome dos fornecedores que não estão sediados em SP e que fornecem ou o produto 34 ou o produto 45?" seria formulada como:

```
select   F.NOME
  from   FORNECEDOR F, FORNECIMENTO FN
where    not F.SEDE = 'SP'
   and   F.NUMERO = FN.NUMERO
   and   (FN.CODIGO = '34' or FN.CODIGO = '45')
```

O resultado seria:

Nome
Garoto
Praline

A forma genérica de uma *consulta simples* em SQL é, então:

```
select   <lista resultante>
   into  <relação resultante>
  from   <lista de relações>
where    <qualificação>
```

onde

<lista de relações> é uma lista de elementos, separados por vírgulas, da forma "R r", onde R é

um nome de relação do banco de dados em questão e r é uma *variável da consulta* varrendo R;

<lista resultante> é uma lista de elementos, separados por vírgulas, da forma "r . A", onde r é uma variável da consulta e A é um atributo do nome de relação varrido por r;

<relação resultante> é um novo nome de relação que terá como atributos aqueles listados em <lista resultante>

<qualificação> é uma expressão booleana sobre comparações, negadas ou não, da forma:

- uma *seleção* da forma 'r.A <op><constante>', onde <op> é um dos operadores {<,≤,=,≥,>} e <constante> é qualquer constante numérica ou alfabética;
- uma *restrição* da forma 'r.A <op> r.B';
- uma *junção* da forma 'r.A <op> s.B';

Nota: A cláusula INTO não faz parte da sintaxe corrente de SQL e poderá ser omitida. Foi introduzida para facilitar a definição de consultas cujo resultado deve ser armazenado.

O resultado de uma consulta simples em um estado do banco de dados é definido da seguinte forma:

1. Forme o produto cartesiano *P* das relações indicadas em <lista de relações>;
2. Selecione as tuplas de *P* que satisfazem a <qualificação>;
3. Projete estas tuplas nos atributos indicados em <lista resultante>. Este será o resultado da consulta.

Por fim SQL permite, ainda, formular a união de duas consultas. Assim, se Q_1 e Q_2 são consultas em SQL, e R_1 e R_2 são nomes de relações, as expressões.

'Q_1 union Q_2' e '$R_1 \cup R_2$'

também são consideradas consultas em SQL, e são chamadas de *consulta-união*.

SQL permite, ainda, a formulação de consultas mais sofisticadas, contendo subconsultas na qualificação, bem como comparações mais poderosas. No entanto, o processamento de consultas mais complexas pode ser reduzido ao processamento de consultas simples através de uma série de transformações.

3.2.3 Atualizações em SQL

Atualizações são formuladas em SQL de forma bem semelhante a consultas. Por exemplo, a remoção "Elimine todos os fornecimentos do produto 34" seria formulada como:

```
delete   FORNECIMENTO
where   codigo=34
```

Inserções podem ser de apenas um registro como, por exemplo, "Adicione um novo produto com código '35' e nome 'Pirulito'", que seria indicada por:

```
insert into PRODUTO:
        <'35','Pirulito'>
```

ou podem manipular vários registros, como no seguinte caso: "Crie uma nova tabela contendo o nome e número de todos os fornecedores do produto '34'":

```
insert into TEMP:
select   F.NOME, F.NUMERO
      from        FORNECEDOR F, FORNECIMENTO FN
where   F.NUMERO = FN.NUMERO
   and  FN.CODIGO = '34'
```

A inserção de múltiplos registros difere da forma de consulta com cláusula INTO apenas no fato de poder referenciar relações já existentes no banco.

Atualizações também podem alterar apenas o valor de um ou mais campos de um grupo de tuplas, como na seguinte operação "Mude todas as quantidades fornecidas para milhares de unidades (ou seja, divida por mil todas as quantidades fornecidas)":

```
update FORNECIMENTO
      set QUANTIDADE = QUANTIDADE / 1000
```

Isto conclui a nossa breve descrição de atualizações em SQL.

3.2.4 Definição de Esquemas Externos e Mapeamentos em SQL

SQL permite definir esquemas externos através de um comando especial para introduzir novos esquemas de relação por definição. Esquemas introduzidos desta forma serão chamados de *visões*. Por exemplo, considere o seguinte esquema externo sobre o banco de dados usado como exemplo nas duas seções anteriores:

Relação do esquema externo:

```
PRODUTO_NESTLE [ CODIGO,NOME,MELHOR_FORN ]
```

Mapeamento para o Esquema Conceitual:

> A relação associada a PRODUTO_NESTLE conterá triplas da forma *(c,n,m),* onde *c* é o código de um produto fornecido pela Nestlé, *n* é o nome do produto e m/ é o seu melhor fornecedor.

O único esquema de relação deste esquema externo e sua definição seriam descritos em SQL da seguinte forma:

```
define view   PRODUTO_NESTLE ( CODIGO,NOME,MELHOR_FORN )
            as select  P.CODIGO, P.NOME, P.MELHOR_FORN
      from    PRODUTO P, FORNECIMENTO F
            where      F.NUMERO = '41.738'
               and     F.CODIGO = P.CODIGO
```

A forma geral do comando DEFINE VIEW descrevendo um esquema de relação introduzido por definição será, então:

```
define view <esquema de relação> as <consulta>
```

Uma vez definidas, pode-se consultar visões como qualquer outra relação. No entanto, atualizações sobre visões criam certos problemas ao serem traduzidas para o esquema conceitual e, usualmente, são restritas a classes bem específicas.

Note que o mapeamento definindo o novo esquema de relação em termos do esquema conceitual é descrito, no comando DEFINE VIEW, através de uma consulta. Da mesma forma, os mapeamentos do esquema conceitual global para os esquemas conceituais locais podem ser descritos por consultas em SQL, assumindo que todos os SGBDs locais são baseados no modelo relacional. Esta observação é validada pelo fato de ambos os tipos de esquemas admitirem apenas relações como estruturas de dados neste caso. Exemplos serão apresentados na Seção 1.4.

Já o mapeamento dos esquemas conceituais locais para os esquemas internos em geral não pode ser descrito através desta técnica, pois os esquemas internos não admitem apenas relações como estruturas de dados.

Isto conclui a nossa discussão sobre SQL.

3.3 O SUBSISTEMA DE ARMAZENAMENTO

Esta seção define o subsistema de armazenamento (SAR) que será adotado como exemplo em todos os capítulos que tratam de processamento de comandos da LMD.

O SAR é responsável pelo armazenamento interno de bancos de dados, incluindo as estruturas auxiliares de acesso, além de oferecer uma interface consistindo de uma série de operações sobre estas estruturas internas. A descrição do SAR clarifica, portanto, qual a linguagem-objeto do tradutor de comandos da LMD e isola o otimizador dos detalhes de armazenamento, exceto no que se refere à computação da função de custo. O SAR servirá também como meio de se abstrair os vários métodos de acesso e suas operações (não estudado neste texto).

O SAR será dividido em dois níveis:

nível interno: define a interface com o processador da LMD;
nível físico: define as estruturas físicas finais e os respectivos acessos físicos.

Esta seção tratará de cada um destes níveis em separado. Lembremos, neste ponto, que o banco de dados é suposto relacional, o que significa que as estruturas lógica de dados são relações.

3.3.1 O Nível Interno do SAR

Esta subseção define as estruturas internas e operações que compõem a interface oferecida pelo SAR ao processador de comandos da LMD.

3.3.1.1 Estruturas Internas

As estruturas internas serão *tabelas* definidas como seqüências de *registros internos* (ou simplesmente *registros*) semelhantes. Um registro interno é a menor unidade que o SAR acessa. Cada registro possui um campo especial que o identifica univocamente na tabela. Este campo é chamado de *identificador do registro* e pertence a um tipo de dados especial, chamado *IDR*.

Uma tabela é descrita através de um *esquema de tabela* $T[A_1, ... , A_n]$, onde *T* é o *nome da tabela* e $A_1, ... , A_n$ são os *atributos* da tabela, ou seja, nomes para os campos dos registros da tabela. Assume-se que cada atributo está associado a um tipo de dados, omitido da definição do esquema por simplicidade.

Quatro tipos de tabelas serão considerados: tabelas externas, tabelas de inversão, tabelas internas e tabelas transientes.

Tabelas externas residem em memória secundária e conterão relações do banco ou relações temporárias resultantes de comandos.

Tabelas de inversão (TINVs) são tabelas agindo como arquivos invertidos para tabelas externas. Ou seja, cada TINV *U* está associada a uma *tabela subjacente T* e a uma lista $L=(L_1 , ... ,L_n)$ de *atributos de inversão* de *T* (com relação a *U*). Além disto, para cada valor $v=(v_1, ... , v_n)$ ocorrendo na projeção de *T* em *L*, há um registro $<i,v_1, ... ,v_n,l>$ em *U*, onde *i* é o identificador do registro, e *l* é a lista de todos os identificadores de registros *t* de *T* tais que $t[L]=v$. Neste caso, assume-se que *U* é descrita por um esquema da forma $U[IDR,L_1, ... , L_n ,P]$, onde o tipo de dados de *IDR* é IDR e este atributo corresponde ao campo contendo o identificador do registro; $L_1 , ... , L_n$ é a lista de atributos de *T* onde foi feita a inversão; o tipo de dados de *P* é *lista de identificadores* e este atributo corresponde à lista de identificadores em cada registro. Note que uma TINV pode ser implementada como uma árvore-B, uma tabela de "hashing" ou outra estrutura adequada, mas a forma exata é abstraída neste modelo.

Uma *tabela interna* é idêntica a uma tabela externa, exceto que reside em memória principal. Tipicamente, conterá resultados intermediários de consultas que são suficientemente pequenos para ocupar pouca memória.

Uma *tabela transiente* é uma estrutura de dados usada como forma de comunicação entre operações do SAR que funcionam como um par produtor-consumidor. Consiste, essencialmente, de uma area de memória e operações para se acrescentar e retirar registros da área. Tal estrutura permite a uma operação passar gradualmente os registros de um resultado intermediário para a operação seguinte. É usada quando o resultado intermediário é grande demais para ser armazenado em uma tabela interna e a operação seguinte não necessita da tabela completa para iniciar o processamento dos seus registros.

Isto conclui a descrição das estruturas internas. Deixaremos em aberto os detalhes de implementação dos diversos tipos de tabelas, já que isto é usualmente coberto em textos versando sobre estruturas de dados.

3.3.1.2 Operações sobre Tabelas

A interface do SAR oferece três operações para criação/eliminação de tabelas, úteis ao processamento de comandos da linguagem de definição de dados:

CRIA_TAB(T,X)

Cria uma tabela externa T com atributos X.

CRIA_INV(T,Y,U)

Cria uma tabela de inversão *U* invertendo *T* em *Y*. Os atributos de *U* são fixados "a priori", como discutido anteriormente. *T* deverá ser uma tabela externa.

Esta operação é útil também em certas estratégias para processamento de consultas, onde inversões temporárias são criadas apenas para o processamento de uma consulta. Porém, neste texto não será descrita nenhuma estratégia que faça uso de inversões temporárias.

DESTROI(T)

Destroi a tabela *T*.

Para processamento específico de comandos da linguagem de manipulação de dados, a interface a nível interno do SAR oferece quatro classes de operações: união, ordenação, pesquisa seqüencial, pesquisa direta e junção. Cada uma destas classes de operações pode ter mais de uma realização no SAR e, em particular, as tabelas recebidas como entrada podem ser de vários tipos. As diferentes formas de realização serão discutidas junto com a descrição das operações.

Seja <*op*> um dos operadores {<,≤,=,≥,>}. Sejam *T* e *U* nomes de tabelas e *X* e *Y* listas de atributos de *T* e *U*, respectivamente, do mesmo comprimento. Usaremos no que se segue *P(T)* para indicar uma expressão booleana envolvendo comparações, negadas ou não, da *forma* *T.X* <*op*> *T.Y* ou *T.X* <*op*> <*constante*>. Usaremos ainda *P(T,U)* para representar uma expressão booleana sobre comparações da forma *T.X* <*op*> *U.Y*.

Sejam $T[IDR, A_1, ..., A_m]$ e $U[IDR, B_1, ..., B_n]$ e *X*, *Y* listas de atributos de *T* e *U*, respectivamente. As operações do SAR serão as seguintes:

ORD(T,X,tipo;V)

Uma ordenação de *T* sobre *X* cria uma nova tabela *V* com os registros de *T* ordenados por *X* em ordem ascendente, se tipo for 'asc', ou em ordem descendente, se tipo for 'desc'. *T* não poderá ser uma tabela transiente, neste caso.

UNIÃO(T,U;V)

Uma união de *T* e *U* cria uma nova tabela *V* contendo todos os registros de *T* e *U*, sem eliminar duplicatas.

PESQ_SEQ(T,X,P(T);V)

Uma *pesquisa sequencial* em *T* cria uma nova tabela *V[IDR,X]* contendo os registros de *T* que satisfazem a *P(T)*, com os campos que não estão em *X* eliminados. Duplicatas resultantes do processo não são eliminadas e a identificação dos registros de *V* é gerada automaticamente.

Esta operação é implementada através de uma pesquisa sequencial em *T*.

PESQ_DIR(T,X,P(T),U,Q(T);V)

Para uma *pesquisa direta* em *T* via *U,* é necessário que:

- *U* seja uma tabela de inversão para *T* sobre uma lista de atributos *Y*;
- *T* seja uma tabela externa ou uma tabela interna (são os únicos tipos para os quais é possível criar uma inversão);
- todos os atributos de *T* referenciados em *Q(T)* devem pertencer à lista de inversão *Y;*

A definição desta operação é idêntica à da anterior, exceto que as tuplas recuperadas de *T* devem satisfazer a $P(T) \wedge Q(T)$.

Quanto à sua implementação, *U* é acessada identificando-se cada registro *u* de *U* que satisfaz *Q* (isto é possivel pela terceira restrição acima). Em seguida, cada registro *t* em *T*, cujo identificador ocorre na lista de identificadores em *u*, é recuperado. Tais registros necessariamente satisfazem *Q*. A projeção em *X* de cada registro *t* que satisfaz a *P(T)* forma, então, um registro da tabela resultante. Duplicatas não são eliminadas.

JUNÇÃO(T,U,X,Y,P(T,U),P(T),P(U);V)

De forma genérica, uma *junção* de *T* e *U* constrói uma nova tabela *V[IDR,X,Y]* tal que *v* é um registro de *V* se e somente se existem registros *t* e *u* em *T* e *V* tais que (i) *v[X] = t[X]*, *v[X]=u[Y]* e *v[IDR]* é um novo identificador; (ii) *t* satisfaz a *P(T);* (iii) *u* satisfaz a *P(U);* (iv) *t* e *u* concatenadas satisfazem a *P(T,U).*

Esta operação coincide, portanto, com a operação tradicional de junção da álgebra relacional, seguida de projeções sobre *X* e *Y*. Dois algoritmos serão consideradas aqui: junção interativa e junção por intercalação.

A junção interativa é implementada da seguinte forma: os registros de *T* satisfazendo a *P(T)* são recuperados um a um; para cada registro *t* recuperado, a tabela *U* é pesquisada recuperando-se um a um todos os registros *u* satisfazendo a *P(U).* A partir de *t* e *u* constrói-se um registro *v* que se satisfizer a *P(T,U)* é acrescentado à tabela resultante. Mais precisamente, temos a seguinte definição em pseudo-código:

```
JUNÇÃO_INTERATIVA(T,U,X,Y,P(T,U),P(T),P(U);V):
begin
        inicie V como vazia;
        foreach registro t de T que satisfaz a P(T) do
        begin
                substitua t em P(T,U) criando Q(U);
                foreach registro u de U
                        que satisfaz a P(U) e Q(U) do
                begin
                        acrescente um registro v a V
                                criado a partir de t e u;
                end
        end
end
```

Refinamentos deste algoritmo são obtidos utilizando-se as operações de pesquisa seqüencial e pesquisa direta para acessar os registros de *T* e *U*. O resultado da pesquisa em *T* (ou *U*), em

qualquer caso, é passado sob forma de uma tabela transiente para o corpo do algoritmo de junção. Se pesquisa direta for escolhida para *T* (ou *U*) então *T* (ou *U*) pode ainda ser uma tabela externa ou uma tabela interna; se pesquisa seqüencial for adotada, *T* (ou *U*) pode ser uma tabela externa, uma tabela interna ou uma tabela transiente. O resultado *V* também pode ser criado como uma tabela externa, uma tabela interna ou uma tabela transiente.

Junções sobre tabelas de índices também oferecem opções interessantes para processamento de consultas, embora não sejam adotadas neste texto. Portanto, a operação de junção não será definida sobre tabelas de índices.

A junção por intercalação se aplica quando *P(T,U)* é da forma *T.X <op> U.Y & Q(T,U)*, onde *<op>* é um dos comparadores anteriormente apresentados. Ou seja, deverá haver uma comparação distinguida, possivelmente conjugada com outras comparações. Este tipo de junção exige ainda que os registros de *T* e *U* estejam ordenados pelos atributos *X* e *Y*, respectivamente, em uma *ordem de junção* compatível com *T.X <op> U.Y*, qual seja:

- se *<op>* for '=', então a ordenação poderá ser tanto ascendente quanto descendente (mas a mesma para *T* e *U*);
- se *<op>* for '<' ou '≤', a ordenação deverá ser ascendente;
- se *<op>* for '>' ou '≥', a ordenação deverá ser descendente;

Nesta implementação de junção, os registros de *T* são recuperados um a um e, como na junção interativa, para cada registro *t* de *T*, registros da tabela *U* são recuperados, criando-se registros da tabela *V*. A posição do primeiro registro *u* de *U* que casa com *t* é guardada. Assim, quando o próximo registro *t'* de *T* é lido, a tabela *U* é pesquisada a partir da posição guardada. Mais precisamente, em pseudo-código temos:

```
/*
    P(T,U) é da forma T.X<op>U.Y & Q(T,U)
    T e U estão ordenados por X e Y, respectivamente,
            em uma ordem de junção compatível com T.X<op>U.Y
*/
begin
        inicie V como vazia;
        if U for vazia then retorne;
        inicie u0 com o primeiro registro de U;
        foreach registro t de T que satisfaz P(T) do
        begin
                substitua t em T.X<op>U.Y criando C(U);
                leia os registros de U a partir de u0
                        até encontrar o primeiro que satisfaz C(U) ou U se esgotar;
                if U se esgotou then retorne;
                inicie u0 com u;
                substitua t em Q(T,U) criando Q(U);
                foreach registro u de U a partir de u0
                        que satisfaz C(U), Q(U) e P(U) do
                begin
                         acrescente um registro v a V criado a partir de t e u;
                end
        end
end
```

3.3.2 O Nível Físico do SAR

O nível físico do SAR define a estrutura física final onde serão armazenados os dados, as estruturas auxiliares de acesso aos dados (i.e., tabelas de inversão) e mesmo informações de controle do SGBDD, como o próprio diretório de dados. O nível físico define ainda as ações elementares (ou acessos físicos) que manipulam as estruturas físicas. Esta seção apresenta apenas um esboço do que seria o nível físico do SAR, pois os detalhes de sua implementação estão intimamente ligados aos métodos de controle de integridade utilizados pelo sistema (veja a Seção 9.2).

Para o nível físico do SAR, a memória secundária está dividida em *segmentos* e cada segmento está dividido em *páginas* de tamanho fixo. Cada página possui um identificador único. A memória principal conterá áreas de trabalho, cada área seria capaz de conter uma página. Toda vez que dados contidos em uma página forem necessários, a página é inicialmente lida para uma área de trabalho e os dados são então extraídos. O mecanismo de gerência das áreas de trabalho é deixado em aberto.

O SAR possui apenas duas ações elementares operando sobre páginas (outras ações serão acrescentadas na Seção 9.2):

R(X) leia todas as páginas cujos identificadores estão contidos no conjunto *X*

W(X) escreva novamente em memória secundária todas as páginas contidas na área de "buffers" cujos endereços estão em *X*

e duas operações sobre o conteúdo das páginas:

r(x,p,s) recupere o conteúdo da página *x* a partir da posição *p* até a posição *p+s-1*

w(x,p) mude o conteúdo da página *x* a partir da posição *p* (o novo valor bem como o seu comprimento foram omitidos da definição da operação por simplicidade)

O conceito de ação elementar será usado novamente nos capítulos sobre processamento de transações, controle de integridade e controle de concorrência.

3.3.3 Definição dos Esquemas Internos

Uma vez descrita a forma interna de armazenamento do banco de dados, é possível, então, discutir como são definidos os esquemas internos. Seja *E* um esquema conceitual (local, no caso distribuído), ou seja, uma coleção de definições de esquemas de relação usando o comando DEFINE TABLE de SQL. Na sua forma mais geral, um esquema interno para *E* seria definido em dois passos, refletindo a estrutura do SAR.

Inicialmente, cada esquema de relação seria mapeado em uma ou mais tabelas externas. O mapeamento indicaria a correspondência entre os registros das tabelas externas e as tuplas da relação denotada pelo esquema. Além do mapeamento de esquemas de relação em tabelas externas, na primeira fase da descrição do esquema interno seriam definidas tabelas de inversão representando inversões sobre atributos das tabelas externas. A escolha de que inversões deverão existir depende das consultas e atualizações que são antecipadas, e influencia decisivamente a performance do sistema.

Em uma segunda fase, as tabelas externas e de inversão são mapeadas em segmentos, e os registros destas tabelas em páginas dos segmentos. Convém mapear registros de tabelas diferentes, que são freqüentemente acessados em conjunto, na mesma página ou em páginas contíguas, sempre que possível. Os critérios que governam a estratégia de grupamento serão chamados de *critérios de contigüidade física*.

A regra mais simples para definir esquemas internos seria mapear cada esquema de relação
$R[A_1 , ... , A_n]$ em uma única tabela externa, cujos registros representam tuplas da relação corrente associada a *R*. Neste caso, a tabela externa poderia ser descrita pelo esquema $R[IDR , A_1 , ... , A_n]$, onde o tipo de dados de ID*R* é IDR e este atributo corresponde ao campo contendo o identificador do registro. Assim, uma tabela externa herda o nome e os atributos da relação que armazena. Da mesma forma, cada tabela externa seria armazenada em um segmento diferente. Em todos os exemplos descritos neste texto adotaremos este mapeamento simples.

A linguagem usada para descrever o esquema interno é, às vezes, chamada de *linguagem de definição interna de dados* (LDID). Ela deverá, no caso do subsistema de armazenamento aqui descrito, permitir a declaração de tabelas externas e de inversão, a declaração de segmentos e critérios de contigüidade física, além dos mapeamentos dos esquemas de relação em tabelas externas e das tabelas nos segmentos. Neste texto não entraremos nos detalhes específicos de uma LDID.

3.4 EXEMPLO DO PROCESSAMENTO DE UMA CONSULTA

Para ilustrar o processamento de comandos da LMD, nesta seção serão apresentadas todas as fases do processamento de uma consulta formulada sobre um esquema externo.

O esquema conceitual global será o seguinte:

ESQUEMA CONCEITUAL GLOBAL

```
create table    PRODUTO          (   CODIGO          (integer),
                                     NOME            (char(10)),
                                     MELHOR_FORN     (integer) )

create table    FORNECIMENTO     (   NUMERO          (integer),
                                     CODIGO          (integer),
                                     QUANTIDADE      (integer),
                                     LOCAL           (char(5)) )
```

Suporemos que a rede possui três nós (RJ, SP e DF, digamos) e que o banco está armazenado em dois nós apenas (RJ e SP).

O primeiro nó (RJ) contém toda a relação de produtos e todos os fornecimentos, exceto aqueles para SP. O esquema conceitual local a RJ será, portanto:

ESQUEMA CONCEITUAL LOCAL AO NÓ 'RJ'

```
create table PRODUTO_RJ          (  CODIGO          (integer),
                                    NOME            (char(10)),
                                    MELHOR_FORN     (integer) )

create table FORNECIMENTO_RJ  (     NUMERO          (integer),
                                    CODIGO          (integer),
                                    QUANTIDADE      (integer),
                                    LOCAL           (char(5)) )
```

Assumiremos que o esquema interno deste nó contém as seguintes tabelas (vide Seção 1.3.3):

ESQUEMA INTERNO LOCAL AO NÓ 'RJ'

PRODUTO_RJ	tabela externa armazenando a relação correspondente ao esquema de quem herda o nome
FORNECIMENTO_RJ	tabela externa armazenando a relação correspondente ao esquema de quem herda o nome
PRODUTO_RJ_COD	tabela de inversão sobre o atributo CODIGO de PRODUTO_RJ
FORNECEDOR_RJ_NUM	tabela de inversão sobre o atributo NUMERO de FORNECEDOR_RJ
FORNECEDOR_RJ_COD	tabela de inversão sobre o atributo CODIGO de FORNECEDOR_RJ

Note que o mapeamento do esquema conceitual para o interno é aquele trivial em que cada esquema de relação corresponde a uma tabela externa de mesmo nome.

O mapeamento do esquema conceitual global para o esquema conceitual local em 'RJ' é definido tratando-se PRODUTO_RJ e FORNECIMENTO_RJ como visões do esquema global:

MAPEAMENTO DO ESQUEMA CONCEITUAL GLOBAL PARA O LOCAL EM 'RJ'

```
define view     PRODUTO_RJ ( CODIGO,NOME,MELHOR_FORN )
    as select   CODIGO, NOME, MELHOR_FORN
               from       PRODUTO

define view     FORNECIMENTO_RJ (NUMERO,CODIGO,QUANTIDADE,LOCAL )
    as select   NUMERO,CODIGO, QUANTIDADE, LOCAL
               from       FORNECIMENTO
              where       LOCAL ¬= 'SP'
```

O segundo nó (SP) contém apenas os fornecimentos relativos a SP. O esquema conceitual local será, então:

ESQUEMA CONCEITUAL LOCAL AO NÓ 'SP'

```
create table   FORNECIMENTO-SP (      NUMERO          (integer),
                                      CODIGO          (integer),
                                      QUANTIDADE      (integer),
                                      LOCAL           (char(5)) )
```

Assumiremos que o esquema interno deste nó contém as seguintes tabelas, por sua vez:

ESQUEMA INTERNO LOCAL AO NÓ 'SP'

FORNECIMENTO-SP	tabela externa armazenando a relação correspondente ao esquema de quem herda o nome
FORNECEDOR-SP-NUM	tabela de inversão sobre o atributo NUMERO de FORNECEDOR-SP
FORNECEDOR-SP-COD	tabela de inversão sobre o atributo CODIGO de FORNECEDOR-SP

O mapeamento deste esquema conceitual local para o esquema conceitual global será:

MAPEAMENTO DO ESQUEMA CONCEITUAL GLOBAL PARA O LOCAL EM 'SP'

```
define view     FORNECIMENTO-SP (NUMERO,CODIGO,QUANTIDADE,LOCAL )
      as select       NUMERO,CODIGO, QUANTIDADE, LOCAL
           from       FORNECIMENTO
          where       LOCAL = 'SP'
```

Como dito anteriormente, o terceiro nó não armazena nenhuma parte do banco de dados.

Os mapeamentos do esquema conceitual global para os esquemas conceituais locais são úteis para traduzir atualizações globais em atualizações locais. Para se processar consultas, necessita-se do mapeamento inverso, ou seja, necessita-se considerar cada relação do esquema conceitual global como uma visão sobre a união dos esquema conceituais locais. O mapeamento inverso será, então:

MAPEAMENTO DO ESQUEMA CONCEITUAL GLOBAL PARA OS LOCAIS

```
define view     PRODUTO ( CODIGO,NOME,MELHOR_FORN )
      as select       CODIGO, NOME, MELHOR_FORN
           from       PRODUTO_RJ

define view     FORNECIMENTO (NUMERO,CODIGO,QUANTIDADE,LOCAL )
      as select       NUMERO, CODIGO, QUANTIDADE, LOCAL
           from       FORNECIMENTO_RJ
       union
         select       NUMERO, CODIGO, QUANTIDADE, LOCAL
           from       FORNECIMENTO-SP
```

Considere agora um esquema externo, definido sobre o esquema conceitual global introduzido acima, e contendo apenas uma relação:

ESQUEMA EXTERNO

```
define view     PRODUTO_NESTLE ( CODIGO,NOME,MELHOR_FORN )
          as select   P.CODIGO, P.NOME, P.MELHOR_FORN
           from       PRODUTO P, FORNECIMENTO F
         where        F.NUMERO = '41.738'
           and        F.CODIGO = P.CODIGO
```

Estaremos interessados em processar a seguinte consulta, Q_0, sobre este esquema externo ("Qual o código dos produtos para os quais a Nestlé é o melhor fornecedor?"):

```
select  CODIGO
    from      PRODUTO_NESTLE
   where      MELHOR_FORN = '41.738'
```

Seguindo a Figura 1, o primeiro passo é traduzir Q_0 para o esquema conceitual global. Isto é feito substituindo-se a referência a PRODUTO_NESTLE pela sua definição. A consulta resultante, Q_1, será:

```
select  P.CODIGO
    from      PRODUTO P, FORNECIMENTO F
   where      F.NUMERO = '41.738'
     and      F.CODIGO = P.CODIGO
     and      P.MELHOR_FORN = '41.738'
```

O segundo passo do processamento consiste em traduzir Q_1 para uma seqüência de transferências de dados e consultas sobre os esquemas conceituais locais. Há várias estratégias para tal, algumas das quais serão discutidas no Capítulo 5. Procederemos aqui da seguinte forma. Inicialmente, traduziremos Q_1 para uma consulta Q_2 sobre a união dos esquemas conceituais locais. Este passo é idêntico ao anterior e consiste em substituir-se cada relação do esquema conceitual global pela sua definição em termos da união dos esquemas conceituais locais:

```
select  P.CODIGO
    from      PRODUTO_RJ P, FORNECIMENTO_RJ F
   where      F.NUMERO = '41.738'
     and      F.CODIGO = P.CODIGO
     and      P.MELHOR_FORN = '41.738'
union
select  P.CODIGO
    from      PRODUTO_RJ P, FORNECIMENTO-SP F
   where      F.NUMERO = '41.738'
     and      F.CODIGO = P.CODIGO
     and      P.MELHOR_FORN = '41.738'
```

Note que Q_2 é a união de duas subconsultas, a primeira das quais é local ao nó 'RJ' enquanto que a segunda envolve uma tabela localizada em 'RJ' e outra localizada em 'SP'.

Adotaremos, então, a seguinte estratégia:

1) A primeira subconsulta de Q_2 é processada da seguinte forma:

   a) Inicialmente a seguinte consulta, Q_{21}, que nada mais é do que a primeira subconsulta de Q_2 ligeiramente modificada para produzir uma relação armazenada, é executada em RJ:

```
select      P.CODIGO
  from      PRODUTO_RJ P, FORNECIMENTO_RJ F
into   CODIGO_RJ
where  F.NUMERO = '41.738'
and    F.CODIGO = P.CODIGO
and    P.MELHOR_FORN = '41.738'
```

   Note que o resultado de Q_{21}, CODIGO_RJ, deverá ser substancialmente menor do que as tabelas em RJ, pois contém apenas parte dos códigos originalmente armazenados em PRODUTO_RJ.

   b) O resultado de Q_{21}, CODIGO_RJ, é movido para o destino final, DF, através de uma transferência de dados T_{21}.

2) A segunda subconsulta de Q_2 é processada da seguinte forma:

   a) Para minimizar os dados transferidos, PRODUTO_RJ é inicialmente reduzida através da seguinte subconsulta Q_{22}:

```
select      CODIGO
  from      PRODUTO_RJ
into   PRODUTO-MELHOR_RJ
where       P.MELHOR_FORN = '41.738'
```

   Esta subconsulta é induzida pela cláusula da qualificação de Q_2 que afeta apenas PRODUTO_RJ.

   Note que o resultado de Q_{22}, PRODUTO-MELHOR_RJ/ certamente é menor do PRODUTO_RJ pois duas colunas foram eliminadas e provavelmente muitas tuplas. Portanto, esta redução é benéfica em termos do número de mensagens que economiza.

   b) PRODUTO-MELHOR_RJ é então movida de RJ para SP através de uma transferência T_{22}.

   c) Uma vez recebida esta tabela em SP, a seguinte subconsulta local, Q_{23}, é executada:

```
select      P.CODIGO
  from      PRODUTO-MELHOR_RJ P, FORNECIMENTO-SP F
into   CODIGO-SP
where       F.NUMERO = '41.738'
and    F.CODIGO = P.CODIGO
```

   Esta subconsulta é induzida pelas cláusulas que se referem apenas a *F* e *P* (onde *P* agora varre PROD-MELHOR_RJ).

   d) Finalmente, o resultado de Q_{23}, CODIGO-SP, é movido para o destino final, DF, através de uma transferência de dados T_{23}.

3) Após CODIGO_RJ e CODIGO-SP terem sido recebidas em DF, a sua união, Q_3, é obtida, produzindo o resultado final da consulta

Sucintamente, esta estratégia pode ser expressa através do seguinte programa concorrente:

$$P = (Q_{21} ; T_{21} // Q_{22} ; T_{22} ; Q_{23} ; T_{23}) ; Q_3$$

indicando que a seqüência de operações $Q_{21} ; T_{21}$ pode ser processada (em RJ) em paralelo com a seqüência $Q_{22} ; T_{22} ; Q_{23} ; T_{23}$ (processada em RJ e SP). Após o término de ambas, Q_3 é executada (em DF).

O resto deste exemplo discute como cada subconsulta local poderia ser processada utilizando-se as operações do SAR apresentadas na Seção 1.3. Considere Q_{21} primeiro. Esta subconsulta pode ser, por exemplo, processada por uma junção interativa das tabelas externas PRODUTO_RJ e FORNECIMENTO_RJ, tendo como predicado de junção P.CODIGO=F.CODIGO, como filtro para os registros de PRODUTO_RJ o predicado P.MELHOR-FORN = 41.738, e como filtro para os registros de FORNECIMENTO_RJ o predicado F.NUMERO = 41.738. É possível ainda usar pesquisa direta para acessar estas tabelas pois estão ambas invertidas por CODIGO, através das tabelas de inversão PRODUTO_RJ_COD e FORNECEDOR_RJ_COD, respectivamente. O resultado é produzido como uma tabela externa CODIGO_RJ a ser transferida para DF através de T_{21}.

Para processar Q_{22} só se poderá usar uma pesquisa seqüencial na tabela externa PRODUTO_RJ, pois não há uma inversão em MELHOR_PROD. O resultado é criado sob forma de uma tabela externa PRODUTO-MELHOR_RJ a ser remetida para SP via T_{22}.

Q23 poderá ser processada por uma junção interativa das tabelas externas PRODUTO-MELHOR-RJ e FORNECEDOR-SP, tendo como predicado de junção P.CODIGO = F.CODIGO. Os registros da tabela FORNECIMENTO-SP terão como filtro o predicado F.NUMERO = 41.738. A pesquisa em FORNECEDOR-SP poderá ser direta via a tabela de inversão FORNECEDOR-SP-COD, mas a pesquisa em PRODUTO-MELHOR_RJ terá que ser seqüencial, pois não há inversões sobre esta tabela recém-criada pela consulta Q_{22} e recebida através da transferência T_{22}.

Finalmente, Q_3 é processada através de uma operação UNIAO.

As decisões, ilustradas acima, sobre o desmembramento de uma consulta distribuída em subconsultas locais mais transferências de dados, bem como sobre a escolha das operações do SAR para processar consultas locais contituem o tópico dos dois próximos capítulos.

NOTAS BIBLIOGRÁFICAS

Apers [1983] trata explicitamente de processamento de consultas em bancos de dados distribuídos. Bracchi et all. [1980] e Spaccapietra [1980] discutem o problema genérico de processar consultas de alto nível em BDDs. Stonebraker [1980] contém algumas observações sobre como processar atualizações. A definição original de SQL encontra-se em Chamberlin, Astrahan, Eswaran e Lorie [1976]. O Capítulo 7 de Date [1981] explica as principais características de SQL. A definição do subsistema de armazenamento foi inspirada no

subsistema RSS do Sistema R, descrito brevemente em Astrahan et all. [1981] e em Blasgen et all. [1979]. O Capítulo 10 de Date [1981] também contém uma breve introdução ao RSS. Blasgen e Eswaran [1976] enumeram vários algorítmos para fazer junções, incluindo os dois descritos neste capítulo. Kim [1981] apresenta novos métodos para junção. Rosenthal e Reiner [1982] apresentam uma proposta bem estruturada para um otimizador de consultas, incluindo formas muito interessantes de realizar junções através de tabelas de inversão. Mais referências poderão ser encontradas nas notas bibliográficas dos Capítulos 4 e 5.

CAPÍTULO 4. PROCESSAMENTO DE COMANDOS DA LMD - O CASO CENTRALIZADO

Este capítulo aborda o problema do processamento de comandos da LMD para o caso de bancos de dados centralizados. Assume-se que a LMD e o subsistema de armazenamento são aqueles descritos no capítulo anterior e que o SGBD é homogêneo. A maior parte do capítulo é devotada ao processamento de consultas, enfatizando especialmente o problema de otimização. Atualizações serão tratadas apenas na última seção.

Toda discussão deste capítulo, naturalmente, também se aplica ao processamento local de comandos da LMD no caso de um SGBD distribuído.

4.1 INTRODUÇÃO AO PROCESSAMENTO DE CONSULTAS

Processamento de consultas em um banco de dados centralizado (ou processamento de consultas locais em um ambiente distribuído) corresponde a tradução de pedidos, formulados em uma linguagem de alto nível, para seqüências de operações elementares sobre os dados armazenados em um banco centralizado (ou em um banco local no caso de um SGBDD). Em termos dos esquemas que compõem a descrição de um banco de dados centralizado, podemos distinguir, então, três etapas:

1. Tradução para o Esquema Conceitual.
2. Tradução para o Esquema Interno.
3. Tradução para Acessos Físicos.

A etapa inicial, *tradução para o esquema conceitual*, transforma a consulta submetida pelo usuário, como formulada em termos do esquema externo a que ele tem acesso, em uma consulta equivalente, mas formulada em termos do esquema conceitual. A última etapa, *tradução para acessos físicos*, é de responsabilidade do subsistema de armazenamento (SAR), conforme discutido na Seção 3.3. Portanto, não será abordada neste capítulo.

A etapa intermediária, *tradução para o esquema int*erno, mapeia uma consulta, formulada agora em termos do esquema conceitual, para uma seqüência de operações do nível interno do SAR. Esta é a etapa principal do processo e, como em uma compilação tradicional, possui quatro fases distintas: análise sintática, otimização, geração de código e execução. A fase de análise sintática é a mais simples e não apresenta novidades. Após uma validação inicial da sintaxe do comando, informação acerca das relações referenciadas na consulta é obtida do catálogo. Uma nova validação é feita verificando-se se os atributos usados pertencem às relações, se as comparações respeitam os tipos de dados, etc. Esta fase não será abordada neste capítulo. A fase de otimização é a mais difícil e ocupará a maior parte deste capítulo. De fato, se a LMD é não-procedimental, uma consulta indica que dados devem ser recuperados, mas não como estes dados devem ser recuperados. Cabe, então, ao otimizador definir como, ou seja, selecionar uma seqüência de operações do subsistema de armazenamento que traduza corretamente a consulta e minimize uma dada função de custo. O resultado da fase de otimização será um plano de acesso aos dados, em pseudo-código, representando a seqüência de operações escolhida.

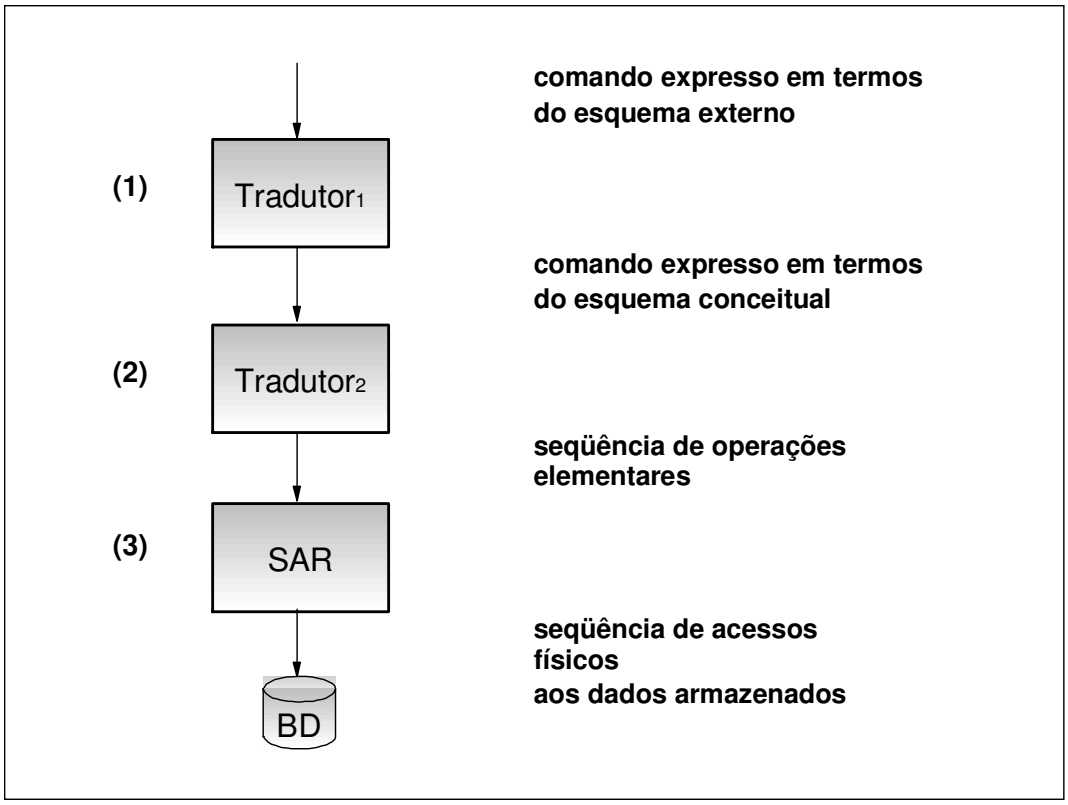

Figura 4.1 - Fases do Processamento de Consultas

As duas últimas fases, geração de código e execução, refletem uma de duas estratégias adotadas: interpretação ou compilação. Se a primeira estratégia for adotada, não há distinção entre estas duas fases: o pseudo-código gerado pelo otimizador é executado interpretativamente. Nesta estratégia, cada vez que uma consulta é re-submetida sofrerá todo o processo de tradução desde o início pois nenhum código é armazenado. Nas seções seguintes, esta será a estratégia adotada.

Uma consulta poderá, por outro lado, ser compilada da forma usual, armazenando-se o código gerado (correspondente à seqüência de operações escolhida pelo otimizador) para posterior execução. Desta forma, diminui-se o tempo de execução de comandos repetitivos, aumentando-se a performance dos programas de aplicação.

Há uma diferença fundamental, no entanto, entre a compilação de consultas de uma LMD e a compilação de programas usuais. Um programa, escrito em uma linguagem de programação normal, é mapeado em um conjunto de instruções que é fixo para a máquina onde o programa será executado. Analogamente, uma consulta da LMD é mapeada em operações sobre as estruturas de dados do banco, incluindo estruturas auxiliares (como Árvores-B). Mas tais estruturas podem mudar entre o tempo de compilação e o tempo de execução. Portanto, no caso de compilação de consultas de uma LMD, é necessário um processo de validação e, se necessário, recompilação antes da execução. A estratégia de compilação não mais será discutida neste capítulo.

Isto conclui a nossa breve introdução ao processamento de consultas centralizadas.

4.2 CLASSIFICAÇÃO DE CONSULTAS

O processo de tradução de consultas dependerá de certas definições e de uma breve classificação para consultas simples que serão introduzidas nesta seção.

A seguinte consulta sobre o banco de dados da Seção 3.2.2 será usada como exemplo:

```
                 select    F.NOME
                  from     FORNECEDOR F, FORNECIMENTO FN, REGIÃO R
P1               where  F.NUMERO = FN.NUMERO
P2                   and  (FN.CODIGO = 10 or FN.CODIGO = 12)
P3                 and  FN.QUANT > 10.000
P4=(P41 or P42)   and     (F.SEDE = R.ESTADO or F.SEDE = 'DF')
P5                   and    R.NOME = 'CENTRO SUL'
```

(P_1 indicará a primeira cláusula da qualificação, e assim por diante).

Seja *Q* uma consulta simples formulada em SQL. Seja $V=(V_1,...,V_n)$ a lista de variáveis de *Q*, *X* a lista resultante de *Q* e *B* a qualificação de *Q*. Assumiremos que:

> *B* está em forma normal conjuntiva, ou seja, *B* é da forma "C_1 *and ...* *and* C_m", onde C_i é uma disjunção de comparações negadas ou não.

C_i será chamada de uma *cláusula* de *B* e uma comparação, negada ou não, será chamada de um *literal.*

Esta suposição é interessante já que cada tupla da resposta da consulta *Q* deverá satisfazer a cada cláusula de *B*. Em particular, a qualificação da consulta usada como exemplo está em forma normal conjuntiva. As cláusulas da qualificação são P_1, P_2, P_3, P_4 e P_5.

Uma cláusula *C* é *univariável* se apenas uma variável da consulta ocorre nos literais de *C*, caso contrário, *C* é *multivariável*. *C* é *homogênea* se ou for univariável ou todos os seus literais forem junções, ou negações de junções, sobre as mesmas duas variáveis; caso contrário *C* é *heterogênea*.

Assim, as cláusulas P_2, P_3 e P_5 são univariáveis, a cláusula P_1 é homogênea e a cláusula P_4 é heterogênea, e ambas são multivariáveis.

Estendendo as definições acima, uma conjunção de cláusulas *B* é *univariável* se todas as suas cláusulas forem univariáveis sobre a mesma variável, caso contrário *B* é *multivariável*. B é *homogênea* se todas as suas cláusulas forem homogêneas e as mesmas duas variáveis ocorrem nas suas cláusulas (assim, se *B* for homogênea, poderá conter cláusulas univariáveis e multivariáveis homogêneas ao mesmo tempo).

Por exemplo, a conjunção de P_1, P_2 e P_3 é homogênea, mas a conjunção de P_4 e P_5 não o é (pois P_4 é heterogênea).

Uma consulta simples é *univariável, multivariável, homogênea, heterogênea*, se a sua qualificação for univariável, multivariável, homogênea, heterogênea.

4.3 TRADUÇÃO PARA O ESQUEMA CONCEITUAL

Esta etapa traduz uma consulta formulada em termos de um esquema externo para uma consulta equivalente formulada em termos do esquema conceitual. Na maior parte dos casos o processo é bastante simples, apresentando dificuldades apenas quando os esquemas usam modelos de dados diferentes.

Após uma análise sintática inicial, com o auxílio de informação contida no catálogo do SGBD, cada referência a uma estrutura externa é substituída pela sua definição em termos das estruturas do esquema conceitual. Em lugar de um algoritmo detalhado, apresentaremos um exemplo ilustrando o processo.

Considere o banco de dados usado como exemplo na Seção 3.2.1, e o esquema externo usado na Seção 3.2.4, cuja definição é repetida abaixo para facilidade de referência:

```
define view    PRODUTO_NESTLE ( CODIGO,NOME,MELHOR_FORN )
  as select    P.CODIGO, P.NOME, P.MELHOR_FORN
       from    PRODUTO P, FORNECIMENTO F
      where    F.NUMERO = '41.738'
        and    F.CODIGO = P.CODIGO
```

Considere agora a seguinte consulta sobre o esquema externo:

```
select    CODIGO
  from    PRODUTO_NESTLE
 where  MELHOR_FORN = '41.738'
```

Esta consulta é traduzida para o esquema conceitual substituindo-se PRODUTO_NESTLE pela sua definição, resultando em:

```
select    P.CODIGO
  from    PRODUTO P, FORNECIMENTO F
 where  F.NUMERO = '41.738'
   and    F.CODIGO = P.CODIGO
   and    P.MELHOR_FORN = '41.738'
```

Isto conclui a nossa breve discussão acerca da tradução de consultas sobre esquemas externos para consultas sobre o esquema conceitual.

4.4 TRADUÇÃO PARA O ESQUEMA INTERNO

Esta seção inicia o estudo da etapa de tradução de consultas formuladas em termos do esquema conceitual para operações do nível interno do SAR. Apenas o problema de se obter uma tradução correta será investigado, deixando a questão de otimização para as seções seguintes.

O problema de tradução será dividido em duas etapas (vide Figura 4.2):

1. Desmembramento: reduz consultas sobre o esquema conceitual a uma série de subconsultas homogêneas e operações de união que podem ser processadas diretamente pelo SAR;
2. Processamento de Consultas Homogêneas: seleciona seqüências de operações do nível interno do SAR que resolvem diretamente consultas homogêneas simples.

As subseções seguintes investigarão cada uma destas fases.

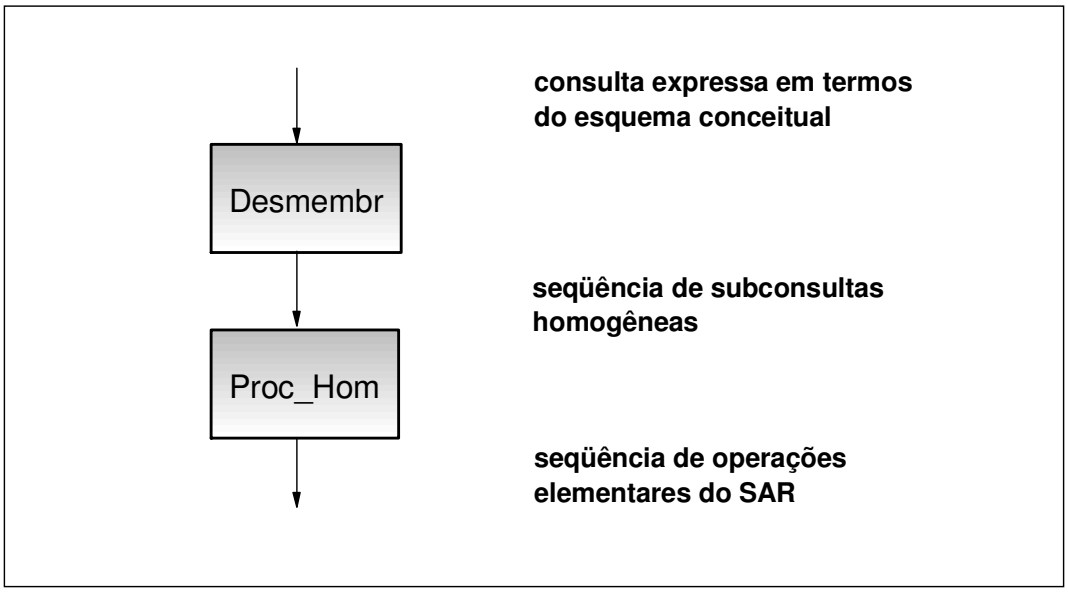

Figura 4.2 - Desmembramento de Consultas

4.4.1 Processamento de Consultas Homogêneas

As operações do nível interno do SAR apresentadas na Seção 3.3 são capazes de resolver diretamente uma consulta homogênea simples e uma união de relações, se fizermos certas suposições razoáveis.

Em primeiro lugar, ignoraremos o problema de duplicatas, que surge da diferença entre o conceito de tabelas do SAR, que podem conter registros duplicados, e o conceito de relações do modelo relacional (e, portanto, de SQL), onde a existência de tuplas duplicadas não faz sentido pois relações são conjuntos. Este problema poderia ser tratado introduzindo-se uma nova operação no SAR explicitamente para eliminar registros duplicados mas, por simplicidade, será abstraído neste capítulo.

Em segundo lugar, assumiremos por simplicidade que o esquema interno é tal que, a cada esquema de relação $R[A_1,...,A_n]$ definido no esquema conceitual, corresponderá uma única tabela externa, cujos registros representarão tuplas da relação corrente associada a *R*. Assumiremos ainda que esta tabela externa é descrita pelo esquema $R[IDR,A_1,...,A_n]$, onde o tipo de dados de *IDR* é IDR e este atributo corresponde ao campo contendo o identificador do registro. Assim, cada tabela externa que armazena uma relação herda o seu nome e os seus atributos.

Consideraremos consultas simples univariáveis e multivariáveis em separado, bem como uniões de relações.

(A) Uniões de Relações

Dentro das suposições acima, a união de duas relações *R* e *S* pode ser realizada diretamente pela operação *UNIAO(T,U;V)*, onde *T* e *U* são as tabelas armazenando as relações *R* e *S*.

(B) Consultas Univariáveis

Seja *Q* uma consulta simples univariável, *R* a relação varrida pela única variável *v* de *Q, X* a lista de atributos de *R* ocorrendo na lista resultante de *Q* e *B* a qualificação de *Q*.

Seja *T* a tabela armazenando *R*.

Inicialmente, observemos que os literais de *B* são formulados em termos da variável *v* e dos atributos da relação *R*. No entanto, como *T* possui todos os atributos desta relação, por construção, podemos criar uma expressão booleana *P* equivalente a *B* simplesmente substituindo cada ocorrência de *v* em *B* por *T*. A expressão *P* será então usada na definição das operações do SAR.

Há duas estratégias básicas para processar *Q:*

Estratégia $_1$:

Uma pesquisa sequencial *PESQ_SEQ(T,X,P;V)* resolverá *Q* diretamente.

T pode ser uma tabela externa, uma tabela interna ou uma tabela transiente. *V* será criada como uma tabela interna, se a sua cardinalidade for pequena, ou como uma tabela transiente em caso contrário. (Estimativa de cardinalidades será discutida na Seção 4.5.4). *V* poderá ainda ser criada como uma tabela externa se for a resposta da consulta, ou se necessitar ser movida para outro nó, no caso de consultas distribuídas.

Estratégia 2:

Suponha que haja uma tabela de inversão *E* sobre *T* em uma lista de atributos *Y*. Suponha que há um conjunto *P′* de cláusulas de *P* tais que todo atributo de *T* ocorrendo em um literal de *P′* também ocorre em *Y*. Então uma pesquisa direta *PESQ_DIR(T,X,P′′,E,P′;V)* resolverá a consulta, onde *P′′* é a conjunção das cláusulas de *P* que não estão em *P′*.

T neste caso será necessariamente uma tabela externa, pois este é o único tipo de tabela que nas estratégias consideradas possui inversões. *V* será passada como uma tabela interna ou uma tabela transiente, exatamente como na estratégia anterior.

À guisa de ilustração, apresentaremos aqui uma outra estratégia para processamento de consultas univariáveis. Como esta estratégia aumenta a complexidade do SAR e do otimizador, não constará do conjunto oficial de estratégias consideradas, no entanto.

Estratégia 3:

Suponha que *P* possua uma coleção

de cláusulas $P_1,...,P_k$ tais que P_i pode ser resolvida por uma pesquisa direta. Seja *P′* a conjunção das cláusulas restantes. Suponha que o SAR suporte um tipo de pesquisa direta em

T (via uma TINV U_i) que retorne apenas a lista L_i de identificadores dos registros de *T* que satisfazem a P_i.
Então a consulta *Q* será resolvida tomando-se a interseção de $L_1,...,L_k$, acessando-se apenas os registros de *T* cujos identificadores estão na interseção (pois estes satisfazem simultaneamente a $P_1,...,P_k$) e filtrando-se aqueles que não satisfizerem a *P'*.

Estratégias semelhantes, usando união de listas de identificadores existem para processar disjunções de cláusulas.

(C) Consultas Homogêneas Multivariáveis

Seja *Q* uma consulta simples homogênea e multivariável. Então há pelo menos uma cláusula da qualificação de *Q* onde ocorrem duas variáveis. Sejam *R* e *S* as relações varridas pelas duas variáveis de *Q*, *X* a lista de atributos de *R* e *Y* a lista de atributos de *S* na lista resultante de *Q*, e *B* a qualificação de *Q*. Sejam *T* e *U* as tabelas armazenando *R* e *S*, respectivamente. Como no caso anterior, seja *P* a expressão booleana equivalente a *B* mas formulada em termos de *T* e *U*.

Consideraremos as seguintes estratégias para processar *Q*:

Estratégia 1:

Uma junção interativa

JUNCAO_INTERATIVA(T,U,X,Y,P(T,U),P(T),P(U);V)

resolverá *Q* diretamente, onde *P(T,U)* é a conjunção de todas as cláusulas de *P* onde ocorrem tanto *T* quanto *U* (há pelo menos uma destas cláusulas), *P(T)* é a conjunção das cláusulas de *P* onde ocorre apenas *T*, e *P(U)* é a conjunção das cláusulas de *P* onde ocorre apenas *U* (tanto *P(T)* quanto *P(U)* podem ser vazias).

Esta estratégia se desdobra em quatro outras, dependendo da escolha de pesquisa seqüencial ou pesquisa direta para acesso a *T* e *U*.

Além disto, a ordem das tabelas pode ser trocada, considerando-se *U* como o primeiro argumento e *T* como o segundo argumento.

T e *U* poderão ser tabelas externas, tabelas internas ou tabelas transientes. *V* será criada como uma tabela interna, uma tabela transiente, ou uma tabela externa.

Estratégia 2:

Suponha que *P* seja da forma "*T.K<op>U.L and B*" e que *T* e *U* em ordem de junção por *<op>*. Então uma junção por intercalação

JUNCAO_INTERCALACAO(T,U,X,Y,P(T,U),P(T),P(U);V)

resolverá *Q* diretamente, onde os argumentos são como na estratégia anterior.

A estratégia se desdobra em outras, dependendo da escolha de pesquisa seqüencial ou direta para *T*, mas para *U* será sempre usada pesquisa seqüencial. De novo, a ordem das tabelas *T* e *U* poderá ser trocada, bem como a forma da tabela resultante *V*.
Estratégia 3:

Semelhante a anterior, exceto que *T* ou *U* não estão na ordem de junção . Neste caso, a junção é precedida da ordenação de *T* ou *U* (ou ambas) na ordem correta. *T* e *U* devem portanto ser tabelas externas ou tabelas internas.

Naturalmente outras estratégias existem para processar consultas homogêneas de duas variáveis, algumas usando listas de identificadores ou criação dinâmica de inversões. Mas as estratégias acima listadas serão as únicas consideradas neste texto.

4.4.2 Desmembramento de Consultas

Esta seção apresenta em linhas gerais o método de desmembramento de consultas, deixando para a seção seguinte os detalhes.

Nesta e nas próximas seções assumiremos novamente que a qualificação das consultas simples está em forma normal conjuntiva, ou seja, que a qualificação é da forma "C_1 *and ...* *and* C_m", onde C_i é uma disjunção de comparações, negadas ou não. Caso a qualificação da consulta já não esteja sob esta forma, ela deverá ser transformada através do método usual.

Uma *subconsulta* de uma consulta simples *Q* é simplesmente uma consulta cuja qualificação é a conjunção de um subconjunto das cláusulas da qualificação de *Q* e cuja lista de relações é uma sublista da lista de relações de *Q*. Uma *subconsulta própria* de *Q* é uma subconsulta de Q cuja qualificação não é idêntica a de *Q*.

Em linhas gerais, a estratégia para processar uma dada consulta *Q*, formulada em SQL, consiste em desmembrar *Q* em uma coleção *QC* de consultas homogêneas simples e uniões de relações, que poderão então ser mapeadas diretamente em operações do SAR. O processo de desmembramento é feito através de duas operações primitivas, *desmembramento por união e desmembramento por combinação.* O desmembramento por união decompõe *Q* em Q_1 e Q_2 de tal forma que a relação resultante de *Q* é idêntica à união das relações resultantes de Q_1 e Q_2. O desmembramento por combinação por sua vez decompõe *Q* em Q_1 e Q_2 de tal forma que a relação resultante de *Q* coincide com a relação resultante de Q_2, que por sua vez depende de Q_1.

O processo de desmembrar uma consulta *Q* em uma coleção *N* de uniões de relações e consultas homogêneas cria ainda uma ordem parcial sobre *N* tal que, para *Q'* e *Q"* em *N*, *Q'* precederá *Q"* sse *Q"* depender do resultado de *Q'* (ou seja, se *Q'* tiver que ser processada antes de *Q"/*). O resultado do desmembramento de uma consulta *Q* será então representado por um *grafo de desmembramento G =(N,E)*, onde *N* é a coleção de consultas homogêneas e uniões de relações em que *Q* se decompõe, e *(Q',Q")* está em *E* se *Q"* depender do resultado de *Q'*. Este grafo é acíclico e contém um sumidouro Q_n , que é a última operação a ser processada e cuja relação resultante deverá coincidir com a de *Q*. O desmembramento será

correto se a relação final produzida por Q_n for idêntica a de Q, para todo estado do banco de dados.

A seção seguinte apresenta os algoritmos que implementam o desmembramento, enquanto que o resto desta seção discute um exemplo de desdobramento.

Seja Q a seguinte consulta simples:

```
                select   F.NOME, P.CODIGO
                    into   RESULTADO
                  from   FORNECEDOR F, FORNECIMENTO FN, PRODUTO P
P1              where   P.NOME = 'BOMBOM'
P2                 and   P.CODIGO = FN.CODIGO
P3               and   FN.NUMERO = F.NUMERO
P4                 and   (FN.QUANTIDADE > 10000 or F.SEDE = 'SP')
```

Um possível desmembramento de Q seria o seguinte. Como P_4 é uma disjunção, Q pode ser substituída equivalentemente pela união Q_0.

Q_0 = RESULTADO = FORN_PROD1 union FORN_PROD2

das relações resultantes de duas consultas definidas da seguinte forma:

```
Q1 =            select   F.NOME, P.CODIGO
                    into   FORN_PROD1
                  from   FORNECEDOR F, FORNECIMENTO FN, PRODUTO P
                where   P.NOME = 'BOMBOM'
                    and  P.CODIGO = FN.CODIGO
                    and   FN.NUMERO = F.NUMERO
                    and   FN.QUANTIDADE > 10000

Q2 =            select   F.NOME, P.CODIGO
                    into   FORN_PROD2
                  from   FORNECEDOR F, FORNECIMENTO FN, PRODUTO P
                where   P.NOME = 'BOMBOM'
                    and   P.CODIGO = FN.CODIGO
                    and   FN.NUMERO = F.NUMERO
                    and   F.SEDE = 'SP'
```

O grafo de desmembramento neste ponto, portanto, é

$$G = (\{Q_0,Q_1,Q_2\},\{(Q_1,Q_0),(Q_2,Q_0)\})$$

indicando que Q_1 e Q_2 deverão ser processadas antes da união Q_0.

Q_1 por sua vez pode ser decomposta em uma combinação de duas consultas, Q_3 e Q_4:

```
Q3 =            select    F.NOME, FP.CODIGO
                    into    FORN_PROD1
                  from    FORNECEDOR F, FORN_PROD3 FP
                where   FP.NUMERO = F.NUMERO
                    and    FP.QUANTIDADE > 10000
Q4 =            select   FN.NUMERO, FN.QUANTIDADE, P.CODIGO
```

```
    into     FORN_PROD3
  from     FORNECIMENTO FN, PRODUTO P
where   P.NOME = 'BOMBOM'
   and     P.CODIGO = FN.CODIGO
```

É importante observar que o resultado de Q_1 é exatamente o mesmo de Q_3, que depende do resultado de Q_4. Note ainda que FORN_PROD3, o resultado de Q_4, contém toda informação, originaria de FORNECIMENTO e PRODUTO, que é necessária à execução de Q_3. Para acomodar o desmembramento de Q_1, o grafo de desmembramento deve ser modificado para

$$G = (\{Q_0, Q_2, Q_3, Q_4\} , \{ (Q_4 , Q_3) , (Q3, Q_0) , (Q_2, Q_0) \})$$

Similarmente Q_2 pode ser decomposta na combinação de Q_5 e Q_4, onde

Q_5 =
```
select   F.NOME, FP.CÓDIGO
    into     FORN_PROD2
  from     FORNECEDOR F, FORN_PROD3 FP
where     FP.NÚMERO = F.NÚMERO
   and     F.SEDE = 'SP'
```

O grafo final de desmembramento será então

$$G = (\{Q_0, Q_3, Q_4, Q_5\} , \{ (Q_4 , Q_3) , (Q_3, Q_0) , (Q_4, Q_5) , (Q_5, Q_0) \})$$

indicando que a primeira consulta a ser processada deverá ser Q_4, em seguida as consultas Q_3 e Q_5 deverão ser processadas em qualquer ordem, e a união Q_0 deverá ser a última operação.

*4.4.3 Algoritmos de Desmembramento

Os algoritmos, em pseudo-código, definindo as operações de desmembramento por união, desmembramento por combinação e o algoritmo definindo o processo de desmembramento completo encontram-se no final desta seção. O algoritmo de desmembramento é não determinístico e pode ser entendido como uma pesquisa arbitrária no espaço de todos os desmembramentos corretos de Q. Embora não seja imediato, pode-se provar por indução no número de operações de desmembramento por união e por combinação que, dada uma consulta Q, este algoritmo produz um desmembramento correto de Q.

No contexto destes algoritmos, uma *operação* é ou a união de duas relações ou uma consulta simples e sempre produzirá como resultado uma relação. Na descrição destes algoritmos, uma consulta simples será denotada como uma quádrupla $Q=(L,T,B,R)$ onde L é a lista de relações de Q, T é a lista resultante de Q, B é a qualificação de Q e R é o nome da relação resultante de Q. Uma união de duas relações, $R_0=R_1$ *union* R_2, será denotada por uma tripla (R_0 ,R_1 ,R_2), onde R_0 é a relação resultante e (R_1,R_2) é a lista de relações da operação. Diremos ainda que uma operação O *contribui* para uma operação O' se o nome da relação resultante de O ocorre na lista de relações de O'.

Além disto, o seguinte comando não determinístico será usada no algoritmo de desmembramento completo:

if $P_1 \rightarrow s_1$ ... $P_n \rightarrow s_n$ endif

Este comando deve ser interpretado da seguinte forma: escolha não deterministicamente uma ramo
' $P_i \rightarrow s_i$ ' tal que a condição P_i é satisfeita e execute o comando s_i.

Os algoritmos encontram-se abaixo.

```
DESMEMBRAMENTO_UNIÃO(Q;Q1,Q2,Q3)
 /*
   entrada:
    Q=(L,T,B,R) -  consulta simples a ser desmembrada por união;
                     B deverá possuir uma cláusula com mais de um literal
   saída  :
    Q1, Q2, Q3  -  subconsultas tais que o resultado de Q é equivalente a união dos
                   resultados de Q1 e Q2, expressa por Q3
 */
 begin
     selecione não deterministicamente uma cláusula arbitrária C de B
            com mais de um literal;
     particione não deterministicamente C em duas disjunções C1 e C2
            tais que cada uma tem pelo menos um literal;
     escolha nomes de relação temporárias não utilizados até o momento e
            coloque-os em R0, R1 e R2;
     B' :=  conjunção das cláusulas em B, exceto as em C;
     Q1 := (L,T,C1 & B',R1);
     Q2 := (L,T,C2 & B',R2);
     Q3 := (R0,R1,R2); /* união de R1 e R2 */
 end




 DESMEMBRAMENTO_COMBINAÇÃO(Q;Q1,Q2)
 /*
   entrada:  Q -   consulta simples a ser desmembrada por combinação ;
                   B deverá possuir mais de uma cláusula
   saída  :   Q1, Q2 - subconsultas tais que o resultado de Q é equivalente
                   a executar Q1 e depois Q2
 */
begin
     particione não deterministicamente B em duas conjunções B' e B"
            tais que B' e B" possuem pelo menos uma cláusula;
     escolha nomes de relações temporárias não utilizados até o momento e
            coloque-os em R1 e R2;
     L1 := sublista de L contendo todos os pares 'R r' tais que r ocorre em B';
     T1 := lista de todos os atributos qualificados A.r que ocorrem em T ou B"
            e tais que r é uma variável ocorrendo em L1;
     B1 := B';
     L2 := sublista de L contendo todos os pares 'R r' que não ocorrem em L1,
            mais o par 'R1 R1';
     T2 := lista resultante T onde cada atributo qualificado A.r tal que
            r é uma variável em L1 foi substituido por A.R1;
     B2 := conjunção B" onde cada atributo qualificado A.r tal que
            r é uma variável em L1 foi substituido por A.R1;
     Q1 := (L1,T1,B1,R1);
     Q2 := (L2,T2,B2,R2);
 end
```

```
DESMEMBRAMENTO(Q;G)
 /*
   entrada: Q - consulta a ser desmembrada
   saída  : G=(N,E) - um grafo de desmembramento para Q
 */
begin
      if Q é uma consulta simples
        then N := {Q}; E := ∅;
        else begin  /* Q é uma consulta-união */
                suponha que   Q    seja a união de Q1 e Q2,
                              onde   Q1=(L1,T1,B1,R1) e Q2=(L2,T2,B2,R2);
                seja R0 um novo nome de relação ;
                P := (R0,R1,R2);
                N := {P,Q1,Q2};
                E := {(Q1,P),(Q2,P)}
             end
      enquanto houver uma consulta simples não homogênea em N faça:
      begin
          selecione uma consulta P não homogênea em N;
          if
            há uma cláusula na qualificação de P com mais de um literal  →
            begin
                DESMEMBRAMENTO_UNIÃO(P,Q1,Q2,Q3);
                acrescente Q1, Q2, Q3 a N;
                remova P de N;
                acrescente (Q1,Q3) e (Q2,Q3) a E;
                foreach (P,Q') em E do
                begin remova (P,Q') de E;
                    acrescente (Q3,Q') a E;
                end
            end
          //
            há mais de uma cláusula na qualificação de P →
            begin
                DESMEMBRAMENTO_COMBINAÇÃO(Q,Q1,Q2);
                acrescente Q1 e Q2 a N;
                remova P de N;
                acrescente (Q1,Q2) a E;
                foreach (P,Q') em E do
                begin remova (P,Q') de E;
                    acrescente (Q2,Q') a E;
                end
            end
          endif
          foreach (Q',P) em E do
          begin remova (Q',P) de E;
              if Q' contribui para Q1 then acrescente (Q',Q1) a E;
              if Q' contribui para Q2 then acrescente (Q',Q2) a E;
          end
      end
 end
```

4.4.4 Descrição Final da Solução

Seja *Q* uma consulta. O grafo de desmembramento *G=(N,E)* resultante do processo de desmembrar *Q* indica uma coleção (*N*) de consultas homogêneas e união de relações capazes de produzir o resultado da consulta, e a ordem (parcial) em que as consultas e uniões deverão ser processadas (pois *G*, por construção, e acíclico).

O objetivo final, porém, é obter uma seqüência *S* de operações do nível interno do SAR que resulte em uma tabela contendo o resultado de *Q*. Mas este passo final é simples, e consiste de:

1. Substitua cada operação *O* em *N* por uma seqüência *S(O)* de operações (ou por apenas uma operação) do nível interno do SAR de acordo com as estratégias discutidas na Seção 4.4.1.
2. Produza uma ordem linear *S* das operações em *N* compatível com a ordem parcial representada por *G*.
3. *S* será então a seqüência procurada.

Note que *S* representa um programa sequencial. Se, por outro lado, for possível execução concorrente de operações, o grafo *G* indicará então como as operações poderão ser despachadas.

4.5 OTIMIZAÇÃO

O algoritmo de desmembramento conforme apresentado na seção anterior, produz uma desmembramento correto de uma consulta. No entanto, o desmembramento obtido é completamente arbitrário, e não leva em consideração a estrutura da qualificação da consulta e informação acerca da cardinalidade das tabelas, inversões existentes e, principalmente, as estratégias para processar uma consulta homogênea. Por outro lado, o algoritmo tem o mérito de definir de forma concisa o espaço de todos os desmembramentos corretos para uma consulta.

O problema imediatamente seguinte consiste, então, em construir um desmembramento correto para uma consulta e escolher uma estratégia de processamento para cada consulta homogênea do desmembramento de tal forma a minimizar uma dada função de custo. Este é essencialmente um problema combinatorial de otimização e como tal será tratado. A primeira solução a ser discutida consiste em uma pesquisa exaustiva no espaço de todas as soluções possíveis para localizar a de menor custo. Uma segunda solução será em seguida apresentada baseando-se em uma heurística para limitar a pesquisa. Por fim, o problema de estimar o custo de uma solução, que por si só é interessante, será abordado.

4.5.1 Pesquisa Exaustiva da Solução Ótima

A forma mais simples de otimizar uma consulta resume-se a uma pesquisa exaustiva no espaço de todas as possíveis soluções do problema para localizar a de menor custo. Uma solução neste contexto pode ser descrita em dois níveis: um desmembramento correto da consulta e,

para cada operação (consulta homogênea ou união de relações) resultante do desmembramento, uma estratégia para processamento da operação. Resumidamente, este procedimento consiste, então, de:

1. enumerar todos os possíveis desmembramentos de uma consulta (primeiro nível da busca);
2. para cada desmembramento possível, enumerar todas as possíveis estratégias para processar cada consulta homogênea ou união de relações gerada pelo desmembramento (segundo nível da busca);
3. estimar o custo de cada solução;
4. escolher a solução de menor custo.

Dada a natureza do problema, técnicas de "branch-and-bound" ou "backtracking" com funções limitantes aplicam-se na pesquisa da solução ótima.

4.5.2 Desmembramento Controlado - Parte I

O número total de possíveis formas de processar uma consulta cresce exponencialmente com o número de cláusulas da qualificação. Uma pesquisa exaustiva para selecionar a melhor solução pode então se tornar proibitiva, especialmente para consultas imediatas que não serão repetidas novamente. É interessante definir portanto uma heurística que limite a busca da solução ótima.

Por outro lado, tal heurística deve produzir uma solução que, embora possa não ser ótima, pelo menos seja razoável.

Descartaremos como "não razoável" qualquer solução que leve desnecessariamente a um crescimento exponencial dos resultados intermediários. Tal situação é conseqüência da formação implícita do produto cartesiano de várias relações (que em certos casos, porém, é exigido para resolver um conjunto de cláusulas homogêneas de duas variáveis). Esta posição justifica-se pois o custo de processar uma consulta e o seu tempo de resposta estão diretamente ligados ao tamanho dos resultados intermediários.

Nesta seção será apresentada então uma heurística que, baseada em uma análise da qualificação da consulta, procura uma solução que evita o crescimento exponencial desnecessário dos resultados intermediários e, ao mesmo tempo, evita uma pesquisa exaustiva de todas as formas possíveis de processar a consulta. Chamaremos esta heurística de *desmembramento controlado*. Ela se processa da seguinte forma:

1. desmembre a consulta original em suas *componentes irredutíveis* (que levam necessariamente à formação de produtos cartesianos implícitos);
2. ordene as componentes irredutíveis de tal forma que as componentes correspondendo a subconsultas homogêneas sejam executadas primeiro e a componente contendo a lista resultante por último (isto reduz o tamanho dos resultados intermediários);
3. otimize o processamento de cada componente irredutível em separado, ignorando as outras componentes. (Como as componentes são mais simples do que a consulta original, exceto se a consulta já for irredutível, o problema de otimizá-las é mais simples que o problema de otimização global. Por outro lado, não convém otimizar uma consulta mais

simples do que uma componente irredutível pois corre-se o risco de produzir resultados intermediários muito maiores do que o necessário).

Para avaliar o efeito do desmembramento controlado, considere o seguinte exemplo. Seja *Q* a consulta abaixo sobre o banco de dados da Seção 3.2.1:

```
            select   P.MELHOR_FORN, P.NOME, FN.QUANTIDADE
             from    FORNECEDOR F, FORNECIMENTO FN, PRODUTO P, REGIÃO R
P1          where    R.NOME = 'CENTRO-SUL'
P2            and    F.SEDE = R.ESTADO
P3            and    P.MELHOR_FORN = F.NOME
P4            and    P.CODIGO = FN.CODIGO
P5            and    F.NUMERO = FN.NUMERO
```

(Quais os nomes dos fornecedores, os nomes dos produtos e as quantidades fornecidas tais que o fornecedor está sediado em um estado da região Centro-Sul, fornece o produto e é considerado o melhor fornecedor do produto).

Esta consulta é interessante pois possui quatro cláusulas de duas variáveis (P_2 a P_5). No entanto, há certas diferenças fundamentais em como estas cláusulas podem ser tratadas. Em primeiro lugar, a cláusula P_2, juntamente com P_1, age no sentido de restringir a relação FORNECEDOR.Além disto, apenas a variável F varrendo FORNECEDOR existe em comum entre P_2 e P_3, P_4 e P_5. Logo é possível otimizar a subconsulta gerada por P_1 e P_2 em separado sem correr o risco de criar um resultado intermediário demasiadamente grande. O seu resultado será necessariamente um subconjunto de FORNECEDOR e conterá toda informação necessária ao resto da consulta.

Por outro lado, as três últimas cláusulas, P_3, P_4 e P_5, formam uma *componente irredutível* no sentido de que, para resolver uma destas cláusulas, é necessário fazer uma junção de duas relações e manter informação sobre ambas para resolver as outras cláusulas e criar o resultado da consulta. Ou seja, não é possível reduzir o seu processamento a subconsultas independentes mais simples. Por exemplo, se for feita a junção de FORNECEDOR com PRODUTO para resolver P_3, é necessário manter CODIGO de PRODUTO e NUMERO de FORNECEDOR para que possam ser resolvidas as cláusulas P_4 e P_5. Além disto, é preciso manter NOME e MELHOR_FORN de PRODUTO para criação das tuplas resultantes. Estes blocos de cláusulas criam resultados intermediários potencialmente explosivos e devem ser otimizados exaustivamente.

Em resumo, uma boa estratégia para processar *Q* consiste em

1. desmembrar *Q* em Q_1 e Q_2, que são geradas pelos conjuntos de cláusulas $\{P_1,P_2\}$ e $\{P_3,P_4,P_5\}$, respectivamente;
2. otimizar Q_1 e Q_2 separadamente, obtendo o desmembramento ótimo destas subconsultas por pesquisa exustiva.

Desta forma estaremos reduzindo o trabalho de otimização pois Q_1 e Q_2 são mais simples do que *Q*. Por outro lado, como Q_1 e Q_2 só tem uma variável em comum (a variável F), não há perigo de gerarmos um resultado intermediário desnecessariamente grande.

Estas observações formam a base intuitiva do desmembramento controlado.

*4.5.3 Desmembramento Controlado - Parte II

Para tornar a discussão sobre desmembramento controlado mais precisa necessitamos das seguintes definições. Seja Q uma consulta. Q é *disconexa* se a sua qualificação puder ser particionada em duas conjunções de cláusulas sem nenhuma variável em comum. Q é *conexa* se não for disconexa. Se Q for disconexa, é possível desmembrá-la em duas subconsultas, Q_1 e Q_2, sem nenhuma variável em comum. Assim, Q_2 é totalmente independente de Q_1, embora ambas sejam necessárias pois se um dos resultados for vazio, assim será o de Q. Uma *variável de junção* de Q é uma variável v tal que a qualificação de Q pode ser particionada em duas conjunções de cláusulas com apenas v em comum. Neste caso, Q pode ser desmembrada na composição de uma subconsulta Q_1 com uma subconsulta Q_2 tais que Q_1 e Q_2 têm apenas a variável v em comum. Uma consulta Q é *irredutível* se for conexa e não possuir uma variável de junção.

Uma subconsulta Q' de Q é *gerada* por um subconjunto $B'=\{C_1, ..., C_n\}$ das cláusulas da qualificação de Q se a qualificação de Q' for a conjunção das cláusulas em B'. Uma consulta Q' é uma *subconsulta* ou *componente irredutível* de Q se Q' for uma subconsulta de Q, Q' for irredutível e não existir uma subconsulta Q'' de Q tal que Q'' é irredutível e Q' é uma subconsulta própria de Q''. Uma subconsulta irredutível possui uma qualificação cujas cláusulas formam um bloco semelhante àquele formado por P_3, P_4 e P_5 no exemplo da seção anterior.

Voltemos agora ao desmembramento controlado. A heurística em questão propõe desmembrar uma consulta em subconsultas irredutíveis, inicialmente, ignorando as subconsultas univariáveis que sempre serão incorporadas a outras. O grafo de desmembramento, T, será uma floresta após este desmembramento inicial pois, caso contrário, uma das consultas não seria irredutível (o leitor deve se convencer deste ponto usando a definição de subconsulta irredutível, ou ler a caracterização mais precisa da heurística apresentada abaixo).

Em seguida T é percorrida das folhas para as raízes. Para cada subconsulta Q' visitada, se Q' não for homogênea, um desmembramento ótimo é obtido por pesquisa exautiva. Q' é então substituída pelo grafo de desmembramento resultante do processo. O resultado final é um desmembramento de Q em subconsultas homogêneas.

Como pode haver mais de uma possível ordenação das componentes irredutíveis sob forma de uma floresta T para uma dada consulta, a heurística propõe ainda escolher uma que force subconsultas homogêneas a serem processadas primeiro, e deixe a componente que contém a lista resultante para o fim.

A consulta tomada como exemplo na seção anterior é conexa e possui uma variável de junção, F. O desmembramento inicial de Q em componentes irredutíveis resulta em três consultas Q_{11}, Q_{12} e Q_2, geradas pelos conjuntos de cláusulas $\{P_1\}$, $\{P_2\}$, e $\{P_3,P_4,P_5\}$, respectivamente. Como Q_{11} é univariável, a heurística dita incorporá-la a Q_{12}, obtendo a consulta Q_1. Ou seja, obtemos as mesmas subconsultas de Q encontradas no exemplo da seção anterior. Como Q_1 é homogênea e, além disto, Q_2 contém a lista resultante, a heurística recomenda processar Q_1 antes de Q_2. Isto significa que o desmembramento inicial é uma árvore cujos nós são Q_1 e Q_2 e é tal que Q_2 é o pai de Q_1.

O próximo passo da heurística é otimizar Q_1 e Q_2 em separado. Como Q_1 é homogênea, não é necessário desmembrá-la. Mas para Q_2 é necessário obter um desmembramento ótimo por pesquisa exaustiva. Digamos que o resultado seja um desmembramento de Q_2 na combinação de Q_{22} com Q_{23}, onde Q_{22} e Q_{23} são geradas pelos conjuntos de cláusulas $\{P_3\}$ e $\{P_4, P_5\}$.

O grafo final de desdobramento seria então uma árvore com três nós, Q_1, Q_{22} e Q_{23}, onde Q_{23} é a raíz, Q_{22} o filho de Q_{23} e Q_1 o filho de Q_{22}.

A análise da qualificação da consulta necessária à heurística pode ser feita representando-se a qualificação da consulta através de um hipergrafo. Um hipergrafo é um par ordenado $H = (N,E)$ onde N é um conjunto não vazio de nós e E é um conjunto de subconjuntos finitos de nós, chamados de hiper-arestas. No caso de todos os conjuntos em E conterem exatamente dois nós, o hipergrafo se reduz a um grafo não-dirigido. As noções de caminho, componente biconexa e ponto de articulação em hipergrafos são definidas como para grafos. (Se o leitor assim o desejar, poderá raciocinar em termos de grafos no que se segue. O exemplo adotado não necessita da generalidade de hipergrafos para ser analisado).

O hipergrafo $H(Q) = (N,E)$ de uma consulta Q é construído a partir da qualificação e da lista resultante da seguinte forma: o conjunto de nós N consiste de todas as variáveis da consulta; para cada cláusula C da qualificação (ou lista resultante T) acrescente a E, se já não houver, uma hiper-aresta consistindo de todas as variáveis ocorrendo em C (ou em T).

É possível mostrar que as subconsultas irredutíveis da consulta são geradas pelas cláusulas correspondendo às hiper-arestas das componentes biconexas do hipergrafo da consulta e as variáveis de junção da consulta são os pontos de articulação do hipergrafo. Portanto, para se determinar as subconsultas irredutíveis da consulta, basta se determinar as componentes biconexas do grafo (ou os seus pontos de articulação).

$$E = \{\{R\},\{F,R\},\{P,F\},\{P,FN\},\{F,PN\}\}$$

Note que a hiper-aresta $\{P,FN\}$ representa tanto a lista resultante quanto a cláusula P_4.

A heurística é apresentada de forma mais precisa através do algoritmo (em pseudo-código) DESMEMBRAMENTO_CONTROLADO. Este algoritmo usa um outro, REDUÇÃO, que determina as subconsultas irredutíveis da consulta e as ordena de forma apropriada, combinando as subconsultas irredutíveis univariáveis com outras. Estes algoritmos são apresentados a seguir.

```
DESMEMBRAMENTO_CONTROLADO(Q;G)
/*
  entrada: Q uma consulta a ser desmembrada
  saída  : G um desmembramento correto e otimizado de Q
*/
begin
    /*
      criação de um desmembramento parcial G de Q em subconsultas irredutíveis,
       onde as subconsultas univariáveis foram incorporadas a outras.
      G será sempre uma floresta.
    */
    REDUÇÃO(Q;G);
```

```
    /*
      substituição de cada subconsulta irredutível Q' em G por um desmembramento de Q',
       produzindo o desmembramento final
    */
    foreach nó P de G em ordem pós-fixa tal que P não é homogênea do
    begin
        ache o desmembramento ótimo de P por pesquisa exaustiva;
        faça G' igual ao grafo do desmembramento;
        faça Q' igual à operação final de G';
        /*
          substitua P por G' em G
        */
        G := G union G';
        foreach (P',P) em G do
        begin remova (P',P) de G;
            foreach P" em G' do
            if P' contribui para P"
              then adicione (P',P"') a G;
        end
        foreach (P,P') em G do
        begin remova (P,P') de G;
            adicione (Q',P') a G;
        end
        remova P de G;
    end
end

REDUÇÃO(Q;G)
/*
  entrada: uma consulta Q
  saída  :  um desmembramento G de Q em subconsultas irredutíveis onde as subconsultas
              univariáveis foram incorporadas a outras; G será sempre uma floresta.
*/
begin
    PRE_ORDENAÇÃO(Q;F);
    TRANSFORMAÇÃO1(F;G);
    TRANSFORMAÇÃO2(G);
end

PRE_ORDENAÇÃO(Q;F)
/*
  pre-ordenação das componentes biconexas de H(Q)
  entrada: consulta Q
  saída  :  floresta rotulada F=(M,B,r) onde cada nó n é rotulado com um conjunto de
              cláusulas de Q
*/
begin
    /*
      organização das componentes em um grafo
    */
    construa o hipergrafo H(Q) de Q;
    determine as componentes biconexas de H(Q);
    construa um grafo rotulado não dirigido H=(M,A,r) tal que
    (1)  para cada componente biconexa C de H(Q) existe um nó n de H
         (diz-se que C gera n);
    (2)  r(n) é o conj. de cláusulas ou lista resultante de Q correspondendo às arestas de C;
    (3)  (n,m) ∈ A se e somente se as componentes C e D
         que geram n e m tem um ponto de articulação em comum.
    /*
       ordenação das componentes em uma floresta
```

```
    */
    construa uma floresta rotulada F=(M,B,r) tal que:
    (1)  F é uma floresta geradora de H (logo M e r são herdados de H);
    (2)  o nó n cujo rótulo contém a lista resultante é uma raíz de F;
    (3)  os nós rotulados com apenas uma cláusula univariável são sempre folhas;
    (4)  os nós rotulados com um conjunto de cláusulas homogêneas devem estar o mais
         próximo possível das folhas.
    /*
         compactação das componentes univariáveis
    */
    foreach folha n de F do
      if n é rotulada com apenas uma cláusula univariável
         then begin
                  acrescente a cláusula ao conjunto que rotula o pai de n em F;
                  remova n de F
              end
end


TRANSFORMAÇÃO1(F;G)
/*
  Criação de um desmembramento parcial de Q - Fase I
  entrada: floresta F=(M,B,r) rotulada com conjuntos de cláusulas
  saída  :  floresta G=(N,E) representando um desmembramento parcial de Q
*/
begin N=∅; E=∅;
    /*
      criação das subconsultas do desmembramento parcial a menos das listas resultantes
    */
    foreach nó n de F em pré-ordem do
    begin
        gere a partir de n uma consulta P da seguinte forma:
        (a) a relação resultante de P é um novo nome de relação;
        (b) a qualificação B de P é a conjunção das cláusulas que rotulam n;
        (c) a lista de relações de P contém todas as variáveis referenciadas em B
              junto com as relações que elas varrem;
        (d) a lista resultante de P coincide com a de Q,
            se o rotulo de n contiver a lista resultante de Q;
              caso contrário, a lista resultante de P permanece indefinida.
        adicione P a N;
        if n' é o pai de n em F e P' a consulta gerada por n'
        then adicione o arco (P,P') a E;
    end
end


TRANSFORMAÇÃO2(G)
/*
  Criação de um desmembramento parcial de Q - Fase II
  entrada e saída  : floresta G=(N,E) representando um desmembramento parcial de Q
*/
begin
    /*
      criação do desmembramento parcial final
    */
    foreach consulta P de G em pós-ordem do
    begin
        /*
          criação da lista resultante final
```

```
        */
        acrescente a lista resultante de P cada atributo de cada  relação  varrida em P
             que é referenciada em um ancestral de P em G;
        /*
           modificação dos ancestrais de P em G para que G
            seja um desmembramento correto
        */
        foreach ancestral P' de P em G do
        begin  /*
                    substitua as relações varridas em P pela relação resultante de P
                */
                elimine todos os pares 'R r' da lista resultante de P' onde R é uma das
                    tabelas varridas em P;
                adicione um par 'RP p' onde RP é a relação resultante de P;

                  /*
                    modifique a qualificação de P para
                      refletir mudanças na lista resultante
                */
                substitua todas as ocorrências de atributos qualificados 'r.A',
                      onde r foi eliminada, por 'p.A'
        end;
    end;
end
```

*4.5.4 Estimativa de Custo

Independentemente do método de otimização usado, pesquisa exaustiva ou pesquisa guiada por uma heurística, é necessário estimar o custo de cada possível estratégia para processar uma consulta. A estimativa de custo pode ser reduzida a duas tarefas básicas:

1. estimar o custo das operações do SAR usadas para processar consultas homogêneas e uniões de relações;
2. estimar como o resultado de uma consulta ou união afeta o custo das consultas ou uniões posteriores no desmembramento.

De fato, lembremos que uma estratégia compõe-se de um desmembramento da consulta seguido da especificação de um método para processar cada consulta homogênea e união de relações do desmembramento. Logo, o custo de uma estratégia é, então, a somatória dos custos das consultas homogêneas e uniões em que resulta o desmembramento, o que nos leva à primeira tarefa. Além disto, uma consulta ou união poderá usar o resultado de outra, o que implica por sua vez na segunda tarefa.

Discutiremos cada um destes problemas em separado no que se segue.

Assumiremos que o esquema interno é definido de forma simples, ou seja, que cada relação é armazenada em uma tabela externa diferente, que por sua vez ocupa um segmento distinto.

Assumiremos ainda que as seguintes estatísticas sobre o banco de dados são mantidas pelo sistema:

- para cada tabela externa *T*:

CAR(T) - número de registros de *T*;
PAG(T) - número de páginas ocupados por *T*;

- para cada atributo *A* de *T*:

  CMX(T,A) - comprimento máximo do campo correspondente a *A* em *T*;
  CMN(T,A) - comprimento mínimo do campo correspondente a *A* em *T*;
  MAX(T,A) - maior valor de *T[A]* ocorrendo no banco, se o domínio de *A* for aritmético;
  MIN(T,A) - menor valor de *T[A]* ocorrendo no banco, se o domínio de *A* for aritmético;

- para cada tabela de inversão *U*:

  CHV(U) - número de chaves distintas em *U*;
  PAG(U) - número de páginas contendo chaves em *U*;

Estas estatísticas são mantidas no catálago do sistema junto com a definição do banco e, para não sobrecarregar o sistema, são atualizadas de forma periódica.

(A) Influência do Resultado de uma Operação sobre Outra

Seja *Q* uma consulta ou união de relações. Suponhamos que a relação resultante de *Q* seja armazenada em uma tabela *T*. Como *T* poderá ser varrida em outras consultas, desejamos estimar as estatísticas de *T* para que seja possível tratar *T* como qualquer outra tabela do banco de dados.

Se *Q* é uma união de relações, a estimativa das estatísticas é simples e não será discutida. Portanto assuma que *Q* é uma consulta simples. (A discussão a seguir se aplica a qualquer consulta simples, embora não necessitemos destes resultados para consultas homogêneas).

Inicialmente, observemos que *MAX(T,A)*, *MIN(T,A)*, *CMX(T,A), CMN(T,A)* são facilmente computados a partir destes mesmos parâmetros mantidos para as tabelas de dados do banco e a partir da qualificação da consulta (apenas necessária para *MAX(T,A)* e *MIN(T,A)*, pois pode haver comparações na qualificação limitando estes parâmetros). Da mesma forma, *PAG(T)* é facilmente estimado conhecendo-se *CAR(T)*. De fato, como são conhecidos os comprimentos dos campos de cada atributo de cada tabela do banco, sempre é possível estimar os comprimentos máximo e mínimo, *CMAX* e *CMIN*, dos registros de *T* (se os campos forem de comprimento fixo, *CMAX = CMIN*, naturalmente). Logo, conhecendo-se o comprimento das páginas, *CPAG*, teremos:

$$CAR(T) * CMIN / CPAG < PAG(T) < CAR(T) * CMAX / CPAG$$

Portanto, ficamos reduzidos à estimação de *CAR(T)*.

Sejam $r_1 \ldots r_n$ as variáveis de *Q* e $U_1 \ldots U_n$ as tabelas varridas por $r_1 \ldots r_n$. Seja *T* a tabela resultante de *Q* e *B* a qualificação de *Q*. Então, por definição, *T* será o subconjunto do produto cartesiano $U = U_1 \times \ldots \times U_n$ definido por *B*. Seja *F(B)* o *fator de seletividade* de *B*, ou seja, a percentagem de tuplas de *U* que satisfazem *B*. Então temos que

$$CAR(T) = CAR(U_1) \times \ldots \times CAR(Un) \times F(B)$$

Basta portanto estimar *F(B).* No entanto, uma estimativa precisa de *F(B)* requer um estudo minucioso que está além do escopo desta seção. Estimativas imediatas, que mostraram ser

eficazes em um sistema real, são descritas abaixo por tipo de cláusula (*i* e *j* varrem o intervalo [1,*n*]):

B é do tipo $r_i .A = "valor"$

$F(B) = 1 / CHV(V)$, se existe uma tabela de inversão *V* invertendo U_i por *A* (assume-se uma distribuição uniforme de tuplas pelos valores de chave).

$F(B) = 1/10$, em caso contrário.

Nota: Não há qualquer significado maior na escolha de *1/10*, exceto a hipótese de que dificilmente uma consulta selecionará mais do que *1/10* das tuplas de uma relação.

B é do tipo $r_i .A > "valor"$ (e semelhantemente para outras comparações, exceto igualdade)

$F(B) = (MAX(U_i , A) - "valor")/(MAX(U_i , A) - MIN(U_i , A))$

$F(B)$ é obtido por interpolação linear do valor no intervalo de valores existentes, se o domínio de *A* for aritmético;

$F(B) = 1/10$, se o domínio de *A* não for aritmético;

B é do tipo $r_i .A = r_j .B$

$F(B) = 1 / MAX(CHV(V_1),CHV(V_2))$,
se existem tabelas de inversão V_1 e V_2 invertendo U_1 por *A* e U_j por *B*, respectivamente (assume-se que cada valor de chave na tabela de inversão com menor cardinalidade ocorre também na tabela de maior cardinalidade).

$F(B) = 1 / CHV(V)$,
se existe uma tabela de inversão *V* invertendo U_1 por *A* ou U_j por *B*.

$F(B) = 1/10$, em caso contrário

B é do tipo $r_i .A > r_j .B$ (e semelhantemente para outras comparações, exceto igualdade)

$F(B) = (MAX(U_i , A) - MIN(U_j , B))/(MAX(U_i , A) - MIN(U_i , A))$,
se o domínio de *A* for aritmético;

$F(B) = 1/10$, se o domínio de *A* não for aritmético;

$B = B_1 \ OR \ B_2$.

$F(B) = F(B_1) + F(B_2) - F(B_1) * F(B_2)$

$B = B_1 \ AND \ B_2$

$F(B) = F(B_1) * F(B_2)$ (Assume-se independência de eventos neste caso).

$B = \neg B_1$

$F(B) = 1 - F(B_1)$

(B) Estimativa do Custo de Processar Consultas Homogêneas e Uniões de Relações

Estimar o custo de processar consultas homogêneas e uniões de relações degenera na estimativa do custo das operações do SAR, tendo em vista as estratégias apresentadas na

Seção 4.4.1 para tratamento de consultas homogêneas e uniões de relações. Portanto esta subseção se atém ao problema de estimar o custo das operações do SAR.

A equação de custo para uma operação genérica *P* terá sempre o seguinte formato:

C(P) = número de páginas lidas para obter os operandos
+ número de páginas gravadas para produzir o resultado
+ *W* * tempo de CPU gasto para produzir tuplas do resultado

onde *W* é um fator de ponderação entre tempo de CPU e atividade de leitura/gravação de páginas. O tempo de CPU em todos os casos será uma função de uma cardinalidade facilmente estimada a partir dos resultados da parte (A) desta seção. O número de páginas lidas ou gravadas depende da forma como os argumentos são passados ou o resultado é gerado. Se for através de uma tabela de dados, este número será estimado a partir do número de páginas ocupadas pela tabela; se for através da uma tabela interna ou de uma tabela transiente, então este número será zero.

As seguintes observações simplificarão a apresentação das equações. Em primeiro lugar, para cada operação seria necessário apresentar uma equação diferente para cada combinação possível de opção de passagem de argumentos e geração do resultado. Isto pode ser evitado, no entanto, definindo-se simplesmente um novo parâmetro, *PAG'(T)*, da seguinte forma:

PAG'(T) = *PAG(T)* , se *T* for passada ou gerada como uma tabela de dados
= *0* , caso contrário

Em certos casos será necessário estimar *PAG(T)*, mas nos limitaremos a estimar *CAR(T)* a partir de fatores de seletividade, já que *PAG(T)* pode ser computado de *CAR(V)* e de outros parâmetros, conforme a discussão da parte (A).

1*) ORD(T,X,tipo;V)*

Intuitivamente, o custo desta operação será:

C = número de páginas de *T* lidas
+ número de páginas de *V* gravadas
+ número de páginas lidas e gravadas para ordenação
+ *W* * tempo de CPU para ordenação

Assuma que *T* é uma tabela de dados e que o algoritmo de ordenação externa por intercalação ("external sort-merge") é usado. Assuma ainda que a intercalação envolve três arquivos de cada vez e que o tempo de CPU da ordenação é desprezível em presença do tempo de entrada e saída.

Como *V* = *T*, as duas primeiras parcelas do lado direito da equação de custo anterior são iguais. A terceira parcela é dada pelo número de páginas lidas egravadas pelo algoritmo de ordenação (veja qualquer livro de estruturas de dados). A quarta parcela é nula em vista das suposições anteriores.

Logo, teremos então:

$C = 2 * PAG(T) + 2 * PAG(T) * log(PAG(T))$

onde log é tomado na base 3 em vista das suposições.

Assuma agora que *T* está em memória principal (logo o custo de lê-la é nulo), mas que *V* poderá ou não ser gerada em memória principal. Assuma que nestas condições a ordenação é feita em memória principal. O custo será então:

$C = 2 * PAG'(V) + W * CAR(T) * log(CAR(T))$

onde *W* é uma constante de proporcionalidade entre tempo de CPU (para ordenação) e tempo de gravação (se *V* for gerada em memória secundária).

2) *UNIÃO(T,U;V)*

O custo será:

C = número de páginas de *T* lidas
+ número de páginas de *U* lidas
+ número de páginas de *V* gravadas
= $PAG'(T) + PAG'(U) + PAG'(V)$

onde $PAG'(V)$ é obtido de $PAG(V)$, que por sua vez é simplesmente:

$PAG(V) = PAG(T) + PAG(U)$

3) *PESQ_SEG(T,X,P(T);V)*

O custo será:

C = número de páginas de T lidas
+ número de páginas de V gravadas
+ *W* * número de tuplas de T processadas
= $PAG'(T) + PAG'(V) + W * CAR(T)$

onde usamos $CAR(V)$ para computar $PAG'(V)$, obtido como:

$CAR(V) = CAR(T) * F(P(T))$

4) *PESQ_DIR(T,X,P(T),U,Q(T);V)*

Há duas situações a considerar:

(i) *U* inverte *T* por *Y*, mas as tuplas de *T* não estão agrupadas por valor de *Y*.

O custo da operação será então:
C = número de páginas de *T* lidas

+ número de páginas de *U* lidas
+ número de páginas de *V* gravadas
+ *W* * número de tuplas de *T* processadas

Na situação considerada aqui, assumiremos que cada tupla de *T* requer a leitura de uma página. Logo, o número de páginas de *T* lidas é igual ao número de tuplas de *T* que satisfazem a *Q(T),* que também coincide com o número de tuplas de *T* processadas. Assumiremos ainda que a estrutura de acesso direto é tal que para se determinar que chaves satisfazem *Q(T)* é necessário acessar no máximo *PAG(U).* Esta é uma estimativa muito conservativa, mas para melhorá-la necessitaríamos assumir uma particular estrutura para *U*, o que não fizemos. Finalmente, observamos que *V* conterá todas as tuplas de *T* que satisfazem a *P(T) AND Q(T).* Logo, teremos:

*CAR(V) = CAR(T) * F(P(T) AND Q(T))*

Lembrando que PAG'(V) é computado de CAR(V), a equação de custo será:

*C = CAR(T) * F(Q(T))*
*+ CAR(U) * F(Q(T))*
+ PAG'(V)
*+ W * CAR(T) * F(Q(T))*

Finalmente, note que como estamos usando uma inversão, tanto *T* quanto *U* devem ser tabelas permanentes do banco, logo são acessadas diretamente de memória secundária, e não podem ser recebidas como tabelas internas ou transientes.

(ii) *U* inverte *T* por *Y* e as tuplas de *T* estão agrupadas por valor de *Y*.

A diferença para o caso anterior está na estimativa do número de páginas de *T* lidas, que neste caso é feito da seguinte forma:

N = (CAR(T) * tamanho das tuplas de *T*) / tamanho da página

O resto da equação permanece o mesmo.

5) *JUNÇÃO_INTERATIVA(T,U,X,Y,P(T,U),P(T),P(U);V)*

Recordemos que uma junção interativa pode ser feita de quatro formas diferentes, dependendo do tipo de pesquisa, seqüencial ou direta, usado para acessar *T* e *U*. Lembremos ainda que o resultado destas pesquisas é sempre passado tupla-a-tupla neste algoritmo, não incorrendo portanto em páginas sendo gravadas. Como os custos destas pesquisas já foram discutidos acima, usaremos *CPESQ(T,P(T))* para representar o custo da pesquisa em *T* usando a qualificação *P(T)* e *CPESQ(U,P(U) AND Q(U))* para representar o custo da pesquisa em *U* usando a qualificação *P(U)* e a qualificação *Q(U)* resultante da substituição de *T* por uma tupla *T* em *P(T,U)* (este custo na verdade será independente da tupla *T* em questão dada a forma como estimamos o fator de seletividade). Assim a equação de custo para junção interativa se reduz a:

C = custo da pesquisa em *T*
+ (número de tuplas que satisfazem *P(T)* * custo da pesquisa em *U*)
+ número de páginas de *V* gravadas

ou seja,

C = CPESQ(T,P(T))
*+ CAR(T) * F(P(T)) * CPESQ(U,P(U) AND Q(U))*
+ PAG'(V)

onde *PAG'(V)* é obtido a partir de *CAR(V),* estimado como:

*CAR(V) = CAR(T) * CAR(U) * F(P(T,U) AND P(T) AND P(U))*

6) *JUNÇÃO_INTERCALAÇÃO(T,U,X,Y,P(T,U),P(T),P(U);V)*

Seja *T* o registro lido de *T* e *t'* o registro anterior. Lembremos que, ao se processar *T*, *U* é pesquisada a partir do primeiro registro que foi usado para processar *t'*. Assumiremos que os registro lidos de *U* são mantidos em áreas de trabalho em memória principal enquanto forem necessários. Assim, *U* só é pesquisada uma vez. Portanto, usando a notação da situação anterior, teremos:

C = CPESQ(T,P(T))
+ CPESQ(U,P(U) AND Q(U))
+ PAG'(V)

4.6 PROCESSAMENTO DE ATUALIZAÇÕES

Esta seção discute brevemente o problema de processar atualizações em bancos de dados centralizados usando novamente SQL como a linguagem de manipulação de dados.

Comecemos com remoções da forma:

```
delete    <nome de relação>
 where  <qualificação>
```

Comandos desta forma são processados em duas fases. Seja *R* a relação referenciada no comando. Inicialmente todas as tuplas de *R* a serem removidas são localizadas como se o comando fosse uma consulta. Em seguida, operações do SAR (não discutidas na Seção 3.3) são invocadas para retirar tais tuplas de *R*.

Considere agora inserções de apenas uma tupla

```
insert into <nome de relação>:  <tupla>
```

ou inserções de múltiplas tuplas:

```
insert into <nome de relação>: <consulta em SQL>
```

Ambos os tipos de inserção são tratados novamente por operações do SAR. Apenas no segundo caso é necessário resolver inicialmente a consulta de SQL para localizar os dados a serem inseridos.

Finalmente, considere atualizações da forma:

```
update  <nome de relação>
    set   <atualização de campos>
 where  <qualificação>
```

Seja *R* a relação referenciada no comando. Neste caso há uma fase inicial em que as tuplas de *R* a serem atualizadas são recuperadas conforme discutido nas seções anteriores, criando-se uma relação temporária *R'*. Em seguida, as atualizações devidas são aplicadas a *R'*. Finalmente, através de operações do SAR, as tuplas antigas de *R* são substituídas pelas novas, que estão contidas em *R'*.

NOTAS BIBLIOGRÁFICAS

A idéia de desmembrar consultas e a técnica de desmembramento controlado são uma adaptação do método de decomposição inicialmente proposto por Wong e Youssefi [1976] e usado no sistema INGRES. Held, Stonebraker e Wong [1975] discutem brevemente como estender decomposição para consultas cujas qualificações são mais complexas, incluindo até funções de agregação. As equações de custo apresentadas são, por sua vez, adaptações das usadas no Sistema *R* e encontram-se descritas em detalhe em Selinger et all. [1979]. Christodoulakis [1983] e Richard [1981] discutem em consideravelmente mais detalhe como estimar o tamanho do resultado de junções e outras expressões da álgebra relacional. O tratamento de consultas e atualizações no Sistema *R* é abordado em uma série de trabalhos. Selinger et all. [1979] descrevem em detalhe o otimizador do Sistema R, cuja performance é analisada em Astrahan, Kim e Schkolnick [1980]. Chamberlin et all. [1981] descrevem em detalhe o processo de compilação de consultas. Vários outros trabalhos abordam extensões do problema de processamento de consultas. Filkelstein [1982] estuda o problema de otimizar conjuntamente seqüências de consultas. Jarke e Kock [1984] e Dayal[1979] investigam o problema de processar consultas cujas qualificações contêm explicitamente quantificadores.

CAPÍTULO 5. PROCESSAMENTO DE COMANDOS DA LMD - O CASO DISTRIBUÍDO

Este capítulo aborda o problema do processamento de consultas e atualizações em bancos de dados distribuídos, estendendo alguns resultados do capítulo anterior. Novamente a maior ênfase recairá sobre o problema de otimização do processamento de consultas.

5.1 INTRODUÇÃO AO PROCESSAMENTO DE CONSULTAS

5.1.1 Discussão Geral

Processamento de consultas sobre um banco de dados distribuído corresponde a tradução de pedidos, formulados em uma linguagem de alto nível, para seqüências de operações elementares sobre os dados armazenados nos vários bancos locais. Difere do caso centralizado em três aspectos básicos:

- o diretório de dados, em geral, é distribuído e a sua forma de armazenamento afeta fortemente a eficiência do processador de consultas;
- como o banco é distribuído, uma relação do esquema conceitual pode estar fragmentada e replicada ao longo da rede. O processador deverá selecionar os fragmentos, localizar as cópias apropriadas e eventualmente movê-las para que a consulta possa ser processada;
- se o sistema for heterogêneo, o processador deverá, ainda, efetuar traduções entre modelos de dados distintos.

Trataremos, brevemente, do problema do diretório de dados em primeiro lugar. Lembremos que o diretório deverá ser acessado antes do processamento da consulta em si para se obter informação acerca das estruturas do banco, incluindo sua localização, bem como estatísticas necessárias à fase de otimização. O diretório, em princípio, pode ser fragmentado e replicado como qualquer banco distribuído. Os critérios usados para sua alocação são os mesmos usados para qualquer banco distribuído. Basicamente, deve-se duplicá-lo na medida do possível, evitando-se assim acessos remotos apenas para se obter informação sobre o banco. Por outro lado, o custo de armazenamento do diretório poderá limitar o fator de replicação, ou forçar a adoção de replicação seletiva de partes do diretório.

Passemos agora aos dois problemas seguintes. À semelhança do caso centralizado, o processo de tradução desmembra-se em quatro etapas, se visto em termos dos esquemas que compõem a descrição do banco de dados distribuído (ver Figura 5.1):

1. Tradução para o Esquema Conceitual Global.
2. Tradução para os Esquemas Conceituais Locais.
3. Processamento Local e Transferência de Dados.
4. Pós-Processamento Global.

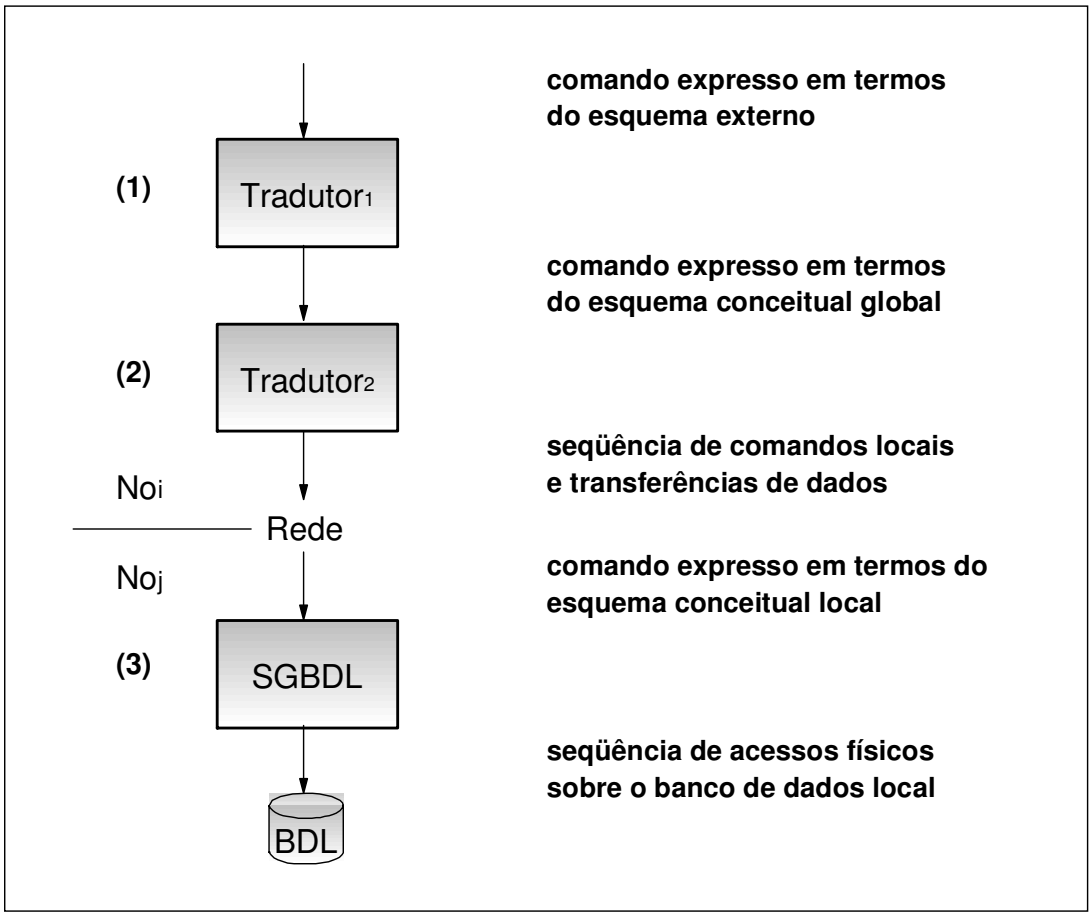

Figura 5.1 Fases do Processamento

Consideremos, inicialmente, o caso mais simples de um sistema homogêneo. A etapa inicial, tradução para o esquema conceitual global, é idêntica ao caso centralizado e mapeia a consulta submetida pelo usuário, como formulada em termos do esquema externo a que este tem acesso, em uma consulta equivalente, mas formulada em termos do esquema conceitual global.

A etapa seguinte, tradução para os esquemas conceituais locais, consiste da tradução da consulta, formulada agora em termos do esquema conceitual global, para uma seqüência de consultas locais e transferências de arquivos. Esta etapa é inteiramente diferente do caso centralizado e deverá resolver os problemas inerentes à distribuição do banco. Novamente, a fase de otimização nesta etapa é a parte crucial do processo.

A terceira etapa, processamento local e transferência de dados, consiste em resolver consultas locais através do SGBD local e não difere, portanto, do caso centralizado. A resolução de uma consulta local poderá, no entanto, envolver transferência prévia de dados.

Finalmente, a etapa de pós-processamento é necessária pois o resultado de uma consulta pode ser deixado sob forma de uma relação distribuída pelas etapas anteriores. Logo, é necessário reunir os fragmentos do resultado em um único nó. Esta etapa, por ser simples, não será tratada explicitamente neste capítulo.

O caso heterogêneo é mais complicado na medida em que os esquemas externos e os esquemas conceituais locais não necessariamente estão no mesmo modelo de dados do esquema conceitual global. Isto torna a primeira e a última etapas mais complexas e está fora do escopo deste texto.

5.1.2 Estratégias de Processamento

A etapa mais crítica do processamento de consultas distribuídas, tradução da consulta para os esquemas conceituais locais, envolve dois aspectos importantes:

1. Tratamento de fragmentos e suas cópias.
2. Escolha das consultas locais e transferências de arquivos.

A forma de resolução destes dois problemas determina uma estratégia de implementação para esta etapa. Em qualquer caso, porém, os parâmetros são sempre o banco de dados em questão (os vários esquemas e as estatísticas sobre o estado corrente), uma determinada função de custo, a própria consulta distribuída e o nome do nó para onde o resultado deverá ser mandado (o *nó final*).

Uma estratégia, talvez a mais simples, consiste em transformar a consulta distribuída em uma consulta local:

PROCESSAMENTO CENTRALIZADO

1. localize os fragmentos necessários à consulta Q;
2. selecione cópias dos fragmentos e um nó n de tal forma a minimizar o custo de mover as cópias para n, e depois mover o resultado para o nó final;
3. mova as cópias selecionadas para n;
4. execute Q localmente em n;
5. mova o resultado de Q para o nó final.

Um refinamento possível consiste em restringir as cópias através de consultas locais antes de movê-las. Um programa consistindo de consultas locais e transferências de dados com o propósito de reduzir o tamanho das cópias a serem movidas é chamado de um *redutor*. Incluiremos no redutor também as transferências finais para o nó onde a consulta submetida será processada. O refinamento aludido resultará, então, na seguinte estratégia:

PROCESSAMENTO CENTRALIZADO COM REDUÇÃO PRÉVIA

1. localize os fragmentos necessários à consulta Q;
2. selecione cópias dos fragmentos, selecione um nó n, defina um redutor P de tal forma a minimizar o custo de executar P, e depois mova o resultado da consulta submetida para o nó final;
3. execute P;
4. execute Q localmente em n;
5. mova o resultado de Q para o nó final.

Esta estratégia será discutida em detalhe na Seção 5.1.2. Ela é interessante pois reduz o problema de processar consultas distribuídas a um problema bem mais simples, mas, por outro lado, leva a soluções sub-ótimas, em geral.

No outro extremo do espectro de possíveis alternativas, temos a estratégia que permite escolher qualquer seqüência de consultas locais e transferências de dados para resolver a consulta distribuída:

OTIMIZAÇÃO IRRESTRITA DE CONSULTAS DISTRIBUÍDAS

1. selecione um nó n e um programa P (em geral paralelo) consistindo de consultas locais e transferências de dados de forma a minimizar o custo total de execução, gerando o resultado em n, e o custo de transferir o resultado para o nó final.
2. execute P;
3. mova o resultado final para o nó final.

Esta estratégia produz um programa ótimo dentre aqueles que utilizam apenas consultas locais e transferências de arquivos. Mas gera, no entanto, um problema de otimização extremamente difícil pois o número de possíveis alternativas a serem exploradas é explosivo.

As Seções 5.1.3 e 5.1.4 discutirão duas variações desta estratégia diferindo no tratamento dos fragmentos, mas ambas baseadas no critério de desmembramento controlado. A primeira delas transforma a consulta distribuída em uma consulta sobre fragmentos antes da fase de otimização:

DESMEMBRAMENTO CONTROLADO DISTRIBUÍDO A NÍVEL DE FRAGMENTOS

1. transforme a consulta Q em uma consulta equivalente Q' já incorporando a estratégia de fragmentação;
2. desmembre Q' em consultas homogêneas sobre os fragmentos, seguindo a estratégia de desmembramento controlado;
3. processe cada subconsulta homogênea sobre os fragmentos através de transferências de dados e consultas locais (a escolha das cópias a serem usadas é feita neste ponto);
4. mova o resultado para o nó final.

A segunda variação a ser estudada desmembra a consulta distribuída antes do tratamento de fragmentos e pode ser descrita brevemente como:

DESMEMBRAMENTO CONTROLADO DISTRIBUÍDO

1. desmembre Q em consultas irredutíveis, ordenando-as segundo a estratégia de desmembramento controlado;
2. processe cada subconsulta irredutível através da redução ao processamento centralizado, otimizando o processo;
3. mova o resultado para o nó final.

Naturalmente, o passo 2 poderia ser alterado como abaixo criando-se uma terceira variação:

2. processe cada subconsulta irredutível através de otimização irrestrita;

Esta breve descrição das estratégias nos dá, então, um panorama das próximas seções.

5.1.3 Interface com o Gerente de Transações

A interface entre o processador de comandos da LMD e o gerente de transações seguirá um padrão único, independentemente da estratégia usada. Assumiremos que o processador de comandos gera um programa concorrente consistindo de consultas e atualizações locais (ou seja, a serem executadas em um único nó) e transferências de arquivos.

Mais precisamente, o processador de comandos gerará sempre programas pertencentes à classe PR, definida indutivamente como:

1. consultas e atualizações locais suportadas pelos SGBDs locais pertencem a *PR*;
2. transferências de tabelas externas de um nó para outro pertencem a *PR*;
3. se *P* e *P'* são programas em *PR*, então a *composição concorrente* de *P* e *P'*, denotada por *P // P'*, também pertence a *PR*;
4. se *P* e *P'* são programas em *PR*, então a *composição seqüêncial* de *P* e *P'*, denotada por *P;P'*, também pertence a *PR*;

Consultas e atualizações locais, como o nome indica, são executadas localmente em apenas um nó pelo SGBD local. Uma transferência indica que alguma tabela externa deve ser movida de um dado nó *n* para um nó *m*. Uma composição concorrente *P // P* indica que *P* e *P'* podem ser executados em paralelo, enquanto que uma composição seqüêncial *P;P'* indica que *P'* só pode ser executado depois que *P* terminar. Em ambos os casos, *P* e *P'* podem ser programas que envolvem tarefas em vários nós.

As seções seguintes descreverão, de forma breve, como uma consulta ou atualização sobre o esquema conceitual global será traduzida em programas do tipo acima.

5.2 PROCESSAMENTO CENTRALIZADO COM REDUÇÃO

Esta seção discute a estratégia mais simples para processamento de consultas distribuídas, qual seja, a transformação ao processamento centralizado. Ênfase será dada a um refinamento em que cópias são reduzidas através de consultas locais antes de serem movidas.

Consideraremos que a rede opera por comutação de pacotes e que o objetivo é minimizar apenas o tráfego de mensagens. Assumiremos, ainda, que o banco é distribuído de tal forma que as relações não são fragmentadas nem replicadas. Ou seja, cada relação reside inteiramente em um nó e não possui cópias. Esta suposição poderá ser relaxada se fragmentos e cópias forem tratados antes da fase de otimização aqui descrita (como na Seção 5.1.3.2).

5.2.1 Introdução

Considere uma estratégia para reduzir o processamento de consultas distribuídas ao caso centralizado que opere em duas fases:

redução: delimite, através de consultas locais, um subconjunto do banco de dados onde a consulta possa ser executada;
execução: envie as partes delimitadas na fase anterior para um nó designado e lá execute a consulta.

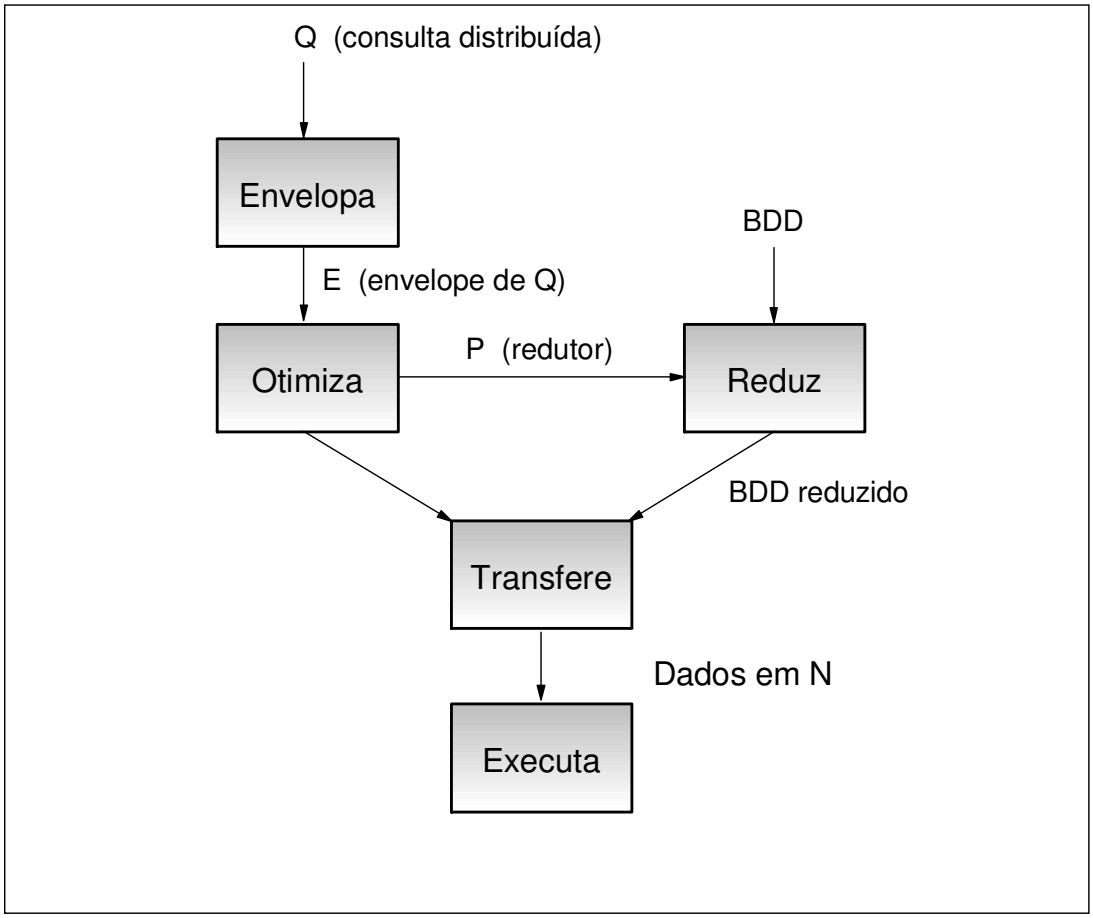

Figura 5.2 - Processamento Centralizado com Redução

O trabalho de otimização, neste caso, se concentra na fase de redução e consiste da escolha de um redutor *P* e um nó *n* de tal forma que o custo de executar *P* e mover os resultados para *n* seja mínimo. O redutor deve ser acompanhado das transferências finais para o nó onde a consulta submetida será processada. A Figura 5.2 resume a seqüência de operações desta estratégia.

5.2.2 Envelopes, Redutores e Semi-junções

Nesta seção investigaremos que operações podem ser usadas para delimitar um subconjunto do banco de dados suficiente para processar uma dada consulta distribuída *Q*. O processo de delimitação começa com a definição de um *envelope* para *Q*, que é um conjunto de consultas a nível do esquema conceitual global. O processo continua com a obtenção de um *redutor* para o envelope, que consiste de uma série de consultas locais e transferências de arquivos. O resultado de processar *Q* contra o banco de dados original e contra o resultado do redutor deve ser o mesmo.

A noção de envelope é definida a nível do esquema conceitual global. Suponha que $R_1,...,R_n$ sejam os nomes de relações mencionados na lista de relações de *Q*. Um envelope para *Q* é um conjunto de consultas $E=\{Q_1,...,Q_n\}$ tal que:

1. Cada Q_i é da forma

```
select t_i
 from R_1 r_1, ... ,R_n r_n
where B
```

onde t_i é a lista dos atributos qualificados de R_i que ocorrem em B e B é uma conjunção de comparações da forma $R_i.A = R_j.B$ ou $R_i.A = R_i$"." $A = k$.

Note que todas as consultas em E possuem a mesma qualificação.

2. Para todo estado S do banco de dados, para todo conjunto S' de relações resultante da execução de E em S, o resultado de Q em S e S' é o mesmo.

Nesta seção não abordaremos o problema de se obter um envelope para uma consulta, mas suporemos que o envelope já é dado.

Um redutor para um envelope E é um programa concorrente P consistindo de consultas locais e transferências de arquivos tal que, para todo estado S do banco de dados, para todo conjunto S' de relações resultante da execução de P em S, o resultado de E em S e S' é o mesmo.

Lembremos que nesta seção a função de custo se restringe ao tráfego de dados na rede. O *benefício* de um redutor P é a diferença entre o custo potencial do tráfego de dados que P economiza e o custo das transferências de dados necessárias para executar P. Dado um envelope $E=\{Q_1,...,Q_n\}$ é possível analisar como um redutor benéfico P deve ser construído. Sejam $R_1,...,R_n$ os nomes de relações envolvidos em E. Recordemos que cada relação está completamente armazenada em um nó, por suposição. Para cada relação R_i, o redutor P deverá conter uma consulta P_i, local ao nó onde R_i reside, tal que:

Projeções: P_i deverá conter na lista resultante apenas os atributos de R_i mencionados em Q_i. O benefício de projeções é sempre positivo e obtido a custo nulo, já que não há transferência de dados.

Restrições: P_i deverá conter na sua qualificação todas as comparações da forma $R_i.A=k$ e $R_i.A=R_i.B$ que forem conseqüência lógica da qualificação de Q_i. (Isto é fácil de se obter pois todas as comparações de Q_i são igualdades. Assim, o problema degenera em um problema de grafos). O benefício de restrições também será sempre positivo e obtido a custo nulo já que não há transferência de dados.

Semi-junções: P_i deverá conter na sua qualificação todas as comparações da forma $R_i.A=R_j.B$ que forem benéficas. O benefício deste tipo de comparação é obtido comparando-se o seu custo C como o volume de dados :f/V/ que economiza, onde:

$C = 0$, se R_i e R_j estão no mesmo nó
$C = c(R_j.B)*c(dom(R_j.B))$, em caso contrário

onde $c(R_j.B)$=cardinalidade de $R_j.B$
$c(dom(R_j.B))$=comprimento dos elementos do domínio de B

O volume de dados que este tipo de comparação economiza é estimado por:

$V = (c_0(R_i) - c_1(R_i)) * c (dom(R_i))$

onde

$c_0(R_i)$ = cardinalidade de R_i antes da semi-junção
$c_1(R_i)$ = cardinalidade de R_i depois da semi-junção
$c(dom(R_i))$ = comprimento médio das tuplas de R_i

As projeções, restrições e semi-junções descritas acima são chamadas de operações *permitidas* por *E*. Uma semi-junção $R_i.A=R_j.B$ é dita *local* se e somente se R_i e R_j residem no mesmo nó. O termo semi-junção advém do fato da comparação $R_i.A = R_j.B$, incluída em P_i, significar a junção de R_i com R_j em *A* e *B*, seguida da projeção sobre os atributos de R_i (pois P_i só contém na sua lista resultante atributos de R_i). Ou seja, a semi-junção é "a metade de uma junção".

As transferências de dados necessárias ao processamento de *P* são implicitamente definidas pelas semi-junções incluídas em *P*. De fato, se a relação R_j não residir no mesmo nó que R_i, será necessário transferir a projeção $R_j[B]$ para o nó onde R_i reside para se resolver a semi-junção $R_i.A=R_j.B$.

5.2.3 Otimização

Esta seção discute o problema de otimização básico da estratégia em discussão. A entrada do otimizador será um envelope *E* e estatísticas *D* sobre o estado corrente do banco. A saída é um redutor *P*, que é estimado ser benéfico para todo estado modelado por *D*, e um nó *n* onde a consulta será executada. O algoritmo de otimização segue o método ganancioso e não garante a obtenção do ótimo global.

```
OTIMIZACAO_REDUCAO(E,D;P,n)
/*
  entrada:      - envelope E    e estatísticas D
  saída:        - redutor P     e nó final n
*/
begin
    /*
      aproximação inicial
    */
    construa P contendo todas as projeções, restrições e semi-junções locais permitidas por E;
    estime o benefício de todas as semi-junções não-locais permitidas por E;
    /*
      incremento a P
    */
    while houver uma semi-junção não-local benéfica e permitida por E do
    begin
        seja sj a semi-junção não-local permitida por E de maior benefício;
        acrescente sj à consulta local em P apropriada;
        acrescente a transferência de dados necessária ao processamento de sj;
        ajuste os benefícios das semi-junções restantes para levar em consideração
        os efeitos de sj;
    end
    /*
      escolha do nó n
    */
    para cada nó m, seja TAM(m) a soma do total de "bytes"
        das relações referenciadas em E e armazenadas em m;
    escolha n como o nó com maior valor de TAM(n);
    acrescente a P comandos para transferir os dados necessários dos outros nós para n;
end
```

Este algoritmo é puramente ganancioso e pode ser melhorado através de uma análise detalhada da seqüência de operações que gera (veja as referências bibliográficas no final deste capítulo). Por exemplo, a escolha do nó final pode tornar alguma semi-junção supérflua. Assim, poder-se-ia acrescentar a seguinte otimização após o final do algoritmo anterior:

seja P_n a consulta local a ser executada em n;
para cada semi-junção $R_i.A = R_k.B$ de P_n do
    se o custo de P sem esta semi-junção for menor do que o custo original de P
        então retire esta semi-junção de P_n

Isto conclui a nossa descrição do método centralizado com redução.

5.3 DESMEMBRAMENTO CONTROLADO DISTRIBUÍDO - PARTE I

Esta seção indica uma primeira forma de se estender a heurística de desmembramento controlado para um ambiente distribuído. A característica principal desta primeira extensão é o tratamento dos fragmentos antes do próprio desmembramento.

5.3.1 Fases do Desmembramento

Desmembramento controlado distribuído a nível de fragmentos é uma forma de implementar a etapa de tradução de consultas, formuladas em termos do esquema conceitual global, para uma série de consultas locais e transferências de arquivos.

As fases desta forma de desmembramento serão as seguintes:

1. Tratamento de Fragmentos: transforma a consulta, expressa em termos do esquema conceitual global, em uma consulta equivalente expressa em termos de fragmentos (sem levar em consideração replicação);
2. Desmembramento controlado: desmembra a consulta em subconsultas homogêneas da forma usual, otimizando o processamento através da tática de desmembramento controlado;
3. Processamento de Subconsultas Homogêneas: traduz cada subconsulta resultante do desmembramento em uma série de subconsultas locais e transferências de arquivos. Nesta etapa, cópias dos fragmentos são selecionadas;
4. Processamento Local: os SGBDs locais processam as subconsultas locais que lhes cabem e transferem arquivos, se necessário.

O processo de otimização será guiado pela heurística de desmembramento controlado como no caso centralizado pois, em um ambiente distribuído, é extremamente importante limitar o espaço de busca da solução ótima já que o número de soluções possíveis é enorme. (além da complexidade inerente da consulta há que se lidar com a distribuição do banco em si).

A função de custo que governa a fase de otimização poderá tanto ser baseada no custo de processamento da consulta, que inclui agora transferências de arquivos, quanto no tempo de resposta. Em qualquer caso, deverá levar em consideração o tipo de rede utilizada: comutação de pacotes ou difusão ("broadcast").

As subseções seguintes tratarão de cada uma das fases do processamento e do problema de otimização e estimação de custo, exceto a fase de processamento local que é idêntica àquela discutida para o caso centralizado.

5.3.2 Tratamento de Fragmentos

O propósito da etapa inicial é transformar uma consulta sobre o esquema conceitual global em uma consulta equivalente que já incorpore informação sobre o critério de fragmentação do banco. O tratamento de fragmentos está baseado na suposição de que o mapeamento do esquema conceitual global para os esquemas conceituais locais é feito em dois níveis:

1. Fragmentação: cada relação é, se desejável, fragmentada;
2. Replicação: cada fragmento é então distribuído para um ou mais nós.

Além disto, supõe-se que há um mapeamento definindo como cada fragmento é formado e um mapeamento definindo como cada relação do esquema conceitual global é reconstruída a partir dos seus fragmentos (ignorando replicação). Ou seja, a forma de fragmentação e replicação de cada relação do esquema conceitual, bem como a forma de reconstrução, são explicitamente definidas. Este cuidado é essencial pois através dos mapeamentos que definem a forma de fragmentação e replicação podemos traduzir inserções, atualizações e remoções globais em seus correspondentes locais e, através dos mapeamentos que definem como reconstruir uma relação do esquema conceitual global a partir dos seus fragmentos, podemos tratar consultas. Observemos ainda que o mapeamento de reconstrução deve agir como o inverso do mapeamento de distribuição para que a definição do banco possa ser considerada como correta. Em geral, não é possível se obter tais inversões automaticamente, daí a necessidade de incluí-las na definição do banco.

Como resultado, temos, efetivamente, um novo nível de descrição do banco de dados distribuído. De fato, a aplicação dos critérios de fragmentação ao esquema conceitual global gera um novo esquema que chamaremos de *esquema conceitual fragmentado*, simplesmente.

Como exemplo considere um banco de dados distribuído em três nós conforme descrito abaixo:

Esquema Conceitual Global (contém apenas uma relação):

$R[A,B,C,D,E]$

Critérios de Fragmentação de *R*:

$R_1[A,B,C]$	tal que R_1 é a projeção de R em A, B, C;
$R_2[A,D,E]$	tal que R_2 é a projeção de R em A,D,E seguida da seleção das tuplas com $A<"a"$;
$R_3[A,D,E]$	tal que R_3 é a projeção de R em A,D,E seguida da seleção das tuplas com $A\geq"a"$;

Critério de Reconstrução de *R*:

R é reconstruída da seguinte forma:

```
select R1.A,R1.B,R1.C,R2.D,R2.E
  from R1, R2
 where R1.A=R2.A
union
select R1.A,R1.B,R1.C,R3.D,R3.E
  from R1, R3
 where R1.A=R3.A
```

Critérios de Distribuição dos Fragmentos pelos Esquemas Conceituais Locais:

Nó 1: contém apenas uma cópia de R_1
Nó 2: contém cópias de R_1 e R_2
Nó 3: contém apenas uma cópia de R_3

Estando o banco de dados distribuído definido da forma acima, o tratamento de fragmentos torna-se idêntico à tradução de consultas sobre esquemas externos para consultas sobre o esquema conceitual. Por exemplo, considere a seguinte consulta sobre o esquema conceitual global do banco anteriormente descrito:

```
select A,D,E
  from R
 where B = 'b'
```

Sem nos preocuparmos com o problema de cópias, podemos traduzir esta consulta em subconsultas sobre os fragmentos utilizando apenas o mapeamento de reconstrução para *R*. A consulta resultante, *Q'*, seria:

```
select R1.A, R2.D, R2.E
  from R1, R2
 where R1.A = R2.A
   and R1.B = 'b'
union
select R1.A, R3.D, R3.E
  from R1, R3
 where R1.A = R3.A
   and R1.B = 'b'
```

Isto conclui a discussão sobre o tratamento de fragmentos.

5.3.3 Desmembramento Propriamente Dito

A fase central do processo, o desmembramento propriamente dito, transforma uma consulta (formulada em termos do esquema conceitual fragmentado) em uma série de consultas homogêneas (também formuladas em termos do esquema conceitual fragmentado) e transferências de arquivos. A heurística de desmembramento controlado é utilizada nesta fase para reduzir o trabalho de otimização mas, ao mesmo tempo, evitar a construção de resultados intermediários vultuosos sem necessidade.

A definição da heurística em nada é alterada em relação àquela discutida para o caso centralizado, exceto no que se refere à função de custo. A consulta será desmembrada em suas componentes irredutíveis. Estas componentes serão parcialmente ordenadas de tal forma que as subconsultas homogêneas sejam executadas primeiro, se possível, e a componente contendo a lista resultante seja executada por último. Cada componente será otimizada em separado, ignorando-se as outras componentes.

A função de custo que governa a fase de otimização poderá tanto ser baseada no custo de processamento da consulta quanto no tempo de resposta, e deverá levar em consideração o tipo de rede utilizada: comutação de pacotes ou difusão ("broadcast").

Consideremos, a guisa de exemplo, o caso de custo de processamento em uma rede por comutação de pacotes. A função de custo, neste caso, será composta de três fatores:

$$C = k_1 * \textit{custo de processamento local} + k_2 * \textit{custo de I/O local} + \textit{custo de transmissão de dados}$$

onde k_1 e k_2 são constantes de mérito relativo. As duas primeiras parcelas são idênticas ao caso centralizado. Já o custo de transmissão de dados é, em geral, estimado levando-se em conta apenas o número de pacotes necessários para transferir uma tabela de um nó para outro. Ou seja, o custo de transmitir um pacote é o mesmo entre qualquer par de nós. O número de pacotes é, por sua vez, estimado em termos do tamanho do pacote, da cardinalidade estimada da tabela e do comprimento médio dos seus registros. Note que a estimação dos dois primeiros fatores já foi discutida quando tratamos do caso centralizado.

Por fim, observamos que a estimação do custo de processamento de uma consulta genérica é a soma das estimativas de custo das subconsultas homogêneas em que foi desmembrada. Por sua vez, em vista das estratégias para processamento de consultas homogêneas distribuídas, a estimativa do custo de processamento destas consultas recai no caso centralizado e na estimação do custo de transferência de tabelas, já discutido.

5.3.4 Processamento de Consultas Homogêneas

Nesta etapa, uma subconsulta homogênea é mapeada em uma série de consultas locais e transferências de arquivos. O resultado da subconsulta homogênea poderá ser deixado fragmentado em vários nós. As subconsultas locais e as transferências de arquivos serão processadas pelos SGBDs locais, explorando o paralelismo da rede sempre que possível.

A subconsulta considerada nesta fase já está re-escrita em termos de fragmentos e é homogênea com relação a este nível de descrição do banco. Os problemas a serem resolvidos consistem, então, da escolha das cópias dos fragmentos a serem usadas e da definição de uma estratégia de transferência de dados para reduzir a consulta a subconsultas locais.

Consideraremos consultas univariáveis em separado daquelas de duas variáveis.

(A) Consultas Homogêneas Univariáveis

Uma consulta homogênea univariável, a nível do esquema conceitual fragmentado, referencia apenas um fragmento. Portanto o seu processamento é simples e consiste apenas de três passos:

1. Escolha da cópia do fragmento a ser utilizada;
2. Envio da consulta para o nó que armazena a cópia escolhida;
3. Processamento local da consulta, permanecendo o seu resultado no próprio nó.

(B) Consultas Homogêneas de Duas Variáveis

Sejam F_1 e F_2 os dois fragmentos referenciados pela consulta homogênea de duas variáveis Q em questão. Há várias estratégias a considerar:

Estratégia 1:

Se existem cópias dos fragmentos em um mesmo nó i, a consulta poderá ser executada localmente em i e o seu resultado aí permanecer.

Estratégia 2:

Sejam Q_1 e Q_2 as subconsultas de Q geradas pelas cláusulas de Q que referenciam apenas F_1 e F_2, respectivamente. Seja Q_3 a subconsulta de Q gerada pelas cláusulas de Q que referenciam F_1 e F_2. Cópias de F_1 e F_2 residindo em nós diferentes, i e j, são selecionadas e Q_1 e Q_2 executadas localmente em i e j. Os resultados são enviados para um terceiro nó k onde Q_3 é executada. O resultado de Q é, então, o resultado de Q_3 e permanece em k.

Estratégia 3:

Seja Q_1 como na estratégia 2. Seja Q'_2 a subconsulta de Q gerada pelas cláusulas de Q que não foram usadas para gerar Q_1. Cópias de F_1 e F_2 residindo em nós diferentes, i e j, são selecionadas e Q_1 é executada localmente em i. O resultado é enviado para j onde QPRIME $_2$ é executada. O resultado de Q é, então, o resultado de Q'_2 e permanece em j.

Estratégia 4:

Idêntica à anterior, invertendo-se os papéis de F_1 e F_2.

Isto encerra a discussão sobre o processamento de consultas homogêneas.

5.4 DESMEMBRAMENTO CONTROLADO DISTRIBUÍDO - PARTE II

A variação do desmembramento controlado apresentado na Seção 5.1.3 possui duas características importantes:

1. Uma consulta é mapeada para o esquema conceitual fragmentado antes do desmembramento;
2. Só há movimento de dados para processar as subconsultas homogêneas (sobre o esquema conceitual fragmentado).

Esta forma de conduzir o processamento pode gerar, no entanto, movimentos de dados desnecessários, pois o desmembramento já é a nível de fragmentos.

Considere, por exemplo, a seguinte consulta:

```
select R.A
  from R, S
where R.B=S.B
```

Suponha que R e S estão fragmentados horizontalmente em R_1, R_2 e S_1, S_2 respectivamente. Então a consulta anterior seria inicialmente transformada em:

```
select R1.A
  from R1, S1
where R1.B=S1.B
union
select R1.A
  from R1, S2
where R1.B=S2.B
union
select R2.A
  from R2, S1
where R2.B=S1.B
union
select R2.A
  from R2, S2
where R2.B=S2.B
```

Assumindo que os quatro fragmentos estão em nós separados, são necessárias quatro transferências de dados, digamos, S_1 e S_2 para o nó de R_1, e S_1 e S_2 para o nó de R_2 para resolver as quatro consultas acima. Além disto, são necessárias duas transferências de dados para mover os resultados dos nós onde estão armazenados R_1 e R_2 para o nó final. Em contraste, os quatro fragmentos poderiam ser simplesmente movidos para o nó final, economizando-se, assim, 2 transferências.

De forma geral, uma alternativa plausível para o método de resolução apresentado na seção anterior seria desmembrar a consulta em componentes irredutíveis antes de tratar os fragmentos. Obter-se-ia, primeiramente, as subconsultas irredutíveis a nível de esquema conceitual global para depois tratar de fragmentos e cópias. Cada subconsulta irredutível Q poderia ser processada por desmembramento irrestrito ou por redução ao processamento centralizado. Em ambos os casos a heurística de desmembramento controlado seria usada com o propósito de reduzir o problema de otimização inicial a uma série de problemas de otimização mais simples (referentes às subconsultas irrestritas), evitando ainda a obtenção de resultados intermediários demasiadamente grandes.

5.5 PROCESSAMENTO DE ATUALIZAÇÕES

Esta seção discute brevemente o problema de se processar atualizações em bancos de dados distribuídos usando novamente SQL como a LMD básica. Inicialmente, os problemas comuns a todas as formas de atualização serão apresentados. Em seguida, o processamento de cada tipo de atualização será discutido em detalhe.

Conforme apresentado em detalhe na Seção 3.2.2, SQL possui quatro comandos básicos para atualização do banco:

- remoções da forma:

```
delete <nome de relação>
where <qualificação>
```

- inserções de apenas uma tupla:

```
insert into <nome de relação>: <tupla>
```

- inserções de múltiplas tuplas:

```
insert into <nome de relação>: <consulta em SQL>
```

- atualizações da forma:

```
update <nome de relação>
    set <atualização de campos>
 where <qualificação>
```

Há três problemas básicos a serem resolvidos no contexto de atualizações em BDDs:

1. localização dos dados afetados pela atualização;
2. atualização dos dados em si;
3. tratamento de fragmentos e cópias.

De forma geral, e independentemente do tipo de atualização, estes problemas são equacionados da seguinte forma. O primeiro problema é resolvido utilizando-se as técnicas descritas anteriormente para processar consultas, exceto que os resultados poderão ser deixados sob forma de uma relação distribuída. O segundo problema é tratado através de novas operações do subsistema de armazenamento (que não são descritas aqui) para atualização de campos de tuplas. O terceiro problema já não é tão simples pois, como as atualizações são descritas a nível do esquema conceitual global, é necessário mapeá-las para atualizações a nível de fragmentos e cópias. Em toda a sua generalidade, este é um problema extremamente difícil, sendo equivalente ao problema de atualizar visões (ou esquemas externos. Veja as referências bibliográficas para tratamentos deste problema). No entanto, através de certas hipóteses sobre como fragmentos e cópias são definidos, gerados e mantidos é possível contorná-lo de forma satisfatória. Para tal, assumiremos que os fragmentos são gerados segundo a política abaixo:

1. cada tupla *t* de uma tabela *t* a nível do esquema conceitual global recebe um identificador interno (isto já era feito pelo SAR no caso centralizado)
2. cada tupla *t'* resultante da fragmentação de *t* carrega consigo a identificação da tupla original *t* e é inserida em todas as cópias do fragmento de *t* a que pertence. Se houver fragmentação horizontal e esta não definir uma partição de *t*, insira *t'* em todos os fragmentos horizontais de *t* possíveis;
3. quando *t* for removida por um comando da LMD, remova todas as tuplas com o mesmo identificador que *t* de todos os fragmentos de *t* e suas cópias;
4. tuplas que não puderem ser mapeadas em nenhum fragmento pelos critérios de distribuição serão tratadas como violações de um critério de consistência implícito.

Cada tipo específico de atualização é tratado da seguinte forma.

Considere inicialmente uma inserção do seguinte tipo:

```
insert into R: Q
```

onde *Q* é uma consulta em SQL. Seja *R*' a relação resultante de *Q*.

Esta inserção será tratada da seguinte forma:

1. execute *Q*, criando *R*' em um único nó;
2. gere novos identificadores para as tuplas de *R*';
3. fragmente *R*' em R'_1 , … , R'_n de acordo com os critérios de fragmentação de *R* e seguindo a política anteriormente descrita para tratamento de fragmentos;
4. envie R'_i para cada nó contendo uma cópia do fragmento correspondente de *R*;
5. faça as inserções locais através de operações do SAR.

O tratamento de inserções de apenas uma tupla é idêntico, apenas que o primeiro passo não é necessário.

Considere agora uma remoção do tipo:

```
delete R where B
```

O tratamento de remoções deste tipo segue o seguinte procedimento:

1. resolva, através dos algorítmos usados para processamento de consultas, a qualificação *B* recuperando apenas os identificadores de tuplas. O resultado pode ser deixado sob forma de uma relação distribuída *R*';
2. envie o conjunto de identificadores recuperados para todos os nós contendo cópias de algum fragmento de *R*;
3. remova localmente, através de operações do SAR, todas as tuplas de cópias de fragmentos de *R* cujos identificadores estão no conjunto recebido.

Finalmente, considere o tratamento de atualizações de campos:

```
update R
    set R.A1=E1,...,R.An=En
 where B
```

Este tipo de comando apresenta uma complicação adicional gerada pelo fato de, ao alterar o valor de um campo, o usuário implicitamente força a migração de uma tupla de um fragmento para outro. Uma forma de processar este tipo de comando seria:

1. recupere todas as tuplas de *R* que satisfazem *B* através dos algorítmos de processamento de consultas usuais. A relação resultante *R*' poderá ser deixada distribuída, mas deverá incluir os identificadores das tuplas;
2. faça todas as modificações definidas pelo comando nos fragmentos de *R*';
3. distribua apenas os identificadores recolhidos em *R*' para todos os nós que contêm cópias de fragmentos de *R*;
4. redistribua *R*' de acordo com os critérios de fragmentação de *R;*
5. em cada nó armazenando uma cópia de um fragmento de *R,* remova as tuplas cujos identificadores estão no conjunto de identificadores recebido e insira as novas tuplas recebidas, se este for o caso.

Isto conclui a nossa breve descrição sobre processamento de comandos da LMD em bancos de dados distribuídos.

BIBLIOGRAFIA

Wong [1977], um dos primeiros trabalhos em otimização de consultas distribuídas, descreve uma heurística de otimização (gananciosa) baseada apenas em transferências de arquivos e consultas locais. Hevner e Yao [1978], também um dos trabalhos pioneiros, apresenta uma heurística mais sofisticada. O algorítmo centralizado com redução prévia é usado no sistema SDD-1 e descrito em Goodman et all. [1979]. A técnica de usar semi-junções para diminuir o tráfego de dados recebeu considerável atenção recentemente. Bernstein e Chiu [1981] foram dos primeiros a explorar esta técnica em detalhe. Yu e Chang [1983] explicam como melhorar o método baseado em semi-junções descrito no texto, incorporando também o problema de tratar cópias múltiplas. O método de desmembramento controlado distribuído é inspirado no algorítmo de decomposição distribuído usado no sistema INGRES e apresentado em Epstein, Stonebraker e Wong [1978] e Stonebraker [1980]. O processo de otimização de consultas usado no Sistema R* é esboçado em Selinger e Adiba [1980]. Daniels et all. [1982] resumem a estratégia de compilação usada no Sistema R*. Maiores detalhes encontram-se em Daniels [1982] e Ng [1982]. Dayal [1983] apresenta o método usado para processar consultas no sistema MULTIBASE, que é heterogêneo e possui uma arquitetura bem peculiar.

Capítulo 6. INTRODUÇÃO À GERÊNCIA DE TRANSAÇÕES

No capítulo anterior foram descritos procedimentos através dos quais comandos que partem do usuário são decompostos em ações elementares e distribuídos através dos vários nós que compõem o sistema. As preocupações básicas recaíam, tipicamente, em técnicas para se determinar quais os mecanismos de acesso mais eficientes às tabelas residentes em cada nó particular; em como deslocar dados entre os nós quando a informação necessária encontra-se distribuída ou mesmo replicada pelos vários nós; etc. Neste capítulo, e de ora em diante, tais questões serão pressupostas como devidamente equacionadas. Tudo se passa como se adentrássemos uma camada mais interior do sistema de gerência do banco de dados à qual são requisitadas tarefas bem determinadas. O leitor deve estar atento para o fato de que tais tarefas ou seqüências de ações elementares manipulam os dados a nível físico, uma vez que o processador de comandos da LMD, discutido no capítulo anterior, já resolveu, a partir de uma análise da consulta do usuário, todos os dilemas a nível lógico.

O objetivo primordial deste capítulo é dar uma visão panorâmica dos problemas enfrentados pelo gerente de transações. O conceito fundamental de uma transação é introduzido na próxima seção. Toda discussão subseqüente dele dependerá criticamente. A seção seguinte aborda aspectos relativos ao gerenciamento de transações e discute os procedimentos que devem ser efetivados em vários pontos específicos durante a execução de transações. A próxima seção toca nos problemas advindos de falhas no sistema que perturbam o ciclo de vida normal de transações em execução. Por questões de eficiência do sistema como um todo é desejável que várias transações executem simultaneamente em cada nó. Isto acarreta problemas de controle de concorrência que são mencionados na última seção deste capítulo. Os três capítulos seguintes detalham as questões relativas ao gerenciamento de transações, controle de concorrência e falhas no sistema, respectivamente.

6.1 O CONCEITO DE TRANSAÇÃO

6.1.1 Noções Básicas

A nível lógico, um banco de dados distribuído é descrito por um esquema conceitual global. Neste e nos capítulos que se seguem, os objetos descritos no esquema conceitual global serão chamados de *objetos lógicos*, e uma função associando a cada objeto lógico um valor será chamada *de estado lógico* do banco. A linguagem de manipulação de dados (LMD) oferece então uma série de comandos para criar, destruir ou modificar objetos lógicos. Embora a natureza dos objetos lógicos e as características da LMD dependam do modelo de dados usado para descrever esquemas conceituais, a discussão neste capítulo será largamente independente destes detalhes.

A nível físico, o banco de dados é definido através de uma série de esquemas internos, um para cada nó em que o banco é armazenado. Os objetos descritos nos esquemas internos serão chamados de *objetos físicos* ou simplesmente *objetos.* A cada objeto físico está associado um único *nome* de tal forma que objetos diferentes recebem nomes distintos. Os operadores atômicos que atuam sobre os objetos físicos serão chamadas de *ações elementares*. Finalmente, a cada instante, o *estado físico* ou, simplesmente, o *estado* do banco de dados associa a cada objeto físico um determinado *valor*, que poderá mudar por meio de uma ação elementar.

O relacionamento entre o esquema conceitual global e os vários esquemas internos é definido através de mapeamentos, que servem a dois propósitos:

1. indicar os objetos físicos que implementam um objeto lógico;

2. indicar como construir um estado lógico do banco a partir do estado físico corrente.

A cada objeto lógico poderão naturalmente estar associados vários objetos físicos através destes mapeamentos.

Neste capítulo, para efeitos dos exemplos, assumiremos que o modelo relacional é usado para descrever o esquema conceitual global e que a linguagem de manipulação de dados é SQL. Portanto, os objetos lógicos serão relações, tuplas e campos de tuplas. Quanto aos objetos físicos, assumiremos que são tabelas do SAR, e não páginas e segmentos conforme seria de se supor tendo em vista a discussão da Seção 3.3. Isto é uma mera conveniência para tornar os exemplos mais simples. Além disto, assumiremos ainda que as tabelas são descritas como "arrays" em PASCAL, o que possibilita usar a notação usual de PASCAL para descrever ações elementares sobre as tabelas.

Um exemplo simples servirá para esclarecer a discussão acima, um tanto compacta. Considere um esquema conceitual global contendo apenas um esquema de relação, definida como (ignorando os domínios dos atributos):

CONTAS[CPF,NOME,SALARIO,ENDERECO,NUMERO_CONTA,SALDO,SALDO_MEDIO]

A chave neste caso é NUMERO_CONTA.

Suponha que este esquema de relação é mapeado em duas tabelas, CADASTRO e CONTAS_CORRENTES (a localização exata das tabelas não é importante), descritas da seguinte forma:

```
type PESSOA = record of
        CPF = integer;
        NOME = array[1..30] of CHAR;
        SALARIO = real;
        ENDERECO = array[1..5,1..50] of CHAR
end;
MOVIMENTO = record of
        NUMERO_CONTA, CPF integer;
        SALDO, SALDO_MEDIO = real
end;
var CADASTRO = array[1..1000] of PESSOA;
CONTAS_CORRENTES = array[1..1000] of MOVIMENTO;
```

O mapeamento de CONTAS nas tabelas CADASTRO e CONTAS_CORRESTES é definido da seguinte forma:

CADASTRO é a projeção de CONTAS em CPF, NOME, SALARIO, ENDERECO;

CONTAS_CORRENTES é a projeção de CONTAS em NUMERO_CONTA, CPF, SALDO, SALDO_MEDIO.

Os objetos lógicos deste banco serão dados pela relação associada a CONTAS, as tuplas da relação associada a CONTAS e os campos destas tuplas. Note que as tuplas, ao contrário das relações, não têm um nome específico. Os objetos físicos serão as tabelas, os elementos das tabelas e seus campos.

Um comando de SQL, a LMD usada no exemplo, refere-se a uma relação pelo seu nome (dado no esquema conceitual), mas identifica as tuplas de uma relação pelos valores dos seus campos. Observe a seguinte atualização:

```
update CONTAS
        set SALDO = SALDO - 1000
        where NUMERO_CONTA = 34.567
```

Através de uma série de transformações, um comando da LMD eventualmente gera uma seqüência de ações elementares. Por exemplo, o comando anterior poderia gerar a seguinte seqüência

```
A1 = (LEIA,CONTAS_CORRENTES[4].SALDO,X)
A2 = (COMPUTE,X:=X-1000)
A3 = (ESCREVA,CONTAS_CORRENTES[4].SALDO,X)
```

onde X representa uma variável tipo "real" e :f/CONTAS_CORRENTES[4] indica que o quarto elemento de CONTAS_CORRENTES corresponde a tuplas de CONTAS com NUMERO_CONTA = 34.567.

Por último, há a noção de consistência de estados. Podemos visualizar também dois níveis de consistência para o banco, *consistência lógica* e *consistência interna*.

Considere consistência lógica primeiro. É fácil de aceitar que, sob o ponto de vista do usuário, cada estado do banco deve satisfazer certos *critérios de consistência*. Por exemplo, o saldo das contas deverá ser sempre positivo. Um estado lógico do banco é *consistente* quando satisfizer a todos os critérios de consistência. Caso contrário, diz-se que o estado é *inconsistente*. Suporemos aqui que não seja de responsabilidade do SGBD cuidar para que violações dos critérios de consistência não ocorram. Isto será obrigação do usuário ao codificar as atualizações do banco.

Já consistência interna se refere à coerência das estruturas internas (objetos físicos) usados para armazenar o banco. Por exemplo, uma árvore-B poderá ser usada para manter um índice sobre determinados campos de uma tabela. Isto implica que as chaves e ponteiros da árvore-B devem estar sempre consistentes com os dados armazenados na própria tabela. Dentro deste conceito, o SGBD deverá, então, ser responsável pela manutenção da consistência interna do banco.

6.1.2 Transações

Suponha que um usuário apresente a seguinte atualização a um banco de dados usado para processar compensação de cheques bancários.

```
Debite $100,00 à conta corrente do cliente N
Credite $100,00 à conta corrente do cliente M
```

Após manipulação pelo processador de consultas, teríamos algo como a seguinte seqüência de ações elementares

A1 = (LEIA,CONTA_CORRENTE[A].SALDO,X)

A2 = (COMPUTE, W:=X-100)

A3 = (ESCREVA,CONTA_CORRENTE[A].SALDO,W)

A4 = (LEIA,CONTA_CORRENTE[B].SALDO,X)

A2 = (COMPUTE, W:=X+100)

A6 = (ESCREVA,CONTA_CORRENTE[B].SALDO,W)

onde A e B são os elementos de CONTA_CORRENTE correspondentes às tuplas de CONTAS com NUMERO_CONTA = N e NUMERO_CONTA = M, respectivamente.

Por alguma razão qualquer (falta de energia, falhas no equipamento ou programas, etc.)., a atividade do banco de dados é interrompida após a execução da ação elementar A3. Ao retornar à atividade, teríamos debitado $100,00 à conta de *A* e ainda não creditado valor algum à conta de B. Isto colocaria o banco de dados num estado inconsistente, pois $100,00 teriam simplesmente "desaparecido". Seria muito conveniente se pudéssemos assegurar que a seqüência de ações elementares acima tivesse a propriedade de que "ou *todas* as ações elementares executam com sucesso ou *nenhuma* delas é executada". Em outras palavras, gostaríamos de estender o conceito de atomicidade de modo a envolver agora também seqüências de ações elementares.

A atomicidade de seqüência de ações elementares poderia também ser violada por interferência de outras ações executadas concorrentemente (exemplos serão dados no Capítulo 8).

A noção de *transação* é introduzida para forçar o sistema a executar uma seqüência de ações elementares como se fosse uma unidade atômica, sem interferência externa.

A nível lógico, uma transação é definida através de um programa em uma linguagem de alto nível, contendo comandos da LMD embebidos, e iniciado e terminado pelos comandos COMEÇO-DE-TRANSAÇÃO e FIM-DE-TRANSAÇÃO. Na sua forma mais simples, uma transação contém apenas um comando da LMD. Exige-se do usuário que codifique as transações de tal forma que quando executadas sozinhas:

T1. sempre terminem;

T2. preservem a consistência do banco de dados.

Exige-se do SGBD, por sua vez, que a cada invocação de uma transação *T*:

S1. a transação *T* seja executada por completo;

S2. a execução da transação *T* se dê sem interferência de outras transações que porventura estejam sendo executadas concorrentemente.

Lembremos que a cada transação *T* corresponde uma seqüência de ações elementares sobre os objetos físicos. O primeiro requisito, S1, garante então que o efeito líquido, isto é, aquele que será tornado público após o término da chamada a *T*, refletirá o fato de que *todas* ou *nenhuma* das ações elementares de *T* foram executadas. Note que isto não implica em que cada uma das ações de *T* deva ser ativada apenas uma única vez. Para ver isto, suponha que existam, no repertório de ações elementares do nó onde *T* foi executada, as ações elementares $\neg A_1$,…, $\neg A_n$ onde$\neg A_i$ indica a ação elementar de efeito exatamente contrário à A_i. Em outras palavras, $\neg A_i$ devolve o banco de dados ao estado anterior à execução de A_i. Retornando ao exemplo anterior sobre compensação bancária, é fácil ver que A_1 , como uma

operação de "leitura", não altera o estado do banco. Portanto, seria válido colocar $\neg A_1$ = (NULO), isto é, à $\neg A_1$ não corresponde operação alguma. Idem, $\neg A_2$ = (NULO).

Continuando, $\neg A_3$ = (ESCREVA, CONTA_CORRENTE[A].SALDO, X) pois a variável X detém o conteúdo inicial de CONTA_CORRENTE[A].SALDO até o término da transação. Note que isto restaura o valor do objeto CONTA_CORRENTE[A] ao valor que apresentava antes da execução de A_3. Podemos definir $\neg A_4$, $\neg A_5$ e $\neg A_6$ analogamente.

Dentro deste espírito, se $E = (A_1, A_2, A_3, A_4)$ for a seqüência de ações elementares gerada por uma execução seqüencial de *T*, então cada uma das seqüências de ações elementares abaixo, quando ativadas pelo sistema, satisfazem ao requisito contido no item (1) acima.

A_1 , A_2 , A_3 , A_4

$A_1 , A_2 , \neg A_2 , A_2 , A_3 , A_4 , \neg A_4 , A_4$

$A_1 , \neg A_1 , A_1 , \neg A_1 , A_1 , A_2 , A_3 , \neg A_3 , \neg A_2 , \neg A_1 , A_1 , A_2 , A_3 , A_4$

Na realidade, embora as ações elementares $\neg A_i$ pareçam um tanto obtusas no momento, desempenharão um papel crucial nas idéias a serem desenvolvidas mais adiante. O leitor atento já deve ter percebido isto. Voltando ao exemplo da compensação bancária, caso o sistema falhe após a execução das ações elementares A_1 , A_2 *e* A_3 , e supondo que os objetos envolvidos residam em memória secundária, ao retornar à atividade o banco de dados estaria em um estado refletindo uma execução parcial de *T*. Neste ponto, após executada as ações

$U = (\neg A_3 , \neg A_2 , \neg A_1)$ o banco de dados recairia novamente em um estado que não reflete nenhuma ação de *T*, podendo retomar suas atividades normais.

O segundo requisito, S2, não exclui a possibilidade de intercalar ações elementares de transações diferentes sobre o mesmo banco de dados local, ou de executar paralelamente ações elementares da mesma transação ou de transações diferentes sobre bancos de dados locais distintos. No entanto, o requisito S2 exige que algum controle seja exercido para que o nível de concorrência permitido não destrua a idéia de atomicidade da execução das transações.

6.2 EXECUTANDO TRANSAÇÕES: INÍCIO, MIGRAÇÃO E TÉRMINO

Recordemos inicialmente a arquitetura para SGBDDs introduzida na Seção 1.1.3. Em um primeiro nível, esta arquitetura divide o SGBDD em uma coleção de SGBDs locais interligados pelo SGBD global. Cada nó da rede possui uma cópia do SGBD global. Este, por sua vez, é dividido em três grandes componentes:

1. diretório de dados global (DDG): contém descrições dos objetos lógicos e físicos e dos mapeamentos entre estes;

2. gerente de transações (GT): interpreta e controla o processamento de consultas e transações submetidas ao sistema;

3. gerente de dados (GD): interface com o SGBD local, fazendo as traduções necessárias no caso de sistemas heterogêneos.

O propósito desta seção será apresentar em mais detalhe como um GT processa transações.

Considere uma transação sobre um determinado banco de dados definida por um programa em uma linguagem de alto nível contendo comandos da LMD embebidos. Como tal, a transação terá que passar por uma compilação inicial. Durante esta fase, os comandos da LMD encontrados poderão ser tratados de duas formas distintas. Uma estratégia seria preparar, já nesta fase, um plano completo para sua execução, o que equivaleria a uma compilação prévia dos comandos. Uma segunda estratégia seria substituir cada comando da LMD por uma chamada para o SGBDD, postergando o tratamento do comando. Em qualquer caso, o resultado da fase de compilação será um programa objeto contendo chamadas para o SGBDD.

Quando a transação *T* é chamada, o gerente de transações do seu nó de origem assume o controle do seu processamento. O GT aciona mecanismos apropriados

- antes do início da transação
- quando do término da transação
- durante a execução da transação

Quando do início da transação, o gerente de transações deve identificar a transação de maneira unívoca. Esta identificação será estendida a cada ação elementar executada a favor desta transação. Desta forma, no caso de ser necessário desfazer parte das ações elementares associadas a uma particular transação, o gerente de transações terá condições de fazê-lo sem problemas. Também é importante iniciar todo um contexto apropriado do qual dependerão os mecanismos de controle de concorrência e controle de integridade, que poderão ser acionados durante a execução da transação.

Durante a execução do programa-objeto que define a transação, o SGBDD é chamado em cada ponto onde havia um comando da LMD. Se a estratégia de compilação prévia não foi adotada, o GT intersepta estas chamadas e invoca o processador de comandos da LMD para criar um plano de execução para o comando. Caso tenha sido, o GT apenas verifica se o plano ainda continua válido, refazendo-o se houve mudanças no banco de dados que o invalide.

Um plano de execução é descrito por um programa concorrente consistindo de consultas e atualizações para serem executadas pelos SGBDs locais, além de transferências de arquivos.

Uma vez de posse do plano de execução para o comando corrente, o GT chama o executor de transações, ET, para interpretar o plano. O papel do ET consiste em enviar as consultas, atualizações e transferências para os nós apropriados e garantir que são executadas na ordem correta e em paralelo, quando o plano assim o permitir.

Durante a execução de transações sob seu controle, o gerente de transações interage tanto com os mecanismos de controle de concorrência quanto com os mecanismos de controle de integridade. É preciso que o gerente de transações esteja atento ao fato de que recursos, do sistema estarão sendo requisitados pelas várias transações em andamento. Ao conceder o uso de determinados recursos o gerente de transações aciona os procedimentos de controle de concorrência para evitar que mais de uma transação tenha acesso, simultaneamente, a recursos que não podem ser compartilhados. Por outro lado, o próprio fato de cancelar transações pressupõe, também, que o gerente de transações mantém um histórico da

seqüência de ações que vão sendo executadas em favor da transação. De outra forma, pode ser impossível realizar a função de controle de integridade uma vez que não haveria meios de se inverter efeitos de ações elementares passadas. Manter este histórico é tarefa dos processos de controle de integridade com os quais o gerente de transações deve, portanto, manter estreito contato durante a execução da transação.

Quando do término da transação, uma decisão deve ser tomada no sentido de tornar público *todos* os efeitos das ações elementares executadas em benefício da transação ou *nenhum* deles, em cujo caso as ações elementares já invocadas devem ter seus efeitos removidos do BD. No primeiro caso, diz-se que a transação foi *confirmada* reservando-se o termo *cancelada* para o segundo caso. Uma transação pode terminar de forma natural ou devido a causas excepcionais. Quando terminada de maneira excepcional, tipicamente devido a falhas ou a incompatibilidades insolúveis na repartição dos recursos do sistema, como veremos nas seções seguintes, a transação será sempre cancelada. Neste instante, entram em ação os mecanismos de controle de integridade que inverterão a transação, restaurando o BD a um estado consistente de onde possa recomeçar suas operações normais. Transações também podem ser canceladas mesmo estando em processamento normal. Isto pode ser visualizado num cenário onde transações são executadas iterativamente e o usuário tem a opção de simplesmente abandonar a transação antes de completá-la. Neste caso, a transação está sendo cancelada a pedido do usuário, não devido a fatores anormais à operação do sistema. É claro que os meios de controle de integridade devem, também neste caso, ser acionados para garantir a consistência do BD. Note que o processo de confirmar uma transação também envolve os mecanismos de controle de integridade, pelo menos para registrar que este fato ocorreu, evitando que uma transação já confirmada tenha seus efeitos removidos quando de uma eventual falha posterior do sistema. Finalmente, ao término da transação, o gerente de transações deve garantir que todos os recursos apropriados temporariamente por esta transação voltem ao controle do SGBD, sendo postos a disposição de outras transações.

Em um cenário distribuído os mecanismos para *terminar* uma transação não são simples. De alguma forma todos os nós que se envolveram com a transação devem refletir ou remover *todos* os efeitos das ações elementares executadas em favor da particular transação. Quando do término da transação, portanto, protocolos especiais devem ser ativados para garantir a atomicidade da transação como um todo, mesmo na presença de falhas repetidas e imprevisíveis que podem afetar a operação normal de vários dentre os nós participantes. O leitor pode imaginar as dificuldades enfrentadas por um processo distribuído cuja missão seja obter o acórdão detodos os nós envolvidos quando uns e outros podem ser acometidos das falhas as mais variadas. No caso centralizado este complicador simplesmente inexiste.

O problema referente à confirmação/cancelamento de transações é fundamental. Em BDs distribuídos os nós participantes devem adotar um protocolo comum para que o processo se desenvolva de forma ordenada. Uma possibilidade seria adotar o *protocolo bifásico para confirmar intenções*, ou simplesmente *protocolo bifásico*. Existem várias versões deste protocolo, diferindo mais em aspectos de implementação que de filosofia básica. Todas seguem o esquema geral abaixo. Suponha que uma transação distribuída chegou ao ponto onde deve ser confirmada ou cancelada. Um dos nós participantes, por exemplo o nó de origem da transação, é estabelecido, "a priori", como o *coordenador* da operação. Este coordenador é conhecido dos demais nós e também dispõe da identidade de todos os nós participantes. Isto não é difícil de ser conseguido durante a fase em que a transação migra de um nó para outro e dados são enviados de volta aos nós requisitantes.

NUMA PRIMEIRA FASE

- coordenador: envia, a todos os nós participantes, mensagens na forma PREPARE-SE.
- cada nó participante

    1. ao receber uma mensagem de PREPARE-SE vinda do coordenador, tenta registrar em memória estável, isto é, de forma que sobreviva à eventuais falhas do sistema, os efeitos locais da transação

    2. a seguir, envia ao coordenador mensagens na forma PREPARADO caso tenha conseguido o registro em memória estável, ou na forma IMPOSSIBILITADO em caso contrário

NUMA SEGUNDA FASE

- coordenador

    1. envia a todos os nós participantes a mensagem CONFIRME, caso deseje confirmar a transação e todos os nós participantes tenham remetido a mensagem PREPARADO na primeira fase;

    2. em qualquer outro caso, o coordenador envia a mensagem CANCELE a todos os nós participantes

- cada nó participante: ao receber o veredito do coordenador, procede de acordo com este.

Se nenhum dos nós participantes, o coordenador ou os canais de comunicação forem acometidos por falhas durante o processo, é fácil de ver que o protocolo funciona a contento. O Capítulo 7 contém mais detalhes sobre o método, bem como explora outras variantes e filosofias que visam solucionar o problema, garantindo sempre a propriedade de atomicidade inerente ao conceito de transação. Lá também são examinados mecanismos para se lidar com as questões da migração e término de transações. Isto conclui a nossa descrição do processamento de transações neste capítulo.

6.3 FALHAS NO SISTEMA

Dada a complexidade dos equipamentos e programas modernos, é ponto pacífico que falhas ocorrerão, quer sejam problemas de "hardware", quer sejam defeitos de "software". Estas falhas têm como efeito indesejável comprometer a *integridade* do BD. Para que seja de alguma utilidade prática, O SGBD deve, portanto, incorporar mecanismos que garantam sua integridade, quando não pelo menos na presença daquelas falhas que ocorrem com mais freqüência. Desta forma, o SGBD pode ser mantido em operação por longos períodos de tempo sendo, quando muito, interrompido por curtos intervalos para que os mecanismos de controle de integridade sanem inconsistências causadas por eventuais falhas. Esta seção sevirá de introdução à área de *controle de integridade* em SGBD, onde os principais problemas e as soluções mais importantes serão mencionadas de forma simplificada. Em capítulo posterior, esta problemática será analisada em maiores detalhes.

Intuitivamente, a única maneira do SGBD se proteger contra falhas, que podem destruir parte dos dados, é criar e manter certa *redundância* no sistema. Desta forma, quando parte do BD é

danificado, sua "cópia redundante" pode ser revivida para recuperar os dados perdidos e restabelecer a operação normal. Inclusive, em sistemas que requeiram alta confiabilidade, as partes mais críticas do próprio "hardware" podem ser duplicadas de forma redundante. Note que, se a "cópia" de um objeto não é recente, então deve-se manter também um *histórico* das operações efetuadas sobre este objeto, de tal modo que o SGBD possa *refazer* estas operações e trazer esta "cópia" ao estado mais recente, idêntico àquele da "cópia" original antes da falha. Caso contrário, transações executadas entre o instante de criação da "cópia" e o momento atual serão perdidas.

6.3.1 Tipos de Falhas

Um nó qualquer, quando em operação normal, depende de um padrão de interligações complexas entre vários elementos. Para efeito de examinar a ocorrência e danos causados por falhas nestes componentes é conveniente agrupá-los da seguinte forma

- procedimentos
- processadores
- memórias
- no caso distribuído, comunicação de dados

Por *procedimentos* entende-se a totalidade dos módulos e programas aplicativos ("software") que compõem o SGBD, podendo-se incluir aqui também os utilitários do sistema operacional usados pelo SGBD. Os *processadores* correspondem tanto ao processador, ou processadores, central como as demais unidades de controle de periféricos, terminais, "modems", etc. As *memórias*, onde reside o BD, aqui entendido como dados mais programas, são de crucial importância. É lá que será acomodada toda redundância introduzida para fins de controle de integridade. Todos os mecanismos de proteção contra falhas prestam especial atenção ao tratamento dispensado aos vários tipos de memórias com que o sistema interage e, em última análise, se fiarão na boa característica de resistência a falhas que tais elementos oferecem. Para que se possa conduzir uma análise mais detalhada, as memórias manipuladas pelo sistema serão subdivididas em:

- memória principal
- memória secundária imediatamente disponível
- memória secundária dormente

A *memória principal* corresponde a memória associada aos processadores, isto é, memória dos processadores centrais, memórias tipo cache, "buffers" de entrada/saída, espaço de paginação, etc. Há certos tipos de falhas às quais o conteúdo da memória principal não sobrevive, devendo ser considerado como irremediavelmente perdido. Estes defeitos serão cognominados de *falhas primárias*. Interrupção no fornecimento de energia elétrica, defeito nos processadores ou procedimentos do sistema podem causar este tipo de falha. Por não sobreviver a este tipo de falha mais comum, diz-se que a memória principal é *volátil*.

O termo *memória secundária imediatamente disponível*, ou *memória secundária ativa*, referem-se à memória de massa, geralmente discos magnéticos, onde o BD é residente e que está a disposição do SGBD a todo instante. O conteúdo da memória secundária ativa não é afetado por falhas primárias que o sistema venha a sofrer. Porém, panes nos cabeçotes de leitura/escrita dos discos ou partículas de poeira que assentem sobre a superfície dos mesmos podem danificar os delicados mecanismos dos cabeçotes e provocar danos irrecuperáveis à superfície magnética destruindo, em todo ou em parte, o conteúdo da memória secundária

ativa. Isto se verificando, diz-se que o sistema sofreu uma *falha secundária*. Fitas magnéticas também podem ser usadas como memória secundária ativa, se bem que cuidados devem ser tomados para que a eficiência do sistema não seja por demais comprometida. Partes raramente usadas do BD, cópias antigas de parte ou da íntegra do sistema, além de aplicativos ativados com pouca freqüência, são candidatos naturais a residirem em fita. Nunca dicionários, catálogos e outros elementos freqüentemente acessados.

Como um último recurso, e em casos realmente catastróficos, o controle de integridade do SGBD pode apelar para a memória secundária dormente ("off-line"). Entende-se como *memória secundária dormente* toda memória fisicamente desconectada do sistema. Geralmente, devido à sua grande capacidade de armazenamento de dados, fitas magnéticas são empregadas para este fim. Em BD de grandes proporções, toda uma fitoteca pode ser necessária. É comum armazenar os componentes da memória secundária dormente em locais próprios, distantes do centro onde opera o sistema. A preocupação básica é evitar que catástrofes sobre um dos lugares, tais como incêndios ou furtos, não afete o outro. De qualquer maneira, eventos que destruam ou inutilizem o conteúdo da memória secundária dormente do sistema serão chamados de *falhas terciárias*.

Existem situações que exigem ações por parte do sistema de controle de integridade do BD, embora não se configurem propriamente como falhas em componentes do sistema. O caso típico é quando, sob operação normal, surje a necessidade de se cancelar transações. Isto pode ocorrer tanto por erro ou a pedido do usuário, como podem ser ações forçadas pelo SGBD como última instância para evitar bloqueio na execução de transações que competem por certos recursos do sistema. Estes casos serão rotulados como *pseudo-falhas* do sistema. Agora, não está em cheque o conteúdo de nenhuma das memórias ou a sanidade dos processadores ou procedimentos associados ao sistema. A cooperação do controle de integridade, porém, é necessária para invocar a transação que inverte o efeito das ações elementares executadas em benefício da transação a ser cancelada. Deste modo, o BD permanece em um estado consistente, além do que é forçada a liberação dos recursos que foram seqüestrados pela transação.

A discussão acima desloca-se a partir de falhas nos componentes mais nobres, isto é, com menor tempo de acesso, para os menos nobres, com tempos de acesso consideravelmente maiores. É importante ter em mente uma noção da freqüência com que os vários tipos de falhas costumam ocorrer na prática, bem como do tempo necessário para que o controle de integridade restaure a operação normal do BD em cada caso.

A Figura 6.1 resume a situação.

TIPO DA FALHA	**FREQÜÊNCIA**	**TEMPO DE RECUPERAÇÃO**
Pseudo-falha	várias por minuto	milisegundos
Primária	várias por mês	segundos
Secundária	várias por ano	minutos
Terciária	uma por século	dias

Figura 6.1 - Características de Falhas

Está implícito aqui que o SGBD pode, ao ser reconduzido à operação normal, detectar que uma falha primária causou a interrupção das operações. Ao voltar à vida, o SGBD acionaria o

controle de integridade para levar o BD a um estado consistente. Falhas secundárias também seriam detectadas pelo SGBD que interromperia momentaneamente suas operações normais e acionaria o controle de integridade. A deteção de falhas terciárias ficaria a cargo de operadores externos ao SGBD os quais, uma vez verificado o problema, rodariam processos específicos que eventualmente restaurariam a memória dormente do sistema.

Outra suposição importante diz respeito a freqüência ou probabilidade de cada tipo de falha. Será sempre suposto que os vários componentes do sistema têm modos de falha independentes, de tal modo que são praticamente negligenciáveis as chances de que ocorrerão efeitos cascata com uma falha provocando outra. Mais ainda, a probabilidade de dupla falha, isto é, de falhas simultâneas, torna-se desprezável.

No caso de BD distribuídos, há ainda o complicador adicional de que a rede comunicação de dados ou os protocolos de comunicação de dados podem, um ou ambos, falhar. Isto ocorrendo, diz-se que o sistema sofreu uma *falha de comunicação de dados* ou simplesmente uma *falha de comunicação*. Como alertado no Capítulo 1, o sistema de comunicação de dados será suposto perfeito do ponto de vista de que mensagens não são perdidas, danificadas ou entregues a destinatários errados. Existem mecanismos próprios para se atingir tais objetivos, mecanismos estes que fogem ao escopo deste texto. Sob a ótica do SGBD, porém, estes processos são totalmente transparentes. O SGBD apenas coloca mensagens, isto é, endereços mais conteúdos, nos "buffers" de saída e supõe que mensagens podem materializar-se em seus "buffers" de entrada. O primeiro complicador encontrado diz respeito à detectabilidade de falhas de comunicação. Torna-se extremamente difícil, se não impossível, para um particular nó da rede descobrir se o outro nó, com o qual deseja se comunicar, não responde porque a comunicação foi cortada por uma falha de comunicações ou se o outro nó está simplesmente sobrecarregado e, portanto, muito lento ao responder. Note que um dado nó não pode simplesmente continuar a retransmitir mensagens, sem tomar nenhum cuidado adicional, até que o nó destinatário finalmente se manifeste. Sob este cenário, uma mesma transação poderia executar repetidas vezes no ambiente remoto.

Focalizando o problema de falhas sob o ponto de vista dos objetos envolvidos, pode-se caracterizá-los como objetos *voláteis, estáveis* ou *reais*.

No primeiro grupo situam-se os objetos cujos valores não sobrevivem a falhas primárias. O exemplo típico é dado pelos objetos que residem na memória primária do sistema. No grupo intermediário estão todos os objetos cujos valores sobrevivem a falhas primárias mas não a falhas secundárias. Aqueles objetos residindo em memória secundária imediatamente disponível classificam-se como tal. Finalmente, no terceiro e último grupo são encontrados objetos cujos valores, uma vez modificados, escapam ao controle do SGBD. Numa transação envolvendo um caixa bancário automático, uma vez dispensando um certo número de cruzeiros, estes cruzeiros existem e estão fora do alcance do SGBD.

A próxima subseção abordará, de forma simplificada, os mecanismos usados para garantir o controle de integridade do BD enquanto que o Capítulo 9 detalhará as estruturas e algorítmos usados nos métodos mais importantes para efetivar esta função.

6.3.2 Proteção Contra Falhas

Como foi visto na subsecção anterior, os mais variados tipos de falhas podem se abater sobre o BD, cobrindo todo um espectro que vai desde situações amenas e imediatamente contornáveis até catástrofes que deixam marcas irrecuperáveis, destruindo dados e procedimentos do sistema. O problema é agravado pela imprevisibilidade das falhas e, ainda,

pelo fato de que novos desarranjos podem advir enquanto o SGBD está se recuperando de outros anteriores, caracterizando múltiplas falhas simultâneas. É importante notar, desde já, que não há proteção total e cabal contra todas as falhas que eventualmente podem vir a perturbar o sistema. O que os mecanismos de proteção visam é tornar o SGBD mais e mais confiável e seguro a medida que se tornam mais sofisticados. No entanto, casos patológicos sempre poderão ser imaginados de tal forma que não possam ser tratados a contento pelos mecanismos de proteção. Não deve ser esquecido, também, o aspecto da eficiência do SGBD como um todo. Dispor de um sistema extremamente seguro e confiável, porém a um custo absurdamente alto, inviabiliza o próprio SGBD, na medida em que não será posto em operação devido a sua impraticabilidade. Todo o desafio está em se imaginar técnicas, estruturas e procedimentos que dêem proteção razoavelmente eficaz contra as falhas mais observadas na prática e que sejam, por outro lado, suficientemete expeditos de modo a não causar impacto por demais negativo nas atividades do usuário.

Os vários mecanismos de proteção podem também ser dicotomizados segundo propiciem ou não certa robustez contra falhas. Muitos mecanismos provêem simplesmente proteção aos dados e procedimentos do SGBD, isto é, ao ocorrer uma falha, o SGBD impede a utilização do BD enquanto aciona os mecanismos de proteção de que dispõe na tentativa de restaurar um estado consistente quando, então, permite novamente que usuários continuem seu trabalho. Outros métodos detetam a ocorrência de certos tipos de falhas e, de forma transparente ao usuário, ativam mecanismos corretores os quais reparam os eventuais danos causados, propiciando uma certa robustez a estes tipos de falhas.

Na presente subsecção serão abordados, a nível descritivo, os métodos mais importantes para controle de integridade. Alguns são de concepção muito simples e dispensam maiores comentários. Outros, mais sofisticados, serão detalhados no Capítulo 9. Dada a pletora de métodos hoje desenvolvidos, a classificação abaixo é um tanto arbitrária e algumas das técnicas poderiam ser reagrupadas de formas diferentes. Mais ainda, muitos dos métodos só garantem a integridade do BD contra certos tipos de falhas. É comum, portanto, que o SGBD disponha de várias estratégias a seu alcance e procure usá-las de forma orquestrada visando cobrir um amplo espectro de falhas. Também, controlar vários métodos distintos permite ao SGBD criar um certo sinergismo ao explorar particularidades dos métodos e, assim, torná-los mais eficientes e eficazes do que se operados individualmente.

Programas Restauradores

Uma das primeiras estratégias que vêem a mente incorpora o conceito de um *programa restaurado*r. Estes programas simplesmente revêem todos os dados do BD após a ocorrência de um desarranjo e tentam salvar o que for possível. Arquivos podem ser perdidos no processo se o algoritmo decidir que não pode recuperá-los. O programa tenta, entretanto, deixar o BD em um estado consistente, embora não completo, no sentido de que os dados sobreviventes satisfazem os requisitos de consistência do BD enquanto não há garantias de que parte dos dados não sejam eliminados.

A situação típica quando programs restauradores são ativados se dá em SGBDs onde os algorítmos empregados em operação normal não garantem a consistência dos dados na memória secundária ativa, a cada instante. Se o SGBD é acometido por uma falha primária quando está justamente no processo de transferir dados da memória principal para a memória secundária ativa o conteúdo da primeira é destruído, impedindo que a operação seja

completada. Neste ponto, a memória secundária ativa, ainda não tendo sido completamente atualizada, representa um estado inconsistente do BD. O programa restaurador vai, então, examinar os arquivos da memória secundária ativa e tentar, de alguma forma, torná-los consistentes. É fácil de imaginar situações onde os dados da memória principal, destruídos quando da ocorrência da falha, sejam de tal forma críticos que a única alternativa seria o programa restaurador eliminar parte dos dados residentes na memória secundária ativa a fim de trazer todo o DB a um estado consistente.

Programas restauradores normalmente entram em ação como um último recurso ou na ausência de mecanismos apropriados para contornar certos tipos de falhas. Embora seja um mecanismo primitivo de controle de integridade, o programa restaurador pode se tornar atraente em BDs simples que implementem estruturas pouco sofisticadas. Isto é tanto mais verdade quando os dados que possam eventualmente vir a serem perdidos estiverem disponíveis em outras formas, de tal modo que as operações possam ser rapidamente refeitas, recuperando todo ou boa parte do que o programa restaurador não conseguiu salvar. Uma descrição mais detalhada destes programas dependeria fortemente da particular concepção de cada BD e, por isto, não será tentada neste texto. Há, porém, sistemas reais que se valem deste recurso em maior ou menor extensão. O leitor encontrará exemplos de tais sistemas na Figura 6.2 e nas notas bibliograficas no final do capítulo.

Descargas

Aqui, novamente, a idéia básica é bastante simples. Periodicamente, o conteúdo de toda a memória secundária ativa é *descarregado* (evitando o anglicismo "dumped") para outros dispositivos ativos, geralmente fitas magnéticas, que são em seguida arquivadas como memória secundária dormente. Ocorrendo uma falha secundária, e outros mecanismos de restauração mais eficientes não operando a contento, o SGBD sempre pode apelar para esta cópia arquivada e retornar a um estado consistente, anterior, do BD. Apenas falhas terciárias podem afetar as cópias arquivadas. Note que se estas se constituem nos únicos objetos de que se pode valer o SGBD para voltar à normalidade, então todas as transações invocadas desde o início da descarga e o momento de ocorrência das falhas serão perdidas. Nenhum traço de sua existência, mesmo daquelas que foram confirmadas, será notado após a recuperação do sistema. Em muitos casos, isto pode ser intolerável. Métodos apoiados em descargas do BD são, portanto, candidatos naturais para se aliarem a outras estratégias que tentam remediar este defeito, mantendo algum tipo de registro das atividades que ocorrem no hiato entre a obtenção de duas descargas consecutivas. Uma variação óbvia consiste em se descarregar não todo o BD, mas apenas as partes do BD que tenham sido afetadas desde a última descarga. Este processo é conhecido como *descarga incremental*. Descargas, como técnica de controle de integridade isolada, não serão descritas em maiores detalhes. Entretanto, alguns métodos mais sofisticados que empregam descargas, de uma forma ou outra, serão pormenorizados no Capítulo 9, onde, então, é tornado explícito o papel desempenhado pelas descargas no método como um todo.

Note que a periodicidade com que as cópias são obtidas deve ser analisada com cuidado. Quanto mais freqüentes, tanto mais eficazes serão no combate a falhas secundárias, visto que representam um estado mais recente do BD em relação ao momento de ocorrência da falha. Por outro lado, maior freqüência implicará em maior degradação do tempo de resposta do sistema, especialmente se o BD for de porte avantajado. O ajuste final deverá ser feito levando-se em conta quão susceptível é o sistema aos vários tipos de falhas combatidas através destas técnicas.

Há duas maneiras principais de se obter descargas, segundo o SGBD interrompe ou não suas atividades normais enquanto a descarga é obtida. Um método válido seria o SGBD determinar, num dado momento, que por ora não aceitaria novas transações. Após aguardar que todas as transações iniciadas antes deste instante fossem completadas, o SGBD acionaria os procedimentos próprios para obter a descarga desejada. Em seguida, retornaria a operação normal aceitando novas transações. Este processo será rotulado de *descarga estática*. Como alternativa, o próprio SGBD poderia eleger algumas das transações correntes para serem canceladas, evitando esperar que completassem. No caso distribuído esta alternativa deve ser abordada com cautela, visto que cancelar transações que migraram para outros nós pode onerar sobremaneira o sistema. A principal desvantagem está, claro, no fato do sistema se tornar indisponível entre o início e término da descarga. Em contrapatida, pagando-se em complexidade quanto aos algorítmos usados, pode-se elaborar mecanismos que obtenham uma *descarga dinâmica* do BD. A cópia é obtida dinamicamente, com o sistema ainda disponível, talvez com um tempo de resposta ligeiramente dilatado. Os algorítmos devem garantir, entretanto, que o BD é congelado em um dado instante *T* de tal modo que apenas transações confirmadas antes de *T* terão seus efeitos refletidos na cópia obtida, enquanto que não é deixado traço algum das demais. No Capítulo 9 ambos estes mecanismos serão examinados em maiores detalhes, visto que integram outros métodos de controle de integridade mais elaborados.

Arquivos Diferenciais

Esta técnica adota a filosofia de coletar todas as modificações efetuadas contra certas entidades do BD, em particular arquivos de dados, em estruturas próprias, especificamente planejadas para este fim, mantendo os valores dos objetos originalmente associados a estas entidades inalterados. Estas estruturas, onde as modificações são agrupadas, receberão o cognome de *arquivos diferenciais*. Tudo se passa como se o SGBD, ao concentrar modificações sobre os arquivos diferenciais, atrasasse o momento de tornar irreversível o efeito de ações elementares sobre o BD, mesmo daquelas correspondentes a transações que já foram confirmadas. A vantagem pretendida é óbvia. Os originais formam, naturalmente, uma cópia que reflete um estado completo e consistente anterior do BD. Ocorrendo uma falha que venha a prejudicar os arquivos diferenciais tudo que o SGBD tem a fazer é reler da memória secundária ativa os originais intactos, eventualmente disperdiçando os trabalho das últimas transações invocadas. Porém, se o usuário já foi informado de que derminada transação confirmou, ignorá-la neste ponto pode bem ser inadmissível. Para maior segurança, os originais e os respectivos arquivos diferenciais podem ser mantidos em dispositivos físicos distintos. Descarregando os originais em memória secundária dormente protejerá o BD contra quaisquer falhas secundárias.

No entanto, em algum ponto no tempo, os arquivos diferenciais terão que ser usados para reconstruir os originais, processo este que pode ser oneroso. Mais ainda, ao acessar dados o SGBD deve pesquisar, antes, se os objetos desejados não se encontram nos arquivos diferenciais, incorrendo, assim, numa perda de eficiência. Duplicar os arquivos diferenciais em memória secundária ativa, em unidades físicas diferentes, pode, portanto, degradar ainda mais o sistema a ponto de inviabilizar a idéia na maioria dos casos. Alguns mecanismos engenhosos e técnicas especiais usadas na organização dos arquivos diferenciais permitem minimizar este problema. Note, em especial, que ações elementares que remodificam o valor de um objeto que já está no arquivo diferencial não necessitam de criar um novo registro para este particular objeto. Basta alterar o conteúdo do registro que lá se encontra. Ações elementares cujo efeito seja a remoção de um objeto são igualmente simples de serem implementadas. A seu favor, este método tem o fato de que facilita bastante a operação de

obter-se descargas incrementais do conteúdo do BD pois basta descarregar os arquivos diferenciais os quais são, em princípio, bem menores que os originais.

Atas

Imagine um SGBD onde o conceito de transação permeia toda sua organização, isto é, transações são as unidades básicas de trabalho do sistema. Falhas, ocorrendo em momentos imprevisíveis, colherão algumas transações em andamento, porém ainda não completadas, isto é, não confirmadas ou canceladas. Ao voltar a cena, o SGBD deve, portanto, desfazer todos os efeitos parciais registrados por cada transação que ainda não completou até o instante de ocorrência da falha. Para que isto possa ser levado adiante, cada ação elementar deve registrar, em uma *ata* (em inglês "log" ou "audit trail"), os valores iniciais e finais de cada objeto modificado, bem como deixar claro em benefício de que transação esta ação elementar foi invocada. Normalmente, a ata reside na memória secundária ativa. Ocorrendo uma falha primária, tudo que o SGBD precisa fazer é consultar a ata e

- *refazer* transações confirmadas cujo efeito ainda não tenha sido registrado na memória secundária ativa;
- *desfazer* transações canceladas cujo efeito já foi registrado em memória secundária ativa;
- *desfazer* transações que ainda estavam em andamento quando do instante de ocorrência da falha.

Note que também deve ir para a ata um registro de início e término para cada transação executada. Assim, os mecanismos de controle de integridade do SGBD teriam condições de identificar todas as transações em andamento a cada instante, bem como não teriam dificuldades em agrupar e associar ações elementares que correspondam a uma mesma transação. A própria ordem seqüencial em que aparecem os registros de ações elementares na ata se presta muito bem a refazer/desfazer transações.

Observe que atas devem residir em memória secundária ativa, visto serem estruturas muito solicitadas uma vez que a invocação de cada ação elementar deixa lá um registro. Já uma falha que destrua parte da memória secundária ativa, afetando segmentos da ata, provocaria uma catástrofe no sistema. Transações confirmadas cujos efeitos não foram ainda registrados sobre os respectivos objetos simplesmente desapareceriam, uma vez ser a ata o único registro destas transações capaz de fornecer informações suficientes para que sejam refeitas. Atas propiciam um meio de se restaurar o BD ao estado completo mais recente com relação ao ponto de ocorrência da falha. A se manter este requisito, o dilema está em se imaginar estruturas eficazes que o satisfaçam, mesmo na presença de falhas secundárias que podem destruir parte da ata. Uma alternativa é duplicar a ata em memória secundária ativa de tal modo que as duas "cópias" residam em dispositivos físicos distintos. Como estamos supondo que a ocorrência de falhas são eventos independentes, isto minimizaria as chances de que partes da ata sejam perdidas. O preço, claro, seria comprometer o tempo de resposta ainda mais. Em princípio, pode-se tornar o mecanismo tão confiável quanto se queira através da duplicação, triplicação, etc., da ata em unidades distintas.

Um problema mais sutil diz respeito ao sincronismo entre a entrada de registros na ata e a alteração de valores de objetos na memória secundária ativa. Um bom procedimento seria adotar um protocolo para uso da ata que garantisse a catalogação de cada ação elementar antes de registrar seus efeitos em memória secundária ativa. Para apreciar que tipo de problemas podem surgir, considere a seqüência de passos abaixo, usualmente seguidos pelo

SGBD ao manipular objetos do BD local:

1. determina se a página onde reside o objeto em questão está presente na memória principal;

2. se não estiver, aciona o sistema operacional local para que este leia a página necessária da memória secundária ativa para a memória principal;

3. perfaz as alterações desejadas no valor do objeto, reconstruindo a página em que este reside, na memória principal;

4. torna a acionar o sistema operacional local para que este reescreva a página atualizada da memória principal para a memória secundária ativa, quando então as modificações se tornam permanentes;

5. registra a ocorrência desta ação elementar na ata, além de associá-la à transação correspondente.

Suponha que uma falha colha o sistema no hiato entre os passos 4 e 5 da seqüência acima. Neste caso, a situação se tornaria confusa pois o SGBD não teria condições de inverter a ação elementar que alterou o conteúdo desta página, dado que não haveria registro algum de sua invocação.

Observe que, se tomada isoladamente, a ata teria que registrar todas as ações elementares efetuadas contra o BD desde que este foi criado. Esta situação é remediada com o emprego de descargas. A última descarga obtida seria revivida e, ao se ler a ata, todas as transações que porventura completaram antes da descarga ter sido iniciada poderiam ser ignoradas. As demais seriam refeitas ou desfeitas, de acordo com a respectiva situação no instante de ocorrência da falha. Como pode ser visto, descargas complementam naturalmente técnicas de controle de integridade que operam através de atas. Neste caso específico, descargas serão conhecidas por *balizadores* (do termo em inglês "checkpoints"). Uma vez sabendo-se que as descargas operarão em estreita cooperação com as atas, é possível otimizar o tempo de resposta quando as primeiras estão sendo obtidas, dispensando-se certas informações redundantes.

Na realidade, a situação é um pouco mais complexa do que pode deixar transparecer a discussão acima. Detalhes do mecanismo serão examinados no Capítulo 9.

Imagens Transientes

Imagine que o BD seja composto por um certo número de páginas de tamanho fixo e que a memória secundária está dividida em setores do mesmo tamanho que as páginas.

Conforme já foi ressaltado anteriormente, ações elementares manipulam objetos do BD, que serão identificados, para propósito desta discussão, com as páginas de memória. Uma transação modificaria, possivelmente, um certo número destas páginas, caso confirmada. É claro que quaisquer alterações na memória secundária ativa só podem ser efetivadas se a página for construída na memória principal e, posteriormente, reescrita em memória secundária ativa. Quando a página modificada é repassada para a memória secundária ativa, porém, tem-se duas alternativas:

1. reescrever a página reconstruída sobre o setor que ocupava inicialmente, destruindo, por conseguinte, seu conteúdo original;

2. reescrever a nova página sobre um setor livre em memória secundária ativa, mantendo seu conteúdo original inalterado.

As técnicas de controle de integridade examinadas até o momento operavam na hipótese de que o SGBD adotava o mecanismo do item 1 acima. Idéias que exploram a metodologia do item 2 seguem o esquema geral abaixo.

Ao modificar a página proceda da seguinte forma:

1. leia a página *i* da memória secundária ativa para a memória principal, caso esta lá ainda não se encontre; seja *k* o endereço desta página na memória principal;

2. modifique, na memória principal, valores de dados residentes nesta página;

3. encontre um setor livre, seja *s*, na memória secundária ativa;

4. reescreva a página *i* na memória principal sobre o setor de endereço *s* na memória secundária ativa;

5. altere o diretório de páginas, indicando que a página *i* se encontra agora no setor *s*;

6. libere o setor ocupado originalmente pela página *i*.

Visto de um certo ângulo, estas técnicas criam uma *imagem transiente* com as modificações desejadas, enquanto mantêm a imagem certificada inalterada. Contrastando com a técnica de arquivos diferenciais, porém, as imagens transientes são efêmeras, substituindo, imediatamente após escritas na memória secundária ativa, as imagens originais a partir das quais foram criadas. Obviamente, no caso do sistema sofrer uma falha primária durante a execução dos passos acima, as imagens originais estariam ainda intactas, além de representarem um estado consistente recente do BD.

Serão descritas duas maneiras de se implementar os mecanismos de imagens transientes: através de vetores de ponteiros e através de listas de intenções.

No esquema de *vetor de ponteiros* o BD é suposto formado por um conjunto de segmentos divididos em páginas. O i-ésimo segmento dispõe de um vetor de ponteiros, V_i , que dá os setores que contêm as suas páginas. Afora estes vetores, o mecanismo usa 2 vetores de "bits". Em um deles, *MAP*, o i-ésimo "bit" representa a situação do i-ésimo setor da memória secundária: 1 equivale a ocupado e 0 a livre. No outro vetor, *EST*, o j-ésimo "bit" indica a situação do i-ésimo segmento: 1 se há transações manipulando páginas deste segmento e 0 em caso contrário. Alterar a j-ésima página do i-ésimo segmento corresponde a

- verificar se o i-ésimo segmento já está sendo manipulado, consultando o "bit" *EST(i);*
- se não, copiar o diretório do i-ésimo segmento em outro vetor, V'_i , em memória secundária;
- se preciso, trazer o setor de endereço $V_i(j) = s$ para a memória principal;
- efetuar as modificações desejadas na página residente na memória principal;
- procurar um setor livre na memória secundária ativa, consultando o vetor *MAP*. Seja *s'* o endereço do setor encontrado;

- reescrever a página modificada, na memória principal, para o setor de endereço *s'*, na memória secundária ativa;
- preparar o setor de endereço *s'* para substituir o original, no endereço $V_i(j)$, intercambiando os valores de $V_i(j)$ e *s'* ;
- por motivo de segurança, copiar o vetor *MAP* para outro vetor, *MAP'*;
- efetivar as alterações mudando *MAP(s)* para 0 e *MAP(s')* para 1;

Observe que, nos passos 2 e 8, são tomadas precauções para que, em caso de falha, o SGBD possa restaurar seu estado anterior simplesmente copiando os vetores *MAP'* e V'_i sobre os vetores *MAP* e V_i, respectivamente.

Uma variação deste primeiro esquema será discutida em detalhe na Seção 9.2.

No segundo esquema, o processo é ligeiramente diferente. Suponha que cada arquivo *Q* do banco esteja dividido em páginas e que o banco possua um diretório *D* indicando que setores contêm as páginas de *Q*. Uma transação, *T*, prossegue normalmente, alterando páginas de acordo com o mesmo princípio de criar imagens transientes destas páginas. Suponha que a transação manipulou dados e que, para isto, alterou páginas de *Q* com endereços $P_1, \ldots, P_n$. O endereço do setor contendo a imagem original da página P_k é i_k e o endereço do setor contendo a imagem transiente de P_k é j_k . Para finalizar, deve-se executar as ações $A_k = D(k) := j_k$, representando as operações de atribuição necessárias à atualização de *D*. A lista de ações $L = (A_1, \ldots, A_n)$ é chamada de *lista de intenções* para *T*.

Esquematicamante, *T* executa de acordo com o plano abaixo.

1) Na fase de execução

   a) *T* invoca cada uma de suas ações elementares, criando imagens transientes de $P_1, \ldots, P_n$

   b) *T* cria, no processo, a lista de intenções *L*

2) Na fase de confirmação

   a) *T* escreve *L* em memória secundária ativa

   b) *T* executa *L*

   c) Libera o espaço ocupado por *L* em memória secundária ativa.

Portanto, se falhas primárias ocorrerem antes do passo 2, *T* é simplesmente ignorada quando o SGBD, ao retornar à vida, se depara com o diretório *D* inalterado em relação ao estado anterior. Após o passo 2a, *T* será confirmada e falhas primárias não podem alterar este fato.

Uma propriedade muito desejável está, obviamente, embutida no mecanismo: repetidas execuções de *L*, mesmo parciais, surtirão efeito equivalente a uma única execução de *L*. Isto assegura que, se o sistema for acometido por uma falha primária enquanto ainda está se recuperando de outra anterior, a transação será, sempre, corretamente confirmada.

O caso de um BD distribuído apresenta complicações características. Um determinado nó deve substituir as imagens certificadas pelas transientes ou, conforme o caso, executar a lista de intenções de uma transação, se e somente se todos os demais nós que atuaram em

benefício da transação atingirem um concenso neste sentido. Aqui poder-se-ia usar o protocolo bifásico para se atingir um entendimento entre os nós participantes.

Muitas variações podem ser construídas usando-se as mesmas idéias básicas. Por exemplo, manter-se duas cópias de cada arquivo do BD a cada instante, uma delas servindo de reserva caso a outra seja danificada por falhas. Modificações seriam refletidas em ambas as cópias imediatamete após cada ação elementar executada. Para evitar inconsistências por ocasião de eventuais falhas, cada cópia disporia de um "bit" cuja função seria indicar que a operação de atualização está em progresso sobre a respectiva cópia, evitando que esta fosse usada como a cópia de reserva a ser usada quando o SGBD se recuperasse de falhas. Assumindo que este "bit" sempre resiste a todos os tipos de falhas que se pretende contornar com este mecanismo, o SGBD sempre resgataria um estado consistente anterior quando voltasse à operação normal após uma falha. Esta e outras variações possíveis não serão discutidas em maiores detalhes no texto que segue.

6.3 CONTROLE DE CONCORRÊNCIA

Um SGBD, suportando bancos de dados com várias aplicações, deverá necessariamente permitir acesso concorrente aos dados. É intuitivo que, em tese, quanto maior for o nível de concorrência permitido, tanto melhor será o tempo de resposta do sistema como um todo. Em tese porque, forçosamente, os mecanismos que controlam o acesso concorrente ao banco impõem um ônus adicional sobre o desempenho do SGBD. Os procedimentos que harmonizam o paralelismo no seio do SGBD serão conhecidos por *mecanismos de controle de concorrência*.

Num cenário de BDs distribuídos, ou mesmo de BDs centralizados com acesso distribuído, a implementação de paralelismo torna-se uma necessidade imperiosa. Com relação ao caso centralizado, hoje são conhecidas técnicas que equacionam os problemas a contento, apoiadas em um tratamento teórico preciso e confirmadas por implementações reais. No que concerne ao caso distribuído, a situação é mais confusa. Isto é devido, em grande parte, ao fato de que os nós da rede operam de forma bastante independente, embora o controle de concorrência deva ser efetivado de forma global, abrangendo informação que pode estar espalhada por vários nós. Aqui, muitos dos aspectos do problema ainda se encontram em fase de pesquisa e experimentação.

Nesta seção serão apresentados os problemas fundamentais a respeito de controle de concorrência bem como serão descritos, de forma suscinta, alguns algorítmos usados para solucioná-los. No Capítulo 8 estas questões serão examinadas em profundidade.

No que tange ao restante desta seção, será suposto que o sistema nunca falha, a não ser quando explicitamente dito em contrário. Equivalentemente, é suficiente supor que as técnicas para controle de integridade, expostas na seção anterior, sejam adequadas para contornar eventuais falhas de maneira satisfatória e transparente. Na realidade, é claro, ambos os mecanismos devem operar de forma integrada. A hipótese acima é colocada simplesmente para isolar os problemas.

6.3.1 Colocação do Problema

Quando transações manipulam dados concorrentemente, certos problemas, chamados de anomalias de sincronização, poderão ocorrer. Exemplos são acessos a dados inconsistentes, perdas de atualizações e perda da consistência do banco. Por exemplo, considere duas

transações, T_1 e T_2, ambas debitando uma determinada quantia a um saldo *S*. Seja a seqüência de ações:

1) T_1 lê o saldo *S*;

2) T_2 lê o saldo *S*;

3) T_2 debita a quantia, escrevendo o novo valor de *S*;

4) T_1 debita a quantia, escrevendo o novo valor de *S*;

O valor final do saldo neste caso refletirá apenas a quantia debitada por T_1, sendo a atualização submetida por T_2 perdida.

O exemplo acima também serve para ilustrar porque controle de concorrência, embora semelhante, não é equivalente ao dilema de gerenciar acesso a recursos partilhados em um sistema operacional. De fato, na seqüência acima, cada transação respeita o princípio de acesso exclusivo ao objeto partilhado *S*. Porém, isto claramente não é suficiente pois uma atualização é perdida. Assim sendo, um mecanismo de controle de concorrência não deverá se limitar a implementar acesso exclusivo a objetos do banco de dados.

O problema fundamental a ser resolvido pelos métodos de controle de concorrência é colocado da seguinte forma. Assuma que todas as transações preservam a consistência lógica do banco de dados e terminam, quando executadas seqüencialmente. Um método de controle de concorrência deverá, então, garantir que em toda execução concorrente das transações:

1) cada transação termina;

2) cada transação é executada sem interferência das outras, e sem que anomalias de sincronização ocorram.

Estes objetivos deverão ser atingidos permitindo-se um máximo de concorrência possível, de forma transparente para os usuários, e para qualquer conjunto de transações acessando qualquer parte do banco de dados.

Note que o controle de concorrência e a gerência de transações são tarefas que se complementam. Cabe ao gerente de transações escalonar as ações de uma transação de tal forma que esta seja processada corretamente, conforme a especificação do usuário. Por outro lado, cabe ao mecanismo de controle de concorrência arbitrar a intercalação das ações de transações diferentes de tal forma a que todas as transações terminem, sejam processadas sem que uma interfira com a outra e sem que anomalias de sincronização ocorram.

O resto desta subseção abordará em mais detalhe o critério de correção imposto aos mecanismos de controle de concorrência.

A idéia do critério de correção comumente aceito é bem simples. Seja *T* um conjunto de transações. Inicialmente observa-se que uma execução seqüencial ou *serial* das transações em *T*, ou seja, uma execução em que as transações são processadas uma após a outra terminar, em uma ordem qualquer, é necessariamente correta. Isto é fácil ver pois em uma execução serial não há processamento concorrente.

O próximo passo é postular que uma execução concorrente *E* das transações em *T* será considerada correta se for computacionalmente equivalente a alguma execução serial das transações. A execução *E* será chamada neste caso de *serializável.* A noção de equivalência

computacional usada aqui exige que o estado final do banco de dados seja o mesmo em *E* e *S* e que as transações leiam os mesmos dados em *E* e *S* (assumindo que *E* e *S* começam no mesmo estado inicial do banco de dados) e, portanto, executem a mesma computação em *E* e *S*.

A postulação deste critério de correção para execuções concorrentes é justificada pois, como *S* é serial, as transações em *T* são naturalmente executadas sem que uma interfira com a outra. É fácil justificar que anomalias de sincronização não ocorrem em *S*. Como *E* é computacionalmente equivalente a *S*, as transações são executadas em *E* sem que uma interfira com a outra e sem que anomalias de sincronização ocorram. Os próximos exemplos ilustrarão estes conceitos.

Suponha que *P* e *C* representem os saldos da poupança e da conta corrente de um determinado cliente (armazenadas em um banco de dados centralizado, por simplicidade). Considere duas transações cujos efeitos desejados são:

T_1: se houver saldo suficiente na poupança, transfira $5.000,00 da poupança para a conta corrente;

T_2: se houver saldo suficiente na conta corrente, debite um cheque de $10.000,00.

Suponha que a seqüência de ações elementares sobre o banco de dados gerada pela execução da transação T_1 (supondo que haja saldo suficiente) seja:

T_{11}: leia o saldo *P* para *X*;

X := X - 5.000;

T_{12}: leia o saldo *C* para *Y*;

Y := Y + 5.000;

T_{13}: escreva o novo saldo *Y* em *C*;

T_{14}: escreva o novo saldo *X* em *P*.

Apenas aquelas ações que acessam o banco de dados receberam rótulos pois são estas que nos interessarão a seguir. Suponha que a seqüência de ações elementares sobre o banco de dados gerada pela execução da transação T_2 (supondo que haja saldo suficiente) por sua vez seja:

T_{21}: leia o saldo *C* para *Z*;

Z := Z - 10.000;

T_{22}: escreva o novo saldo *Z* em *C*.

Primeiro considere uma execução puramente serial em que T_2 é processada antes de T_1. Esta execução pode ser abstraída pela seguinte seqüência de rótulos das operações sobre o banco de dados:

$$S = T_{21}\ T_{22}\ T_{11}\ T_{12}\ T_{13}\ T_{14}$$

Considere agora uma execução concorrente abstraída pela seguinte seqüência de rótulos:

$$L = T_{11}\ T_{21}\ T_{22}\ T_{12}\ T_{13}\ T_{14}$$

Esta execução *L* não é serial, porém é serializável por ser equivalente a *S*. Isto é fácil de ver pois a ação T_{11} em *L* pode ser comutada com T_{21} e T_{22} sem que o resultado do processamento de T_1 seja alterado e sem que o estado final do banco de dados seja modificado. De fato, T_{11} lê o saldo *P* enquanto as ações de T_2 afetam apenas o saldo *C*. Logo, T_{11} lê o mesmo saldo *P* em *L* e em *S*.

Por fim, considere uma execução concorrente abstraída pela seguinte seqüência de rótulos:

$$N = T_{11}\ T_{12}\ T_{21}\ T_{22}\ T_{13}\ T_{14}$$

Esta execução *N* não é serial e também não é serializável por não ser equivalente nem a *S*, nem a uma execução serial *S'* em que T_1 é processada antes de T_2. De fato, T_{12} lê o saldo inicial *C*, que é posteriormente escrito por T_{13}. Desta forma, a atualização expressa por T_2 é perdida e o banco de dados final (em *N*) não reflete a execução de T_2. Logo *N* não pode ser equivalente à *S* ou à *S'*, ou seja, não é serializável.

6.3.2 Métodos de Controle de Concorrência

Nesta subseção serão discutidos dois dos principais métodos usados para implementar controle de concorrência. O primeiro dos métodos a ser discutido utiliza bloqueio aos dados para garantir que toda execução concorrente é serializável. Já o segundo método impõe uma ordenação "a priori" às transações e garante que toda execução concorrente é equivalente à execução serial das transações na ordem imposta.

Métodos baseados em Bloqueios

Uma parte dos problemas levantados na seção anterior residia no fato de que os estados intermediários gerados por uma transação eram visíveis às outras transações. A idéia básica dos métodos de bloqueio consiste simplesmente em impedir que tais estados se tornem visíveis bloqueando o acesso a eles. A solução não é tão imediata como parece, no entanto. Há dois aspectos envolvidos no uso de métodos de bloqueio que precisam ser equacionados: como usar bloqueios para atingir apenas execuções serializáveis, e como gerenciar o bloqueio aos dados.

Deve ficar claro, neste ponto, que apenas usar bloqueios para gerenciar o acesso aos objetos partilhados não é suficiente para criar um controle de concorrência correto. Por exemplo, considere o seguinte protocolo de bloqueio:

1) antes de ler ou atualizar em um objeto, este deve ser bloqueado;

2) após a leitura ou atualização, o objeto poderá ser liberado.

Este protocolo é obviamente incorreto, pois permite gerar, por exemplo, a execução *N* apresentada no final da seção anterior, que não é serializável.

O protocolo de bloqueios mais comumente usado para garantir serialização é chamado de *bloqueio em duas fases* e dita o seguinte:

1) uma transação deve sempre bloquear os objetos antes de acessá-los, e eventualmente liberá-los antes do seu término;

2) depois de liberar o primeiro objeto, uma transação não poderá bloquear novos objetos.

Este protocolo é assim chamado pois, pela segunda regra, cada transação passa por duas fases: em uma primeira fase objetos são apenas bloqueados e em uma segunda fase objetos são apenas liberados. O protocolo pode ser implementado tanto para um banco de dados centralizado quanto para um banco distribuído, conforme será visto no Capítulo 8.

Podemos provar que toda execução concorrente permitida pelo protocolo de bloqueio em duas fases é serializável usando o seguinte raciocínio. Seja *E* uma execução concorrente de um grupo *T* de transações. Suponha que todas as transações em *T* terminam em *E* e que seguiram o protocolo de bloqueio em duas fases em *E*. O ponto em que cada transação T_i libera o primeiro objeto em *E* é chamado de *ponto de bloqueio* de T_i . Considere a execução serial *S* em que as transações são processadas na ordem dos pontos de bloqueio em *E*. Teremos então que *S* e *E* serão equivalentes pois, dado o uso de bloqueios, é possível comutar a ordem das operações em *E* para que se transforme em *S* sem alterar a computação das transações ou o estado final do banco de dados. Logo, *E* é serializável.

A guisa de exemplo, considere novamente as transações T_1 e T_2 da seção anterior acessando um banco de dados centralizado. Considere agora que elas bloqueiam explicitamente os dados seguindo a política do protocolo de bloqueio em duas fases. Suponha que a seqüência de ações elementares sobre o banco de dados, incluindo as ações de bloqueio e liberação, gerada pela execução da transação T_1 (supondo que haja saldo suficiente) seja:

B_{11}: bloqueie o saldo *P* para T_1

T_{11}: leia o saldo *P* para *X*

X := X - 5.000

B_{12}: bloqueie o saldo *C* para T_1

T_{12}: leia o saldo *C* para *Y*

Y := Y + 5.000

T_{13}: escreva o novo saldo *Y* em *C*

L_{11}: libere o saldo *C*

T_{14}: escreva o novo saldo *X* em *P*

L_{12}: libere o saldo *P*

Suponha que a seqüência de ações elementares sobre o banco de dados gerada pela execução da transação T_2 (supondo que haja saldo suficiente) por sua vez seja:

B_{21}: bloqueie o saldo *C* para T_2

T_{21}: leia o saldo *C* para Z

Z := Z - 10.000

T_{22}: escreva o novo saldo *Z* em *C*

L_{21}: libere o saldo *C*

A execução concorrente, abstraída pela seguinte seqüência de rótulos, seria permitida pelo protocolo de bloqueios:

$$L = B_{11}\ T_{11}\ B_{21}\ T_{21}\ T_{22}\ L_{21}\ B_{12}\ T_{12}\ T_{13}\ L_{11}\ T_{14}\ L_{12}$$

Esta execução é serializável conforme discutido na seção anterior. Isto pode ser visto de outra forma, ilustrando o raciocínio da prova de correção do protocolo de bloqueio em duas fases. O bloqueio de *P* para T_1 desde B_{11} até L_{12} implica em que nenhuma ação de T_2 ou qualquer outra transação pode acessar *P* durante este período, pois a transação em questão teria que bloquear também P, o que não é permitido. Isto garante que as ações T_{21} e T_{22} não afetam o valor de *P* lido por T_{11}. Logo T_{11} pode ser comutado com estas ações sem que o processamento de *T1* em *E* se altere. Neste caso particular, esta observação basta para mostrar que, assim sendo, *E* é equivalente à seguinte execução serial (obtida comutando-se &T11. com &T21. e &T22. em *E*):

$$S = B_{21}\ T_{21}\ T_{22}\ L_{21}\ B_{11}\ T_{11}\ B_{12}\ T_{12}\ T_{13}\ L_{11}\ T_{14}\ L_{12}$$

Considere agora uma outra execução concorrente, abstraída pela seguinte seqüência de operações:

$$N = B_{11}\ T_{11}\ B_{12}\ T_{12}\ (\ B_{21}\ \ldots$$

O protocolo de bloqueios não permitiria que B_{21} fosse executada no ponto indicado pois o saldo *C* está bloqueado para a transação T_1. A transação T_2 espera então pela liberação de *C*, sendo que a execução poderia prosseguir da seguinte forma, por exemplo:

$$N = B_{11}\ T_{11}\ B_{12}\ T_{12}\ T_{13}\ L_{11}\ B_{21}\ T_{21}\ T_{22}\ L_{21}\ T_{14}\ L_{12}$$

Note que esta segunda execução também é serializável.

No exemplo acima, o bloqueio aos dados foi incorporado as transações de tal forma a seguir a técnica de duas fases. No entanto, o protocolo de bloqueio em duas fases pode ser implementado de forma transparente aos usuários, conforme discutido no Capítulo 8. A mais imediata destas implementações em um ambiente distribuído segue a seguinte idéia:

1) quando uma consulta é enviada para ler dados de um banco local, os dados são bloqueados automaticamente antes de executá-la;

2) quando uma mensagem PREPARE_SE é recebida na primeira fase do protocolo bifásico, os dados que serão modificados são bloqueados, aqueles que foram apenas lidos podendo ser liberados, antes do nó responder a mensagem;

3) após as atualizações terem sido instaladas no banco, caso o nó receba uma mensagem CONFIRME, ou após o nó ter recebido uma mensagem CANCELE, os dados atualizados localmente são liberados.

Na discussão sobre o protocolo de bloqueio em duas fases assumiu-se que todas as transações terminam. O uso de bloqueios por si só não garante esta propriedade. De fato, um impasse é criado sempre que uma seqüência de transações $T_1\ ,\ldots,\ T_n\ ,\ T_1$, onde T_i espera por T_{i+1} , é formada. Esta situação será chamada de *bloqueio mútuo*. Mecanismos adicionais deverão existir para detetar e resolver bloqueios mútuos. O método de resolução usualmente empregado consiste em reiniciar a transação na seqüência de impasse que consumiu menos recursos até o momento. Em um banco de dados distribuído, deteção de bloqueios mútuos é

especialmente difícil pois poderá ser necessário levantar quais transações esperam por quais transações ao longo de vários nós. Ou seja, a seqüência de impasse poderá se estender ao longo de vários nós da rede.

Quanto a gerência dos bloqueios em si, apenas mencionaremos aqui que, na realidade, so há necessidade de bloquear o acesso concomitante a um objeto quando se está modificando seu valor. Posto de outra forma, não há inconveniente em que várias transações leiam o valor de um objeto sem bloqueá-lo. Isto porque a operação de leitura não muda o estado do banco de dados. Os problemas aparecem quando uma transação atualiza o valor de um objeto e outra lê/atualiza o mesmo objeto, quando então o estado do banco de dados é modificado. Esta alteração, se tornada visível intempestivamente, levaria a inconsistências. Uma maneira de concretizar a idéia seria postular-se diferentes *modos* de bloqueios: *modo partilhado* e *modo exclusivo*.

Várias transações distintas poderiam bloquear um objeto em modo partilhado, desde que nenhuma destas transações fosse atualizar o valor do respectivo objeto. Apenas uma transação poderia bloquear um objeto em modo exclusivo. Neste caso, a transação pretenderia modificar o valor do objeto. O protocolo de bloqueio/liberação de objetos teria, claro, que ser modificado para refletir esta nova mecânica.

Métodos de Pré-Ordenação

O protocolo de bloqueio discutido na seção anterior garante que toda execução E de um conjunto T de transações é serializável. Mais ainda, sabemos que a ordem de serialização é dada pela ordem em que os pontos de bloqueio foram atingidos em E. Ou seja, os usuários têm a ilusão de que as transações executaram seqüencialmente na ordem dos pontos de bloqueio. Porém, esta ordem é determinada "a posteriori", durante a execução das transações e reflete um entrelaçamento arbitrário das ações elementares destas transações. Não é determinada por nenhum fator sobre o qual o usuário tenha controle direto. Uma transação T_i poderá ser iniciada antes de outra T_j mas em E tudo se passa como se T_j tivesse sido executada antes de T_i. Basta para isto que o ponto de bloqueio de T_j tenha sido atingido antes do ponto de bloqueio de T_i em E.

O método de pré-ordenação, como o nome indica, tem um comportamento diametralmente oposto. Uma ordem inicial é dada às transações, digamos de acordo com o instante em que foram iniciadas. As transações são processadas concorrentemente, mas o controle de concorrência garante que a execução final é equivalente à execução seqüencial das transações na ordem imposta "a priori".

Mais precisamente, o *protocolo de pré-ordenação* para um banco de dados distribuído dita o seguinte:

1) As transações, quando são submetidas, recebem uma *senha* ou *número de protocolo*, único em toda a rede, dado pelo gerente de transações do nó onde foram submetidas;

2) As ações de uma transação herdam a sua senha;

3) Localmente, em cada nó onde uma parte do banco está armazenada, duas ações de transações diferentes são processadas em ordem de senhas, exceto se a sua ordem relativa puder ser invertida sem afetar a computação (isto é, se as ações *não conflitam*).

4) Se uma ação violar a condição do item anterior, é rejeitada e a transação correspondente reiniciada com outra senha.

A correção deste protocolo é imediata. Seja *E* uma execução concorrente de um grupo *T* de transações. Suponha que todas as transações terminem em *T* e que o protocolo de pré-ordenação tenha sido seguido. Considere a execução serial *S* em que as transações são processadas na ordem de senha. Teremos, então, que *S* e *E* serão equivalentes pois, como as ações locais que não podem ser comutadas foram executadas em ordem de senha, é possível comutar a ordem das outras operações em *E* para que se transforme em *S* sem alterar a computação das transações ou o estado final do banco de dados. Logo, *E* é serializável.

Considere, novamente, as transações T_1 e T_2 das seções anteriores supostas acessando contas em um banco de dados centralizado, por simplicidade. A seqüência de ações de T_1 então considerada é repetida aqui para facilitar referências:

T_{11}: leia o saldo *P* para *X*
X := X - 5.000

T_{12}: leia o saldo *C* para *Y*
Y := Y + 5.000

T_{13}: escreva o novo saldo *X* em *P*

T_{14}: escreva o novo saldo *Y* em *C*

A seqüência de ações de T_2, por sua vez, será:

T_{21}: leia o saldo *C* para *Z*
Z := Z - 10.000

T_{22}: escreva o novo saldo *Z* em *C*

Suponha agora que T_1 e T_2 receberam senhas s_1 e s_2 tais que $s_1<s_2$. Estas senhas são repassadas para as ações de T_1 e T_2. Considere uma execução concorrente de T_1 e T_2 representada pela seguinte seqüência de ações:

$$L = T_{11}\ T_{12}\ T_{13}\ T_{21}\ T_{22}\ T_{14}$$

Esta seqüência seria aceita como está pelo protocolo de pré-ordenação. De fato, acompanhando a dinâmica do processamento, teríamos:

T_{11} é submetida para o protocolo e é aceita, já que nenhuma outra ação foi executada.

T_{12} e T_{13} são submetidas e aceitas pois são da mesma transação que T_{11}

T_{21} é submetida. Como T_{21} lê o saldo *C* e T_{13} escreve neste saldo, a ordem relativa de T_{13} e T_{21} não pode ser invertida sem alterar a computação. Logo estas operações terão que ser processadas em ordem de senha. Mas a senha associada à T_{21} é maior do que a senha associada à T_{13} (e T_{13} já foi processada). Logo T_{21} é aceita.

T_{22} é aceita por motivos semelhantes.

T_{14} é submetida. A senha associada à T_{14} é menor do que a senha das duas últimas ações processadas, T_{21} e T_{22}. Porém, como T_{14} afeta o saldo *P* e T_{21} e T_{22} acessam o saldo *C*, a ordem relativa de T_{14} e T_{21} / T_{22} pode ser invertida sem que a computação seja alterada. Portanto, T_{14} é aceita e processada.

Considere agora uma seqüência diferente:

$$N = T_{11}\ T_{12}\ T_{21}\ (\ T_{13}\ \ldots$$

A ação T_{13} é necessariamente rejeitada pelo protocolo de pré-ordenação, provocando o reinício de T_1. De fato, observe inicialmente que a ordem de T_{13} e T_{21} é importante pois T_{21} lê o saldo C e T_{13} cria um novo valor para C. Logo a ordem relativa de T_{13} e T_{21} não pode ser invertida sem afetar a computação. Assim, estas ações têm que ser processadas em ordem de senha, ou seja, T_{21} depois de T_{13}, neste caso. Como na seqüência acima T_{21} já foi processada quando T_{13} é recebida, esta última ação é rejeitada, forçando o reinício de T_1 com um número de senha mais alto.

Usando a notação $\neg O$ para indicar a ação que desfaz o efeito da operação O, a seqüência acima continuaria da seguinte forma, por exemplo:

$$N = T_{11}\ T_{12}\ T_{21}\ \neg T_{11}\ \neg T_{12}\ T_{22}\ T_{11}\ T_{12}\ T_{13}\ T_{14}$$

Diversas implementações do protocolo de pré-ordenação, todas transparentes aos usuários, serão discutidas no Capítulo 8. Faremos aqui apenas alguns comentários sobre o problema de gerar senhas de forma unívoca e sobre o relacionamento do protocolo de pré-ordenação com o protocolo bifásico.

Em bancos de dados centralizados senhas podem ser geradas de forma unívoca utilizando-se o próprio relógio do processador principal. Por outro lado, em um sistema distribuído, se os gerentes de transações gerarem senhas desta forma, duas transações poderão potencialmente receber a mesma senha. Isto violaria o princípio de que cada senha deve estar associada a uma transação distinta. O dilema é solucionado justapondo-se à hora local a identidade do nó onde a transação se origina, suposta única para cada nó da rede. Pode-se conseguir, assim, que transaçãoes distintas tenham senhas únicas em relação ao sistema como um todo.

Quando o SGBDD faz uso do protocolo bifásico para confirmar intenções problemas aparecem com o protocolo de pré-ordenação. Note que, na fase dois daquele protocolo, os nós que participaram da execução de uma transação não poderão mais cancelar, unilateralmente, os efeitos locais desta transação e deverão garantir que, quando a ordem do coordenador eventualmente chegar para que a transação seja confirmada, estejam preparados para tornar seus efeitos permanentes no banco de dados local. Este problema pode ser resolvido alterando-se o protocolo de pré-ordenação para que processe as ações de PREPARE_SE como se fossem as operações que atualizam os dados. Uma vez que o nó local emitiu uma mensagem de PREPARADO em resposta a uma mensagem PREPARE_SE, com senha S e indicando que um objeto X será atualizado, nenhuma outra operação com senha maior do que s e que acesse X poderá ser processada. Este problema seria solucionado, por exemplo, bloqueando-se X desde o processamento de PREPARE_SE até que a atualização seja efetivada.

Finalmente, é necessário examinar o problema de terminação. No contexto desta implementação, uma transação T_i poderá não terminar se for *reiniciada ciclicamente* já que, a cada vez que uma de suas ações chegou "atrasada", por assim dizer, com relação à outra ação com que conflita, a transação T_i é reiniciada. Uma solução para este problema consiste em reiniciar a transação T_i com uma nova senha bem maior do que a anterior para minimizar as chances de T_i ser novamente reiniciada.

A política de aumento de senha ainda tem um outro fator determinante. Existe a possibilidade do processamento de várias transações sincronizar-se de tal forma a causar o *reinício mútuo* de todas elas. Ou seja, cria-se uma seqüência de transações T_{i0} , T_{i1} ,..., $T_{I, m-1}$, T_{i0} tal que T_{ij} força o reinício de $T_{i\ ,\ j+1}$ (soma módulo m). Se o aumento da senha for o mesmo para todas estas transações, corre-se o risco de repetir-se a situação indefinidamente. Para se resolver

este problema (probabilisticamente) basta randomizar o incremento dado as senhas. Este esquema simples naturalmente não evita completamente o problema de reinício cíclico, mas o torna pouco provável.

NOTAS BIBLIOGRÁFICAS

Lindsay [1980] contém uma descrição mais ou menos detalhada das fases de início, migração e término de transações, especialmente desta última. A parte de término de transações é bem formulada. Também inclui uma boa descrição do mecanismo controle de integridade envolvendo atas e balizadores. Gray [1978] descreve algumas das estratégias usadas na implementação do Sistema R, desenvolvido no laboratório de pesquisas da IBM, em San Jose. Descreve com algum detalhe o mecanismo de atas, bem como o protocolo bifásico. Gray et all. [1979] apresenta o mecanismo de recuperação do Sistema R. Lindsay et all. [1979] proporciona uma descrição detalhada do protocolo bifásico. Lorie [1977] apresenta um método de imagens transientes. Severance e Lohman .1976] descrevem uma técnica de arquivos diferenciais. Randell et all. [1978], Verhofstad [1978], Randell [1979] e Kohler [1981] contêm tutoriais sobre controle de integridade. Bernstein e Goodman [1981] por sua vez contém um ótimo tutorial sobre controle de concorrência. Mais referências sobre os tópicos aqui cobertos encontram-se nas notas bibliográficas dos três próximos capítulos.

Capítulo 7. Executando Transações

Este capítulo será dedicado a estudar os mecanismos que devem ser acionados quando do início de uma transação, a expor técnicas para controlar a migração de transações e, finalmente, apresentar algoritmos ou protocolos que, ao término da transação, garantam sua atomicidade, esta última sendo de crucial importância para o correto funcionamento do BD. A Seção 6.2 contém uma descrição preliminar das idéias que serão detalhadas nas seções seguintes. A organização deste capítulo compreende uma seção introdutória seguida de três outras seções. Na primeira seção, o ambiente distribuído onde a transação opera é revisto. Nas três seções seguintes os mecanismos que entram em cena quando do inicio, migração e término de transações são examinados em detalhe.

7.1 UM NÓ DA REDE

Nesta subseção, partes da arquitetura do sistema serão recolocadas, agora com ênfase especial na interligação entre os módulos que mais de perto controlam a execução de transações. Desta forma, o leitor poderá ter em mente o cenário geral onde tais execuções são processadas. Detalhes serão discutidos nas subseções seguintes.

Sob o ponto de vista lógico, cada nó do sistema pode ser entendido como mostra a Figura 7.1.

Dentre todos aqueles componentes que formam o BD, estão mostrados apenas os elementos que serão mencionados neste capítulo. Seis grandes módulos são visíveis na referida figura:

- ambiente de programação local
- gerente de transações
- gerente de dados
- SGBD local
- sistema operacional local
- banco de dados local

De acordo com as indicações da figura, o seguinte desenrolar de operações seria típico. Inicialmente, o usuário obtém acesso ao sistema através de um dos nós da rede. Isto significa que um processo local é instalado para servir ao indivíduo neste nó. Usando de todos os recursos que o sistema local lhe oferece, tais como editores de texto e depuradores de programas, o elemento constrói um programa aplicativo numa linguagem de programação de alto nível. Em seguida, este programa é submetido à compilação ou preparado para futura interpretação. Estamos supondo que o programa aplicativo contém comandos da linguagem de manipulação de dados imersos entre comandos da linguagem de programação e que o programador é responsável por indicar começo e fim de transações. Uma maneira de se obter este último efeito seria limitar os comandos que descrevem uma transação entre linhas de código tais como "COMECO_DE_TRANSAÇÃO" e "FIM_DE_TRANSAÇÃO". No ato de compilar o programa fonte montado pelo usuário, ou prepará-lo para interpretação, cada seqüência de comandos da LMD que compõe uma transação é passada ao gerente de transações. Este aciona o processador de comandos da LMD sob seu controle, o qual produz o plano de execução daquela transação. O processo de análise é bastante complexo e já foi objeto de estudos nos Capítulos 4 e 5. O plano construído é, então, devolvido ao processo controlado pelo usuário que se encarrega de agregá-lo ao código objeto que está gerando, ou

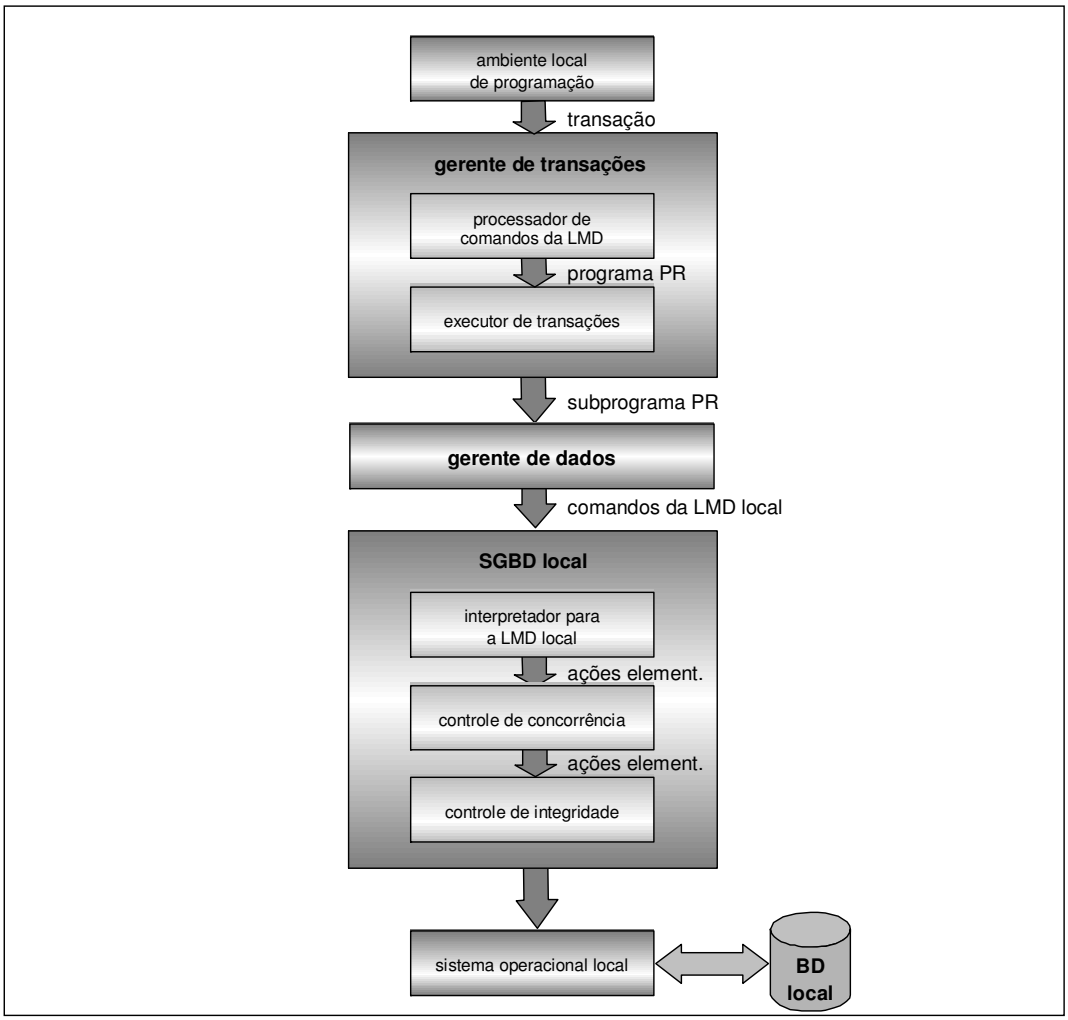

Figura 7.1 - Um Nó da Rede

à forma intermediária que está criando para posterior interpretação. Outra maneira de proceder seria instruir o processo do usuário para, cada vez que identificar comandos da LMD em meios aos comandos da linguagem de programação, incorporar ao código que está gerando indicações para invocação do gerente de transações. Assim, o código final conteria apenas chamadas para o gerente de transações e não o plano completo para execução de cada transação. Obviamente, se o código é gerado com intenções de ser reutilizado com freqüência no futuro esta última hipótese deve ser descartada. Note, inclusive, que o processo controlado pelo usuário pode estar interagindo com o SGBD de maneira conversacional. Neste cenário, alguns comandos da linguagem de alto nível seriam executados, seguidos de um comando na linguagem de manipulação de dados. Este último é repassado ao gerente de transações enquanto o processo do usuário espera pela sua conclusão. Chegando a resposta esperada, o ciclo é reiniciado até que a conversação seja dada por encerrada e o usuário cancele o processo local, abandonando o BD. Como uma outra alternativa nesta fase, o programa de que o usuário necessita poderia já ter sido compilado anteriormente, encontrando-se, agora, armazenado na biblioteca de programas local. Tudo que é necessário fazer é instruir o sistema local para que recupere o referido programa.

Neste ponto, suponha que o processo controlado pelo usuário, de uma forma ou de outra, disponha de código, objeto ou em forma intermediária, de um programa aplicativo. Como

vimos, este código pode incorporar planos completos para execução de transações ou apenas a informação necessária para a elaboração destes planos durante sua execução ou interpretação. O próximo passo seria o processo do usuário dar início à execução deste código. Assim que chega a um ponto onde se depara com material referente ao BD, o processo do usuário repassa este material ao gerente de transações local. O gerente de transações contém dois módulos importantes: um processador de comandos da LMD e um executor de transações. De acordo com os cenários mencionados acima, há dois casos a considerar. Se o material entregue já contém o plano da transação, este passa direto ao executor de transações, o qual inicia a execução do referido plano. É o caso de um programa usado com freqüência, cujo código é armazenado e posteriormente recuperado a cada invocação. A outra possibilidade se dá quando o material enviado ao gerente de transações contém apenas comandos da linguagem de manipulação de dados. Neste caso, estes são examinados, em primeira instância, pelo processador de comandos da LMD que produz um plano de execução. Só então o resultado é entregue ao executor de transações. Em qualquer dos dois casos, o resultado final que o executor de transações recebe é um programa pertencente a classe PR, conforme especificado no Capítulo 5. Um tal programa, relembrando, é uma receita que contém, para cada um dos nós que deverão participar da execução da transação, quais consultas e atualizações locais deve efetuar, bem como que resultados, ou tabelas, deve encaminhar a outros nós. Obviamente, a ordem em que estas operações serão executadas também deve estar indicada no programa. É tarefa do executor de transações, tendo em mãos o programa da classe PR que lhe é passado pelo processador de comandos da LMD, interpretar a receita e despachar tarefas aos vários nós participantes de forma orquestrada, garantindo que a receita é seguida à risca.

Este capítulo começa a focalizar os aspectos da execução de transações justamente no instante que o programa PR, suposto como uma transação completa, é entregue ao executor de transações. A primeira tarefa deste último é consultar a receita do programa PR e separar partes do programa, despachando-as aos nós remotos apropriados. Isto é feito através de trocas de mensagens entre os gerentes de transações envolvidos. Os gerentes de transações que receberam tarefas passam a executá-las e, quando terminam, informam que tal é a situação ao gerente de transações que solicitou o trabalho. É claro que, se for o caso, o gerente de transações do nó original também executará parte do programa. As mensagens trocadas entre gerentes de transações devem conter especificações do trabalho a ser realizado nos locais remotos. A fim de minimizar os custos de tais comunicações, que podem ser elevados, é interessante que a descrição do trabalho seja feita em "alto nível" e, também, que o máximo de informação seja enviado em cada mensagem. Assim procedendo, estaríamos minimizando tanto o número de mensagens trocadas quanto o comprimento de cada uma delas. Ha várias maneiras de se controlar a migração de transações nesta fase. Algumas delas serão examinadas em detalhes em seções posteriores deste capítulo.

Se o BD for heterogêneo, a solicitação de trabalho recebida pelo sistema local de cada nó virá codificada em forma de comandos padronizados da linguagem de manipulação de dados pivot. Deve, portanto, ser traduzida para a linguagem de manipulação de dados usada no particular nó para o qual foi endereçada. Sob esta hipótese, o gerente de transações local, ao receber a mensagem de trabalho, passa esta ao gerente de dados local, que se encarrega de ajustá-la às condições locais. Uma vez que o gerente de dados obtenha um programa PR, descrevendo parte da transação original, e o tenha ajustado à linguagem de manipulação de dados local, o resultado é passado ao SGBD local. Este último deve apelar para um compilador ou interpretador desta linguagem que possibilite traduzir as instruções de trabalho em ações elementares. É neste ponto que o SGBD local gera código de acesso às várias estruturas do BD local, de acordo com os métodos de acessos de que dispõe localmente.

Lembre que ações elementares manipulam objetos físicos do BD. Os problemas de endereçamento e acesso, portanto, devem estar equacionados quando chegamos a este ponto. Uma vez mapeada em ações elementares, a solicitação de trabalho passa à execução sob o controle do SGBD local. Para cada solicitação de trabalho que recebeu, o SGBD local cria um agente local, ou um processo local, que vai se desincumbir da execução das ações elementares que compõem esta solicitação de trabalho. É responsabilidade do SGBD local coordenar a operação de cada agente local com os módulos de controle de integridade e controle de concorrência, ambos sob sua tutela, de forma a possibilitar uma execução ordenada e segura a cada transação. É até este ponto que vai o interesse deste capítulo, sem entrar no mérito dos mecanismos de controle de integridade e controle de concorrência, visto que serão alvo de atenções nos capítulos seguintes.

Completando o quadro da Figura 7.1, temos os agentes locais a gerarem chamadas para o sistema operacional local. Como o SGBD local lida a nível de operações elementares, estas chamadas contêm endereços físicos de objetos bem definidos. O sistema operacional local, por sua vez, ao atender estas chamadas completa o acesso movendo objetos do BD local para a memória principal e vice-versa.

Com relação aos mecanismos de controle de concorrência, será suposto que operem a contento e sem problemas. Sob a ótica do presente capítulo, tudo que se pretende fazer é seguir a vida de uma transação desde o momento em que o executor de transações inicia o processo até sua completa execução. A interação entre os mecanismos de controle de concorrência e os algoritmos de migração e término de transações por ser mais tênue e, portanto, mais fácil de ser visualizada, será mencionada apenas quando oportuna. Mesmo porque, lembre-se, a função básica do controle de concorrência é proporcionar a ilusão de que apenas uma transação executa de cada vez.

No que se refere aos mecanismos de controle de integridade, o que se pretende é expor a interação entre estes e aqueles mecanismos que entram em cena quando do início, migração e término de transações. A perfeição imposta neste ponto aos primeiros não deve esconder sua interação com os últimos. Apenas que, no momento, não será detalhada nenhuma das técnicas de controle de integridade, porque isto é tarefa para o Capítulo 9. Entretanto, simplesmente supor recuperação total na presença de falhas não é suficiente para garantir o processamento ordenado de transações em ambientes distribuídos, como veremos adiante. É necessário que protocolos específicos sejam invocados e ações concretas sejam tomadas, especialmente quando da migração e término de transações, mesmo estando os mecanismos de controle de integridade funcionando a toda prova. O que vamos supor a respeito dos métodos de controle de integridade, por ora, é que provêem proteção total e cabal contra quaisquer tipos de falhas. Embora o sistema esteja sujeito a falhas, quando estas ocorrem estes mecanismos serão sempre capazes de recuperar toda a informação que lhes foi confiada como sendo vital, não importando o tipo de falha que se abata sobre o SGBD ou a freqüência com que estes eventos ocorram.

7.2 INÍCIO DE TRANSAÇÕES

Voltando à Figura 7.1, estamos focalizando agora o gerente de transações quando recebe, do processo controlado pelo usuário, um conjunto de comandos da LMD descrevendo uma transação. O nó onde isto se passa, isto é, o nó onde reside o processo do usuário, será chamado de *nó de origem* da transação. Os comandos recebidos são, então, passados ao processador de comandos da LMD, um módulo controlado pelo gerente de transações do nó de origem. A seguir, o processador de comandos da LMD cria um programa da classe PR

descrevendo o plano global de execução da transação. Este plano prevê, eventualmente, a cooperação de vários outros nós remotos na execução da transação. Cada um destes nós será chamado de *nó participante*. É claro que o próprio nó de origem também participará da execução da transação, caso isto esteja previsto no plano global de execução desta transação. A responsabilidade de distribuir este plano para os nós participantes compete ao executor de transações, outro módulo controlado pelo gerente de transações do nó de origem. Retraduzir instruções recebidas do nó de origem, refletindo o particular dialeto da LMD usado em um nó participante, é tarefa delegada ao gerente de dados deste nó participante. A partir desta retradução, o SGBD do nó participante compila estas instruções para ações elementares coerentes com o modelo físico presente neste nó. Este mesmo SGBD controla a execução destas ações elementares. Para tal, o SGBD do nó participante cria e instala um agente local, munido do respectivo descritor de transação, cuja tarefa será invocar e executar as referidas ações elementares.

O *agente local* se constitui, simplesmente, em mais um processo que é criado junto ao sistema local. É instalado pelo SGBD local e este tem poderes para ativá-lo ou extinguí-lo, conforme suas conveniências. O *descritor de transação* nada mais é que um registro residente na memória do correspondente agente local e composto de campos que serão utilizados no decorrer da execução da transação. Tanto o SBGD como o executor de transações têm acesso a cada agente local e seu respectivo descritor que porventura estejam ativos nos locais onde residem. Cada vez que um nó participante recebe mensagem solicitando trabalho local em favor de determinada transação, um agente local e um descritor são instalados neste nó participante para atenderem à referida solicitação. Desta maneira, pode-se assumir que cada transação ativa em um dado nó terá um ou vários agentes locais operando localmente em seu favor, quer seja este seu nó de origem ou não. Outro ponto importante é que o agente local, como todo outro processo em execução localmente, pode residir, em determinados intervalos de tempo, na memória principal do sistema. Em caso de falhas primárias, seu conteúdo será, portanto, suposto irrecuperável. Há de haver, é claro, mecanismos que resguardem informação suficiente para que o SGBD de cada nó, ao voltar à vida, tenha condições de tratar as transações a que serviu de maneira adequada, evitando inconsistências. Na seção seguinte, serão discutidos mecanismos para coordenar estes algoritmos locais de modo a preservar a ordem em todo o sistema.

São estes os campos que compõem o descritor de uma transação: identificação e estado de retorno.

O campo de *identificação* é preenchido com informação que identifica a transação, univocamente, perante todo o sistema. Se este for o nó de origem da transação, a identificação deve ser fabricada localmente. Se o presente nó é apenas mais um nó participante para o qual a transação migrou, então a mensagem que requisitou trabalho a este nó deverá, de alguma forma, transmitir a identificação que já fora associada a esta transação em seu nó de origem. Assim, ao migrar de um local para outro a transação mantém sua identidade original. Caso uma identificação única apenas no âmbito local de cada nó fosse suficiente, um contador local ou mesmo o valor do relógio do processador central, desde que devidamente protegidos contra falhas, serviriam como fonte de identificadores unívocos. Para produzir uma identificação unívoca globalmente, pode-se justapor a este valor local a identidade que o nó de origem da transação detém como membro da rede de comunicação de dados onde está imerso o BD distribuído. Esta identidade é suposta única, o que é razoável pois servirá de endereço para troca de mensagens entre os nós. Este esquema tem a vantagem adicional de levar, junto com a identidade da transação, a identificação de seu nó de origem. De fato, este benefício será útil mais tarde e, portanto, será tido que tal é o caso.

A necessidade de uma identificação global pode ser compreendida em vista do fato de que a transação pode migrar para outros nós. Como entre dois particulares nós podem circular mensagens em favor de várias transações requisitando trabalho ou reportando resultados, é necessário que cada mensagem identifique a particular transação que a gerou, caso contrário não haveria meios de se associar resultados a transações. Mais ainda, o controle de integridade em cada nó participante deve ter a capacidade de, ao retornar após uma falha, identificar que ações elementares foram executadas em benefício de quais transações. Só assim poderá desfazer/refazer os efeitos locais desta transação, caso isto seja necessário. Pense, por exemplo, no mecanismo de atas onde cada ação elementar é lá registrada para fins de controle de integridade. Como transações distintas originadas em nós diferentes podem migrar para um nó comum, a identificação global evita confusões quando um SGBD local tenta agrupar, lendo a partir da ata, as ações elementares que pertencem a uma mesma transação. Há outras situações, de normalidade mesmo, quando necessitamos cancelar transações e, em conseqüência, recorrer à ata para inverter os efeitos de ações elementares. Uma delas se configura durante a execução do protocolo bifásico quando, mesmo na ausência de falhas locais em determinado nó participante, o SGBD residente no local de origem decide pelo cancelamento global da transação devido a falhas em outros nós. Bloqueios globais são outro exemplo de eventos que provocam cancelamentos não forçados por falhas no sistema. Nestes casos, novamente, consultas à ata seriam necessárias para obter-se as ações elementares da transação que foi eleita para cancelamento e, em seguida, desfazer-se seus efeitos.

O campo de *estado de retorno* é usado pelo protocolo bifásico para confirmar intenções que executará neste nó quando do término da transação. A finalidade precisa deste campo de estado de retorno ficará clara quando abordarmos a descrição daquele protocolo. Sumariamente, este campo conterá os diferentes "estados" por que passará o protocolo à medida que for executado. Por ora, basta destacar que deve ser iniciado como DESCONHECIDO.

O início de execução da transação pode ser tomado como aquele instante quando o executor de transações do nó de origem recebe, do processador de comandos da LMD, o programa da classe PR que descreve a transação. Porém, antes de iniciar a distribuição do plano de execução aos nós participantes, o executor de transações deve tomar uma providência importante: instruir o SGBD local para que, através dos mecanismos de controle de integridade, armazene a lista de nós participantes e a identificação da transação em local seguro. Esta atitude será vital mais tarde quando forem analisados, na presença de falhas, o protocolo bifásico e a migração de transações. A migração da transação pode, após garantida a segurança destes dois itens, ser iniciada pelo executor de transações do nó de origem o qual principia a despachar mensagens aos nós participantes. Quanto à lista de nós participantes, o executor de transações não deve ter dificuldades em obtê-la visto que, nesta fase, já dispõe do plano global de execução desta transação. Basta inspecionar este plano para obter a referida lista.

Resumindo, o seguinte se passa no local de origem quando da iniciação de uma transação, a começar pelo instante em que o executor de transações do nó de origem recebe o plano de execução desta transação:

1) executor de transações cria uma identificação global para a transação;

2) executor de transações compila a lista de todos os nós participantes com respeito a esta transação;

3) executor de transações instrui o SGBD local para acionar os mecanismos de controle de integridade para que registrem em lugar seguro:
   a) a identificação desta transação;
   b) a lista de nós participantes que recebeu;
4) SGBD local retorna ao executor de transações uma mensagem informando do sucesso da operação;
5) executor de transações da origem está pronto para iniciar a migração da transação.

A transação está ativa neste nó a partir do instante em que o passo 3 se completar.

Já em um nó participante, o processo de iniciação é desencadeado quando cada mensagem solicitando trabalho em benefício desta transação é recebida. Estamos incluindo nesta categoria também o nó de origem da transação, quando a este é delegado à execução de algum trabalho local em favor da transação. A única diferença é que, neste caso, obviamente, mensagens não seriam trocadas pois estamos operando em um mesmo nó.

Os passos são os seguintes:

1) executor de transações recebe uma mensagem, vinda de outro executor de transações, solicitando trabalho;
2) se esta é a primeira solicitação de trabalho em favor desta transação, o executor de transações instrui o SGBD local para que registre em lugar seguro que esta transação está agora ativa neste nó;
3) executor de transações instrui o SGBD local para criar um agente local para esta solicitação contendo, em memória própria, um descritor cujos campos são iniciados assim:
   a) campo de identificação recebe a identificação da transação que veio junto com a mensagem que requisitou o trabalho;
   b) campo de estado de retorno recebe o valor DESCONHECIDO;
4) este nó esta pronto para iniciar a execução local desta solicitação de trabalho.

A transação só estará ativa neste nó quando o passo 2 se completar por ocasião da chegada da primeira mensagem solicitando trabalho em favor desta transação.

Note que, sendo ou não esta a primeira solicitação de trabalho em benefício desta transação, não são tomadas quaisquer providências por parte do nó remoto no sentido de registrar, em lugar seguro, que esta solicitação de trabalho está sendo atendida. Entretanto, após a execução da primeira ação elementar que venha a modificar o valor de qualquer objeto do BD, este fato será automaticamente registrado. Lembre-se de que precisamos registrar os efeitos de cada ação elementar executada, junto com a identidade da respectiva transação, para a contingência do nó local ser atingido por uma falha e precisarmos desfazer efeitos referentes a ações elementares invocadas no passado. Os mecanismos de controle de integridade se encarregarão de, caso haja o registro, tomar as medidas cabíveis. O sistema local pode, entretanto, falhar naquele intervalo entre o instante de recebimento da mensagem solicitando trabalho e o momento em que os efeitos da primeira ação elementar são registrados a salvo, em local seguro. Neste caso, o sistema local desconheceria a existência da mensagem solicitando trabalho quando estivesse se recuperando de uma eventual falha. Em

conseqüência, não tomará atitude nenhuma, podendo deixar o nó solicitante a esperar indefinidamente por uma resposta que não virá. Suporemos que, quando da chegada de qualquer mensagem, esta é posta a salvo em área própria e que sobreviva a falhas :hp1.antes:ehp1. de enviarmos ao nó remetente a confirmação (do inglês, "ack") de que a referida mensagem foi recebida. Assim, os próprios protocolos de comunicação se encarregarão de reenviar a mensagem caso o sistema falhe antes de salvá-la. Tendo-a salvo, claro, estará disponível para inspeção após o sistema retornar à operação normal.

7.3 MIGRAÇÃO DE TRANSAÇÕES

Após terem sido iniciadas no local de origem, transações podem migrar para outros nós da rede. A migração tem lugar quando um certo nó requisita que outro nó remoto efetue trabalho, ou computações, em benefício de determinada transação. O nó remoto, por sua vez, envia ao solicitante um aviso quando termina de executar a tarefa que lhe foi delegada. No modelo que assumiremos não será necessário que o nó remoto envie "resultados" ao solicitante. Toda transferência de arquivos ou tabelas será explicitamente prevista no plano global da transação cuja execução está sendo controlada pelo nó de origem. Equivalentemente, pode-se encarar o sinal de "fim de execução" como sendo o "resultado" que o nó remoto envia ao nó solicitante. A cada mensagem enviada, porém, o nó que a remete espera uma resposta explícita por parte do nó recebedor. Assim, meramente obter confirmações (do inglês, "acks") vindas do nó recipiente não será suficiente para o nó remetente. Será preciso que o primeiro construa e despache uma mensagem específica para cada mensagem recebida do segundo. Para esta troca de informações o SGBD faz uso de uma rede de comunicação de dados que está a sua disposição. No transcorrer desta seção, em primeiro lugar, será examinado o ambiente em que o SGBD opera, em relação a esta rede de comunicação. Em seguida, os tipos de mensagens trocadas serão descritos. Depois, o ambiente em que se processam as migrações, com suas limitações e liberdades, será enfocado. Finalmente, a própria migração de transações será abordada tanto na ausência como na presença de falhas que podem interromper o processamento normal de qualquer dos nós envolvidos.

7.3.1 A Rede de Comunicação de Dados

Será suposto que os SGBD locais operem a rede de comunicação a nível de programa aplicativo. Isto pressupõe que todos os detalhes dos protocolos que implementam a comunicação serão tidos como totalmente transparentes aos SGBDs. Em particular, problemas tais como:

- mensagens perdidas quando em curso
- mensagens entregues adulteradas
- mensagens entregues em endereços errados
- mensagens duplicadas
- mensagens entregues fora de ordem

serão supostos como devidamente resolvidos em níveis inferiores àquele da camada aplicativa onde o SGBD atua. Qualquer bom texto tratando a respeito de redes de computadores e dos respectivos protocolos de comunicação de dados dará uma visão satisfatória dos problemas encontrados e das soluções propostas. As notas bibliográficas no fim deste capítulo contêm sugestões neste sentido. Em qualquer caso, o SGBD toma como certo que toda mensagem será entregue sem adulterações, ao destinatário correto e dentro de

um intervalo de tempo finito. Além do cenário de normalidade, há dois outros que podem ser encontrados:

1) Reposta alguma jamais se materializa junto ao nó que gerou a requisição. Então, forçosamente, o nó recipiente falhou no hiato compreendido entre o instante quando recebeu a mensagem e antes que pudesse gerar uma resposta explícita;

2) Resposta do nó solicitado tarda em vir. Então pode-se concluir que:

   - a rede de comunicações está com elevados índices de tráfego, causando atrasos, ou
   - nó recipiente está com carga além do normal e, portanto, lento ao responder. Ou ambos.

Na prática, e difícil distinguir os dois casos. Visualizando a situação sob o ponto de vista do nó que enviou a mensagem, este não tem condições de separar as duas hipóteses, uma vez que dispõe apenas da própria rede como único veículo para sondar o meio exterior. Não há como distinguir entre "tardando em chegar" e "não virá jamais". A única maneira de amenizar o problema envolveria, de uma forma ou outra, a operação de relógios ou contadores locais que medissem a passagem do tempo desde que a mensagem foi entregue à rede. Iniciados quando a mensagem é enviada, dispariam alarmes (do inglês, teria ocorrido um "timeout") assim que o intervalo sem resposta se tornasse longo demais. Isto dispararia atitudes específicas por parte do SGBD que enviou a mensagem tais como o envio de outra mensagem, mais curta, indagando a respeito das condições no local de destino, ou mesmo o simples reenvio da mensagem original. Estes mecanismos, especialmente quando implementados de maneira "ad hoc", devem ser abordados com cautela, pois podem produzir efeitos colaterais, muitas vezes não previstos, contribuindo para tornar o quadro geral ainda mais complexo e confuso. Neste nível introdutório, estas técnicas não serão utilizadas ou descritas em maiores detalhes. Exemplos concretos de seu uso podem ser encontrados nas referências que tratam de protocolos de comunicação em redes de dados citadas ao fim deste capítulo. No que segue, este problema será ignorado e um "tempo finito" tido como satisfatório, uma hipótese que pode ser mais ou menos razoável dependendo a que se preste o BD em questão.

7.3.2 As Mensagens

Há dois tipos de mensagens envolvidas, quais sejam, mensagens de trabalho e mensagens de resposta. Como o próprio nome indica, o gerente de transações local, detectando a necessidade de trabalho ou informação que devam ser requisitados a um certo nó remoto, constrói uma *mensagem de trabalho* e a despacha para este nó remoto. Uma tal mensagem de trabalho envolve os seguintes campos: identificação e descrição.

A identidade de toda transação ativa em cada nó é suposta conhecida do gerente de transações local uma vez que este é responsável por coordenar a execução de cada uma delas. Desta forma, não deve haver dificuldade em copiar esta identidade para o campo de *identificação* da mensagem de trabalho. Este campo contém, assim, a identidade global da transação. O campo de *descrição* informa exatamente o que deve ser executado remotamente em favor desta transação, incluindo aqui também as mensagens PREPARE-SE, CONFIRME e CANCELE. Estas mensagens dizem respeito apenas ao protocolo bifásico para confirmar intenções e serão ignoradas no momento. Serão incorporadas ao cenário geral na última seção deste capítulo. A particular maneira como a descrição de trabalho é codificada neste campo é imaterial para a discussão que segue. Note, porém, que descrever o trabalho em uma "linguagem de alto nível" teria a vantagem de encurtar o comprimento das mensagens

trocadas. Os SGBD remotos seriam responsáveis por compilar a descrição do trabalho a ser executado para ações elementares executáveis localmente.

Ao receber uma mensagem de trabalho, o nó recipiente observa o protocolo de iniciação descrito na seção anterior. Isto feito, o agente local trata de se desincumbir das tarefas que lhe foram confiadas. Uma vez completo o trabalho solicitado, o executor de transações acionado constrói uma *mensagem de resposta* contendo os campos: identificação; remetente; veredicto.

O campo de *identificação* contém a identificação da particular transação a que se refere a mensagem. No espaço de *remetente* é colocada a identidade do nó que responde além de outras informações específicas da tarefa executada, como veremos adiante. Finalmente, no campo de *veredicto* será colocada uma indicação a respeito do que ocorreu durante a execução do trabalho requisitado. Há cinco possibilidades: CANCELADO, EFETUADO, PREPARADO, IMPOSSIBILITADO ou COMPLETADO. As três últimas alternativas são de uso exclusivo do protocolo bifásico para confirmar intenções e também serão postas de lado nesta seção.

Para todos os efeitos lógicos, tudo se passa como se construir e enviar mensagens, quer de trabalho ou de resposta, fosse uma "ação elementar" invocada em benefício da transação em questão. Como veremos, estes atos receberão tratamento similar as ações elementares próprias no que diz respeito a proteção contra falhas.

7.3.3 A Árvore de Execução

O plano de execução da transação é descrito por um programa *P*, da classe PR, recebido pelo executor de transações do nó de origem. A todo tal programa corresponde uma *árvore de execução*, *A(P)*. As folhas de *A(P)* serão rotuladas com as operações básicas de *P*, quais sejam, consultas, atualizações e transferências de tabelas. Os nós intermediários de *A(P)* receberão o rótulo de ";" ou de "//". A estrutura de *A(P)* pode ser lida diretamente a partir da descrição de *P*. Mais precisamente, a construção de *A(P)* pode ser descrita pelo seguinte procedimento recursivo:

1. se *P* for da forma $P = O$, onde *O* e uma das operações básicas, então *A(P)* será formada por de um único nó rotulado com *O* ;

2. se *P* for da forma $P = R_1 ; \ldots ; R_n$, então o procedimento é invocado recursivamente para se obter $A(R_i)$, para todo i, $1 \leq i \leq n$. Isto findo, *A(P)* é formada por uma raiz, rotulada com ";", tendo como descendentes diretas as *n* árvores *A(Ri),* na ordem *1 ... n*.

3. se *P* for da forma $P = R_1 \ // \ldots // \ R_n$, o procedimento é invocado recursivamente para se obter $A(R_i)$, para todo para todo i, $1 \leq i \leq n$. Tendo obtido $A(R_i)$, então *A(P)* é formada por uma raiz, rotulada com "//", a qual estão ligadas como descendentes diretas as *n* árvores $A(R_i)$.

Como um exemplo simples, seja

$$P = ((\ Q_i\ ;\ (\ T_{i,j} \ // \ T_{i,k}\ ))\ //\ (\ Q_j\ ;\ T_{j,k}\ ))\ ;\ Q_k$$

onde Q_i indica uma consulta a ser processada pelo nó i e $T_{n,m}$ indica uma transferência de tabela do nó *n* para o nó *m.* Então, *P* pode ser escrito na forma $P = P_1;P_2$ onde

$$P_1 = ((\ Q_i\ ;\ (\ T_{i,j} \ // \ T_{i,k}\ ))\ //\ (\ Q_j\ ;\ T_{j,k}\ ))\ \text{ e }\ P_2 = Q_k$$

Então, de acordo com o passo 2, devemos invocar o procedimento recursivamente para obter $A(P_1)$ e $A(P_2)$. Como $P_2 = Q_k$ é uma operação básica, segundo o passo 1, $A(P_2)$ é obtida simplesmente rotulando-se um nó com Q_k. Após a recursão ter completado $A(P_1)$, teríamos a árvore como mostra a Figura 7.2 à esquerda.

Resta construir $A(P_1)$. Mas $A(P_1)$ é da forma $P_1 = P_{11} \,//\, P_{12}$ onde

$$P_{11} = Q_i \,;\, (\, T_{i,j} \,//\, T_{i,k}\,) \quad \text{e} \quad P_{12} = Q_j \,;\, T_{j,k}$$

Devemos, segundo o passo 3, apelar de novo à recursão a fim de obtermos $A(P_{11})$ e $A(P_{12})$. Uma vez isto feito, teríamos $A(P)$ na forma mostrada à direita na Figura 7.2. Embarcando na recursão para construir $A(P_{11})$, obtemos $P_{11} = P_{111} \,//\, P_{112}$ onde

$$P_{111} = Q_i \quad \text{e} \quad P_{112} = T_{i,j} \,//\, T_{i,k}$$

e, de acordo com o passo 2, formaríamos a árvore indicada na Figura 7.3, à esquerda.

A árvore de execução de P_{111} é obtida imediatamente pelo passo 1. Um nível de recursão a mais e teremos P_{112}. Neste ponto, o resultado seria como indicado à direita na mesma figura. Restaria apenas P_{12}. O passo 2 seria aplicável novamente e o resultado final é mostrado na Figura 7.4, concluindo o exemplo.

Da última figura está claro como a hierarquia presente em $A(P)$ reflete a estrutura de parêntesis embutida em P. A particular maneira como o exemplo foi ilustrado, ou seja, construindo-se o nível mais alto seguido do detalhamento dos níveis mais baixos, não reflete fielmente a maneira de operar do procedimento recursivo. Este último caminha em sentido inverso, isto é, constroi primeiro as subárvores mais elementares para só depois interligá-las a uma raiz comum. O resultado final, claro, será o mesmo.

Considere, agora, as árvores de execução mostradas na Figura 7.5. A árvore da esquerda corresponde ao programa P dado por $P = (\, P_1 \,//\, (P_2 \,//\, P_3\,) \,//\, P_4\,)$ o qual, obviamente, é equivalente ao programa $P' = (\, P_1 \,//\, P_2 \,//\, P_3 \,//\, P_4\,)$ representado pela árvore de execução à direita, na mesma figura. A conclusão a se tirar é que, em geral, um nó, seja x, com rótulo "//", tendo como descendente imediato outro nó, seja y, também com rótulo "//", representa redundância desnecessária. A árvore de execução pode ser refeita eliminando-se o nó y e colocando-se as subárvores imediatas de y como descendentes imediatos de x. Este processo pode ser repetido até que em nenhum caminho da raiz principal até uma folha encontremos dois rótulos "//" consecutivos. O mesmo pode ser dito com respeito ao rótulo ";". Em conclusão, pode-se obter, para cada programa P da classe PR, uma *árvore de execução na forma canônica*. Esta forma canônica é caracterizada pelo fato de que em toda sequência de rótulos encontrados ao se percorrer o único caminho desde a raiz até uma folha, os rótulos "//" e ";" aparecem alternadamente. Um programa P da classe PR é tido como um *programa na forma canônica* se a árvore de execução $A(P)$, obtida aplicando-se o procedimento recursivo descrito acima, for da forma canônica. Suporemos que o processador de comandos da LMD ao construir um programa da classe PR ou, equivalentemente, sua árvore de execução, produzirá sempre uma estrutura na forma canônica. Note que, dado um programa qualquer ou sua árvore de execução construidos pelo processador de comandos da LMD, seria bastante fácil anexarmos um outro módulo a este processador que reduziria o produto final a sua forma canônica.

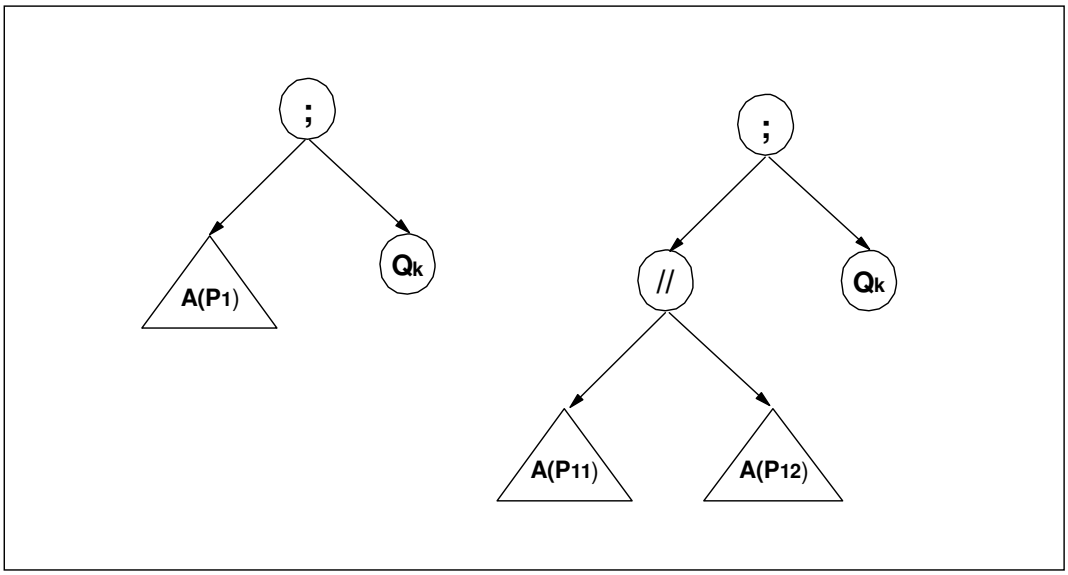

Figura 7.2 - Construindo uma árvore de execução - I

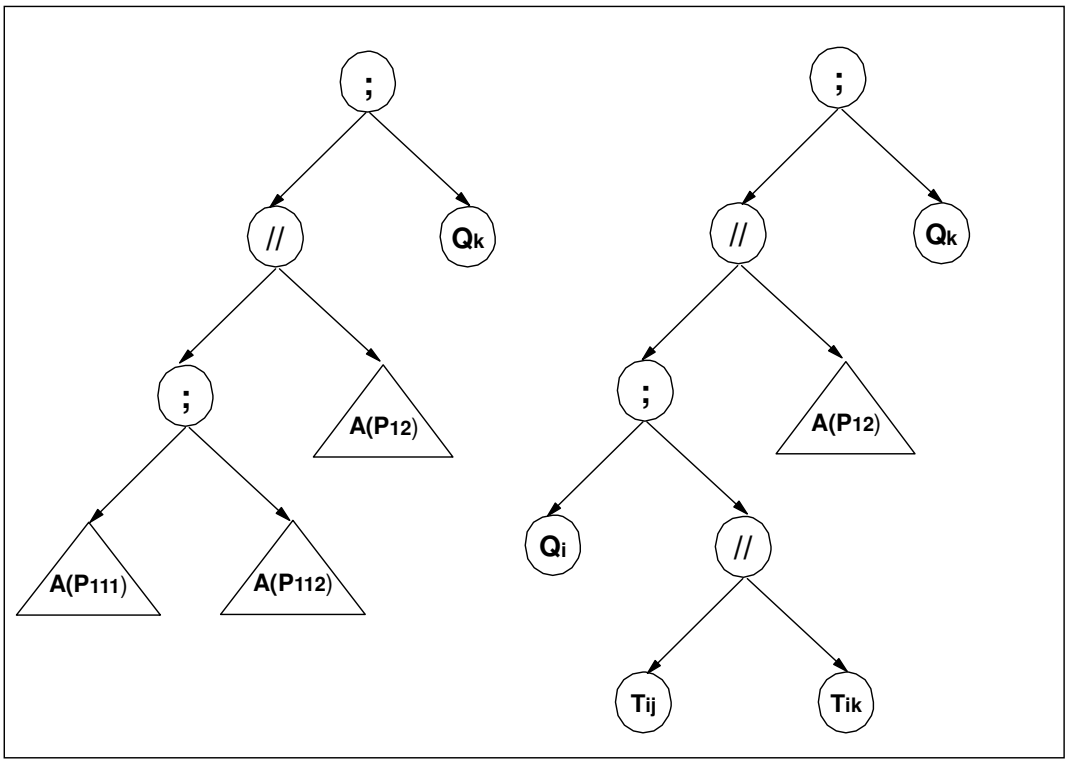

Figura 7.3 - Construindo uma árvore de execução - II

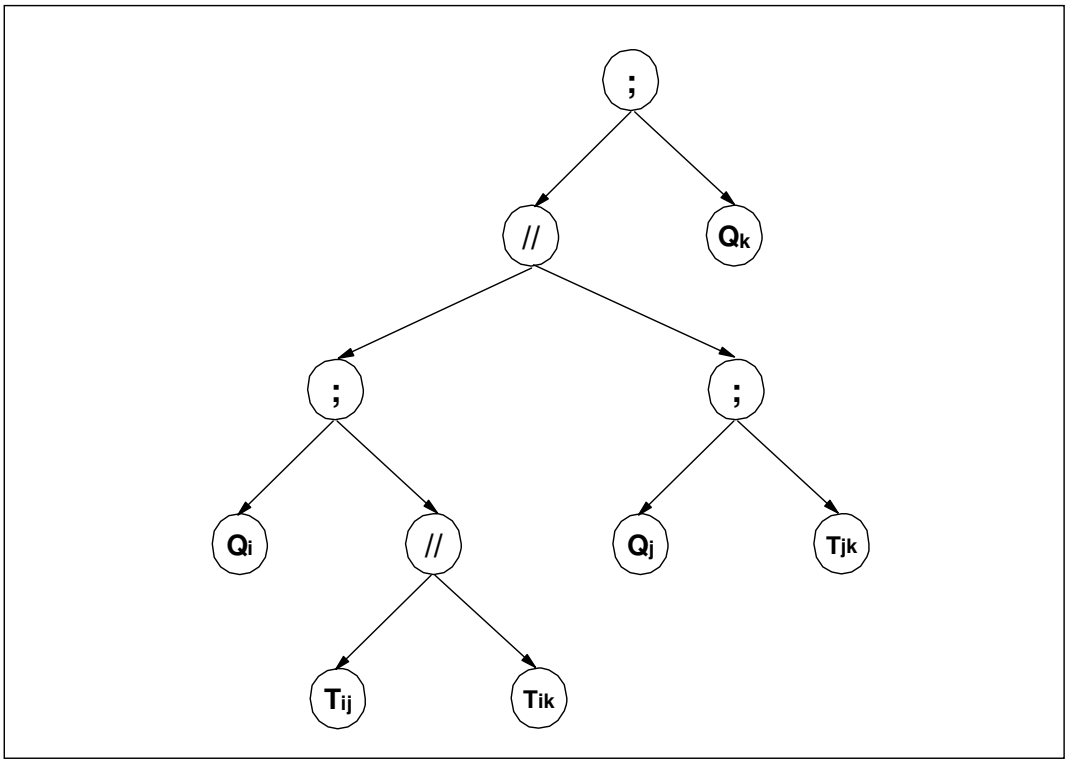

Figura 7.4 - Construindo uma árvore de execução - III

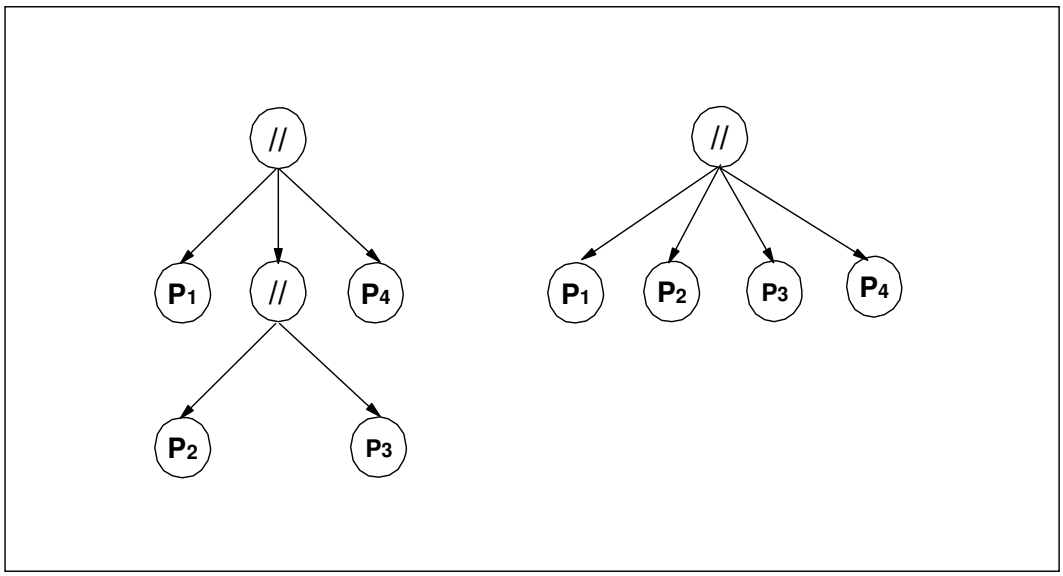

Figura 7.5 - Duas árvores de execução equivalentes

É importante observar também que para cada programa *P* da classe *PR* o método descrito acima produz um única árvore *A(P).* A recíproca também vale, ou seja, para cada árvore *A* produzida pelo algoritmo podemos reconstruir o único programa do qual *A* é a árvore de execução. Esta bijeção garante que podemos substituir o plano de execução da transação, *P*, pela sua árvore de execução, *A(P)*, sem quaisquer prejuízos. Como ilustração do processo operando de maneira inversa, o leitor pode verificar que a árvore da Figura 7.6 representa o seguinte programa da classe *PR*

$$P = ((\ Q_1\ ;\ T_1\ )\ //\ Q_2\ //\ ((\ A_1\ ;\ Q_3;\ T_2)\ //\ T_3\ ))$$

Existe um outro ponto a considerar. Na prática, cada árvore *A(P)* seria implementada como uma estrutura de dados multiligada. Não deve ser inferido, porém, como ilustrado na Figuar 7.4, que todas as árvores de execução terão estrutura binária. Isto se deve apenas a uma particularidade do exemplo que foi então discutido. Em princípio, cada nó da árvore de execução pode ter qualquer número de descendentes diretos. Em consequência, na implementação real cada nó da árvore deverá ter uma lista de ponteiros, um para cada descendente direto deste nó. Há métodos clássicos para se representar árvores cujos nós podem ter multiplos descendentes de modo a limitar o número de ponteiros em cada nó. A Figura 7.7 usa de uma destas representações para reconfigurar a árvore apresentada na Figura 7.6.

A metodologia para se obter a árvore binária é simples

1. as ligações de cada nó com seu descendente imediato mais à esquerda são mantidas; as ligações com os demais descendentes imediatos desaparecem

2. é introduzida uma ligação entre cada dois descendentes imediatos consecutivos de um mesmo nó

As linhas tracejadas na figura são ligações auxiliares, também chamadas de alinhavos. Há um alinhavo indo de cada último descendente imediato de um nó até seu ancestral imediato. Esta particular maneira de transformar árvores gerais em árvores binárias tem a vantagem de que a cada árvore geral corresponde uma única árvore binária e vice-versa. As notas bibliográficas contêm referências a textos que tratam este assunto em detalhe. Suporemos, de ora em diante, que a árvore de execução produzida pelo processador de comandos da LMD seja sempre na forma binária. Como já acabamos de alertar, isto não implicará em perda de generalidade. Desta forma, poderemos nos referir ao descendente imediato esquerdo e descendente imediato direito de cada nó da árvore sem introduzir ambigüidades.

7.3.4 A Migração

De posse das árvores de execução, a intuição que guiará o executor de transações durante a migração de transações fica transparente. O racional é bastante simples. O ponto de partida é sempre um programa da classe PR enviado pelo processador de comandos da LMD ao executor de transações local. O nó de origem da transação controla todo o processo. De início, envia mensagens de trabalho a quantos nós participantes for possível de modo a atingir o máximo de paralelismo. A seguir, espera pelas mensagens de resposta destes nós confirmando o sucesso de cada operação. À medida que respostas venham chegando, o nó de origem repete o processo descobrindo que novas operações poderão ser executadas em paralelo enviando-as, em seguida, para os nós participantes onde devem executar. Após aguardar novas mensagens de resposta, o ciclo é repetido até completar o processo de migração desta transação.

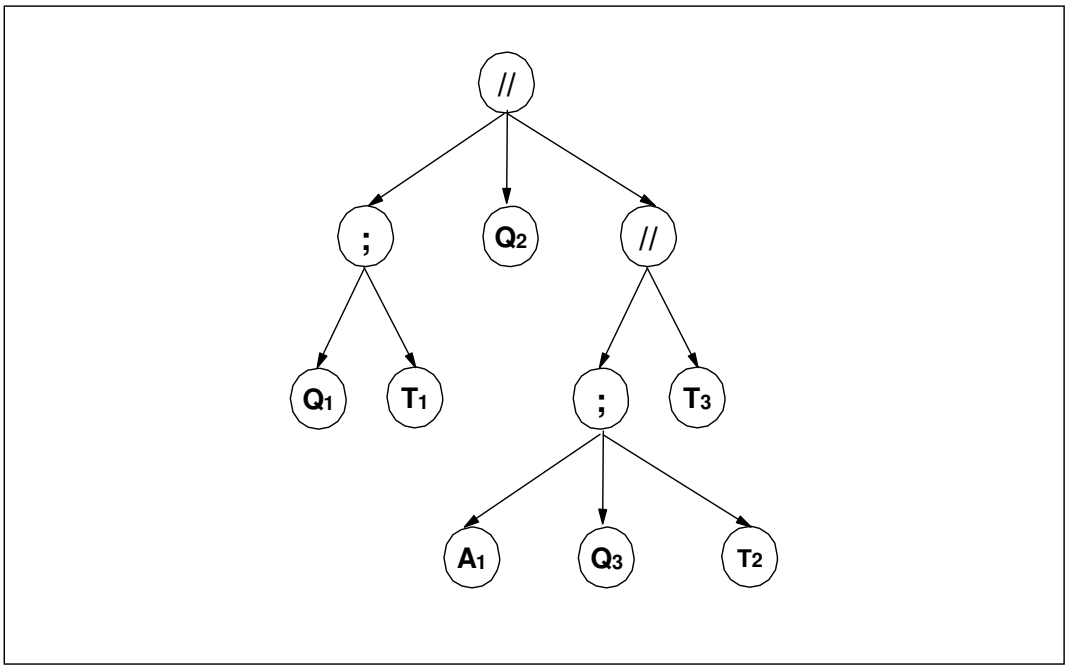

Figura 7.6 - Uma árvore de execução

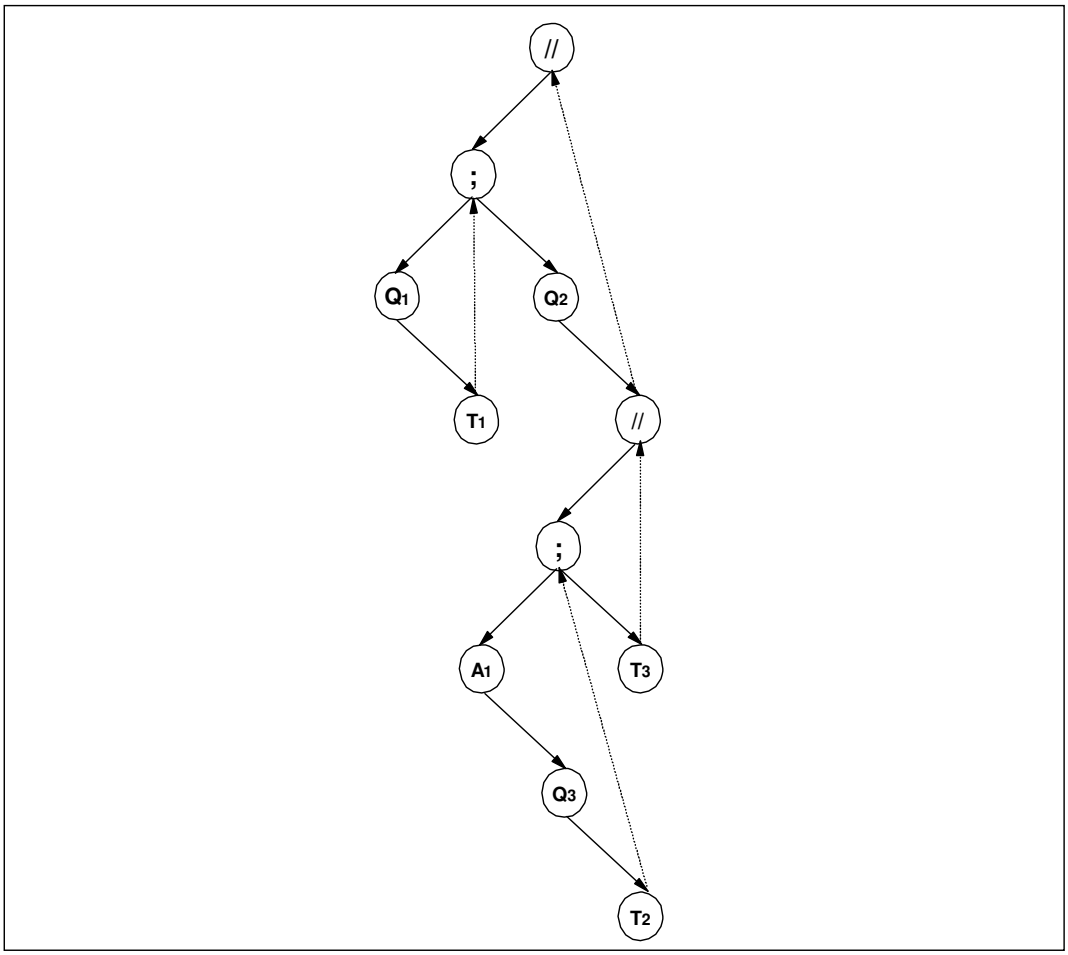

Figura 7.7 Uma árvore de execução com estrutura binária

Durante o processo de migração, e na ausência de falhas, o *nó de origem* executa os seguintes passos básicos.

1. iniciação;
2. envio de mensagens;
3. execução de computações locais, se houver;
4. análise de mensagens recebidas;
5. se a migração da transação ainda não terminou, retorne ao passo 2, caso contrário passa a gerenciar o término da transação.

Já um *nó participante*, operando como um escravo, perfaz os seguintes passos eternamente (falhas receberão um tratamento especial):

1. os agentes locais executam suas tarefas, se houver;
2. quando chegarem mensagens de trabalho, o executor de transações local inicia e instala os respectivos agentes locais, liberando-os em seguida para que executem suas tarefas;
3. quando um agente local terminar seu trabalho, comunica tal fato ao executor de transações local. Este último envia uma mensagem de resposta ao nó de origem da transação, reportando o veredito à cerca da execução local.

Note, em primeiro lugar, que esta sistemática vale para cada transação que esteja ativa no BD a cada instante. Em particular, pode haver várias transações com o mesmo nó de origem ou usando os mesmos nós remotos comuns. Em qualquer caso, as transações são individualizadas pela identificação que recebem ao passarem pela fase de iniciação executada pelo gerente de transações de seu respectivo nó de origem. É responsabilidade dos mecanismos de controle de concorrência garantir a não interferência de ações elementares pertencentes a transações distintas e executadas contra o BD físico local. Isto posto, podemos assumir a ilusão de que apenas uma transação de cada vez está ativa em todo o BD. Assim, para entendermos o processo de migração, basta examinarmos a operação do executor de transações em seu nó de origem, bem como a operação de um nó participante típico.

Para facilitar o entendimento, será apresentado a seguir um mecanismo algo simplificado e, portanto, com certas ineficiências. Quando este estiver claro, as necessárias correções serão mencionadas e o quadro estará completo. As duas principais hipóteses simplificadoras que assumiremos momentaneamente dizem respeito a operação de um nó participante e a falhas. Talvez a diferença mais gritante esteja no fato de que falhas serão, de momento, ignoradas. Em adição, no cenário simplificado, o nó de origem envia apenas *operações básicas* aos nós remotos. Mais tarde, o nó de origem terá liberdade para enviar *subprogramas completos* envolvendo várias operações básicas.

Tomando primeiro o caso de um nó remoto e assumindo que receba apenas operações básicas, não há o que detalhar em relação aos passos básicos descritos acima para este tipo de nó. A parte de iniciação e instalação de agentes locais já foi discutida na subseção anterior. Quanto a execução de uma operação básica, basta lembrar que o gerente de dados local a traduz para a LMD local, entregando a tradução ao SGBD local. Este, por sua vez, gera ações elementares e aciona o agente local, já instalado, para que as execute. Em princípio, a construção de mensagens de resposta pode ficar a cargo do agente local, e seu envio pode ser delegado ao executor de transações. Este último também pode ser o módulo responsável por receber mensagens de trabalho e encaminhar seu processamento entre os módulos locais.

Os mesmos comentários valem com relação aos passos 1 e 3 do algoritmo básico apresentado acima para o nó de origem exceto que o passo 1, de iniciação, é um pouco mais elaborado, conforme vimos na subseção anterior. O tratamento do término de transações, previsto no passo 5, será alvo da próxima subseção. Resta examinar os passos 2 e 4. O primeiro corresponde a como proceder para identificar quais as operações básicas que podem ser encaminhadas em paralelo. O segundo diz respeito a como reestruturar a árvore de execução com base nas mensagens de resposta que porventura cheguem. Analisaremos estes passos em seguida. Lembre que estamos trabalhando com árvores na forma canônica e binárias.

Em primeiro lugar, vamos ao algoritmo para decidir, a cada instante, quais operações básicas podem ser enviadas em paralelo para execução nos vários nós remotos. O procedimento consiste em visitar cada nó da árvore seguindo uma ordem bem definida. O percurso através da árvore é guiado pelo rótulo de cada nó. Há duas maneiras de se obter o mesmo resultado final, qual seja, a remessa do maior número de operações que podem ser executadas em paralelo neste momento segundo previsto na árvore de execução que temos em mãos. Pode-se acumular as operações básicas em uma lista à medida que o algoritmo percorre os nós da árvore e, ao final, despachar todas as operações aos respectivos nós remotos. Ou pode-se acionar um módulo separado para que, à medida que estas operações sejam identificadas, este último já as despache a nós remotos apropriados. Ficaremos com a segunda alternativa. Esta opção torna o algoritmo um pouco mais simples e dispensa a lista de nós. Na realidade, não há diferenças drásticas entre as duas alternativas. Lembre que o algoritmo para identificar os nós executa em um único processador, aquele do nó de origem. Além disso, a árvore de execução deve conter algumas dezenas, no máximo centenas, de nós. Assim, se adotássemos a primeira alternativa, construindo a lista de nós, a execução do algoritmo seria, de qualquer modo, bem mais rápida que o processo de se enviar mensagens através da rede. Para a rede de comunicações tudo se passa como se uma fila de mensagens a serem enviadas se acumulasse instantaneamente, independente do processo de formação da lista de nós. Outro aspecto importante é que a ordem com que as mensagens vão sendo enviadas é imaterial. Lembre que a lista produzida nesta fase constitui-se apenas de operações que podem ser executadas em paralelo.

O algoritmo para despachar mensagens será apresentado recursivamente abaixo através do procedimento ENVIAR_MENSAGENS. Na descrição do procedimento, *P* é o programa *PR* a ser analisado, *A* é a raiz da árvore de execução de *P* e *O* é uma das operações básicas.

1) rótulo de *A* é *O*: despache *O* ao nó apropriado, junto com seu endereço na árvore de execução, e retorne;
2) o rótulo de *A* é "//":
   a) siga pelo ponteiro esquerdo de *A*. Seja *R* a raiz da subárvore encontrada;
   b) invoque ENVIAR_MENSAGENS(*R*) recursivamente;
   c) se o ponteiro direito de *R* for um alinhavo, retorne;
   d) caso contrário, atribua a *R* a raiz da subárvore indicada por este ponteiro
   e) retorne ao passo 2b.
3) o rótulo de *A* é ";":
   a) siga pelo ponteiro esquerdo de *A*. Seja *R* a raiz da subárvore encontrada;
   b) invoque ENVIAR_MENSAGENS(*R*) recursivamente;
   c) retorne.

No passo 2 do algoritmo visitamos sucessivameete cada um dos descendentes imediatos da raiz *A*. Já no passo 3 devemos apenas visitar o descendente imediato mais à esquerda da raiz visto que as operações especificadas por seus descendentes devem ser executadas na ordem indicada. Ao construir a mensagem de trabalho, suporemos que o endereço da operação a ser efetuada, relativo a árvore binária de execução, seja também incluído nesta mensagem. Isto facilitará a análise das mensagens de resposta relativas a cada solicitação de trabalho. Uma posição na árvore binária pode ser facilmente indicada através de uma sequência de caracteres "e" e "d". Neste esquema "e" significaria "à esquerda" e "d" indicaria "à direita". O nó rotulado *"$T_{j,k}$"* na Figura 7.4, por exemplo, teria sua posição indicada por edd.

Um exemplo completo servirá para clarificar o algoritmo. Assuma que O_1 a O_8 sejam operações básicas e tome *P* como o programa abaixo

$$P = ((\ O_1 \ // \ (\ O_2 \ ; \ O_3 \ ); \ O_4 \ ) \ // \ O_5 \ // \ (\ O_6 \ ; \ O_7 \ ; \ O_8 \ ))$$

A árvore de execução de *P*, *A(P)*, é mostrada na Figura 7.8 . Os nós estão numerados de 0 a 12 para facilitar referências. Vamos agora examinar a invocação ENVIAR_MENSAGENS(*A(P)*). O mecanismo de recursão será simulado usando-se uma pilha auxiliar. O conteúdo da pilha, a cada instante, refletirá as invocações pendentes. A raiz de *A(P)* é rotulada "//" e, de acordo com o passo 2, devemos invocar o algoritmo recursivamente sobre as subárvores cujas raizes são localizadas sobre os nós 1, 2 e 3. Em consequência, este será o conteúdo da pilha, com 1 no topo. Isto será indicado como *PILHA = 1,2,3*. Desempilhando um nó obtemos 1, equivalendo a invocação ENVIAR_MENSAGENS(*A(R)*), onde *R* é a subárvore cuja raiz é o nó 1. Como o nó 1 é rotulado ";", o passo 3 indica que devemos invocar ENVIAR_MENSAGENS sobre o nó 7 apenas. Em termos da pilha, teríamos *PILHA = 7,2,3*. Desempilhando, obtemos o nó 7 e o passo 2 modifica a pilha para *PILHA = 9,10,2,3*. A próxima chamada recursiva teria como argumento a árvore com raiz sobre o nó 9. Agora o passo 1 despacha a operação básica O_1 para seu nó remoto e reduz a pilha a *PILHA = 10,2,3*. A próxima invocação de ENVIAR_MENSAGENS produziria *PILHA = 11,2,3* , usando-se o passo 3. Segundo o passo 1, as próximas duas chamadas resultariam no envio das operações básicas O_2 e O_5 para execução remota e deixariam a pilha na forma *PILHA = 3*. Mais uma invocação do algoritmo produziria, pelo passo 3, *PILHA = 4*. A última chamada recursiva enviaria O_6 a seu destino, esvaziando a pilha. Isto completaria a execução de ENVIAR_MENSAGENS(*A(P)*). A sequência de mensagens enviadas seria O_1, O_2, O_5 e O_6. Examinando o programa *P* e a árvore *A(P)*, podemos constatar que estas seriam exatamente as operações básicas que poderiam ser executadas em paralelo quando a transação iniciasse sua migração. A Figura 7.8 ilustra o fato, destacando estas operações.

Uma vez enviadas as mensagens de trabalho, o nó de origem se ocupa de suas obrigações locais e aguarda mensagens de resposta. Observe que algumas dentre as operações básicas despachadas podem ter sido escaladas para executar no próprio local de origem. Também não está excluído o cenário onde um determinado nó recebe várias operações básicas para execução durante esta fase. É importante salientar que o processo é assíncrono. No exemplo que acabamos de analisar, isto implica em que as mensagens de resposta não estão obrigadas a chegar em qualquer ordem pré-especificada. Na prática, isto dependerá da carga de cada nó remoto que for acionado.

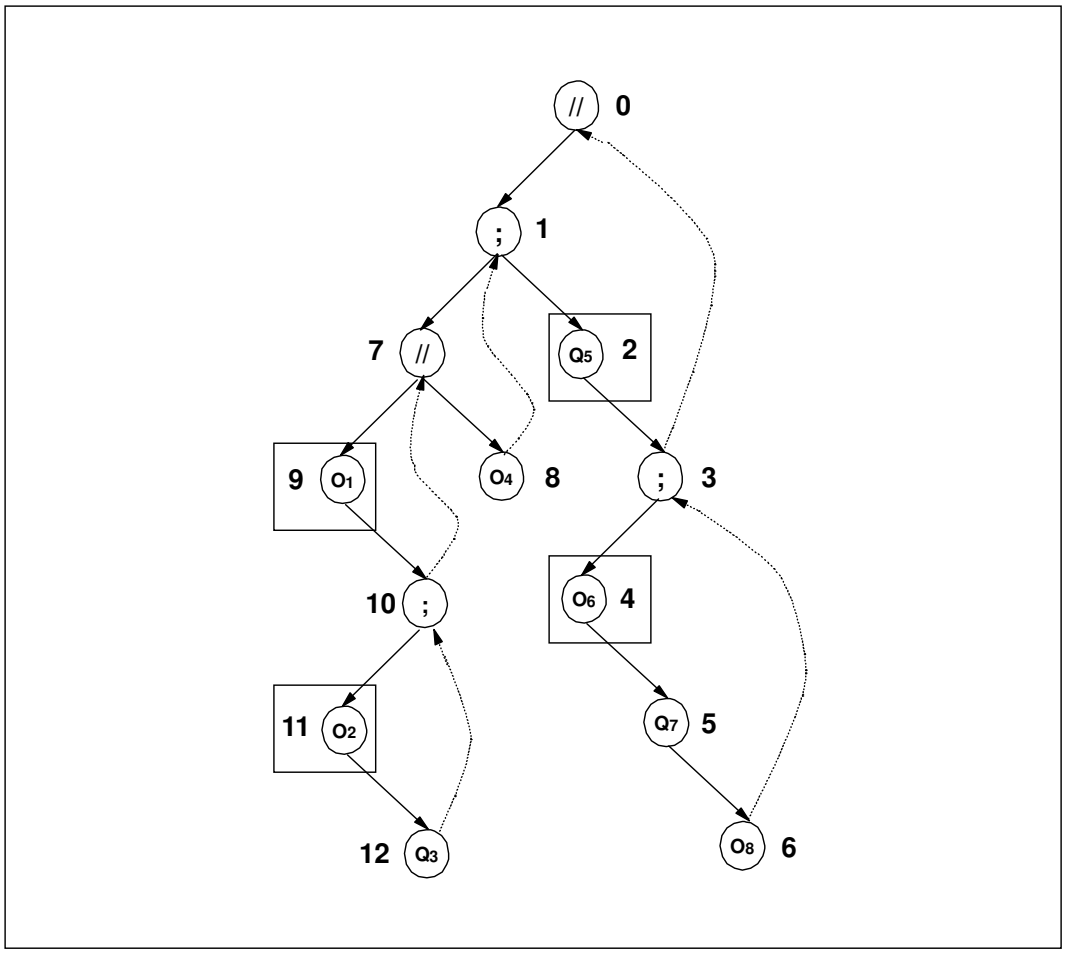

Figura 7.8 - Despachando operações binárias em paralelo.

Voltando ao procedimento original que coordena a migração de transações a partir do nó de origem, resta examinar o passo 4, qual seja, análise das mensagens de resposta que chegam. A idéia é eliminar da árvore cada nó cuja mensagem de resposta tenha chegado e seja positiva. Isto é verificado observando-se o campo de veredito da mensagem de resposta: se contiver EFETUADO, então a resposta é positiva, se for CANCELADO, é negativa. Enquanto essas eliminações sucessivas resultarem em que subárvores inteiras já tenham sido completamente executadas, o processo de eliminação deve ser propagado para os ancestrais do nó. Finalmente, após o processo de eliminação, a parte da árvore afetada deve ser reanalisada para se determinar quais as próximas operações que podem ser executadas em paralelo. Para tal, o procedimento ENVIAR_MENSAGENS é usado. Chamaremos o processo de análise de mensagens de resposta de RECONFIGURAR. Note que a mensagem de resposta contém um campo de identificação e outro de rementente. A partir do primeiro o executor de transações do nó de origem pode localizar a correspondente árvore de execução. Suporemos que no segundo campo o nó remetente inclua o endereço, em relação a árvore de execução binária, da operação básica que acabou de completar. O nó remetente conhece este endereço pois esta informação veio com a mensagem de trabalho. A descrição precisa de RECONFIGURAR segue abaixo. Nesta descrição, *A* é a árvore de execução e *END* aponta para o nó a que se refere esta mensagem de resposta. O rótulo *O* indica uma operação básica enquanto que *X*, *Y* e *Z* são variáveis auxiliares.

1) se *END* for a raiz da árvore, então RECONFIGURAR e a migração desta transação estão terminadas. Caso contrário, partindo de *END*, siga pela direita até encontrar um nó com alinhavo. Seja *Z* o nó para onde o alinhavo aponta;

2) *END* não é o descendente imediato esquerdo de *Z*:

   a) a partir de *Z*, siga pela esquerda e depois sempre pela direita até encontrar um nó cujo ponteiro direito aponte para *END*. Seja *Y* este nó.

   b) elimine *END* da árvore. Basta atribuir ao ponteiro direito de *Y* o valor do ponteiro direito de *END*, mesmo que este seja um alinhavo.

3) *END* é o descendente imediato esquerdo de *Z:*

   a) O ponteiro direito de *END* não é um alinhavo:

      i) caminhe uma vez pela direita a partir de *END*. Seja *X* o nó encontrado;.

      ii) elimine *END* da árvore atribuindo *X* ao valor do ponteiro esquerdo de *Z*;

      iii) se o rótulo de *Z* for ";", execute ENVIAR_MENSAGEM(*X*);

   b) O ponteiro direito de *END* é um alinhavo:

      i) enquanto o ponteiro direito de *END* for um alinhavo e *END* for distinto da raiz da árvore, caminhe pelo alinhavo. Chame de *END* o nó encontrado.

      ii) se *END* for a raiz da árvore, RECONFIGURAR e também a migração da transação estão terminadas.

      iii) se *END* não for a raiz da árvore:

         (1) siga sempre pela direita até encontrar um nó com alinhavo. Seja *Z* o nó para onde o alinhavo aponta;

         (2) seja *X* o nó para onde o ponteiro direito de *END* aponta;

         (3) elimine *END* da árvore. Basta atribuir *X* ao valor do ponteiro esquerdo de *Z* ;

         (4) se *Z* for rotulado ";", execute ENVIAR_MENSAGENS(*X*)

Antes de apresentar um exemplo há certos comentários que serão úteis. Lembre que o objetivo é eliminar *END* da árvore. O caso trivial quando toda a árvore de execução se resume em apenas um nó rotulado com uma operação básica é previsto no passo 1. Ignorando este caso simples, o passo 1 posiciona *Z* sobre o ancestral imediato de *END*. A situação é mostrada na Figura 7.9. Note que *END* não pode ter descendente à esquerda pois é uma operação básica. Partindo de *END* e caminhando pela direita até encontrar um nó com alinhavo nos leva ao nó *W*. O alinhavo de *W* aponta para *Z*. Isto feito, há duas possibilidades segundo *END* é ou não o descendente imediato à esquerda de *Z*. O passo 2 trata da segunda possibilidade. Em 2a, alcançamos primeiro *R* e, seguindo pela direita, achamos *Y*. A parte 2b elimina *END* da árvore segundo indica a linha tracejada na referida figura. Outro fato a observar é que, na situação do passo 2, o rótulo de *Z* deve, necessariamente, ser "//". Portanto, todos os "irmãos" de *END*, isto é, os nós de *R* a *W* na Figura 7.9, já foram inspecionados pelo procedimento ENVIAR_MENSAGENS e não devemos repetir o processo neste ponto. Esta observação é crucial. O passo 3 se encarrega do caso quando *END* é o descendente imediato esquerdo de *Z*. Observe que *END* só pode ser o descendente imediato esquerdo de *Z*, como é mostrado na Figura 7.10, porque *Z* foi alcançado a partir de um alinhavo no passo 1.

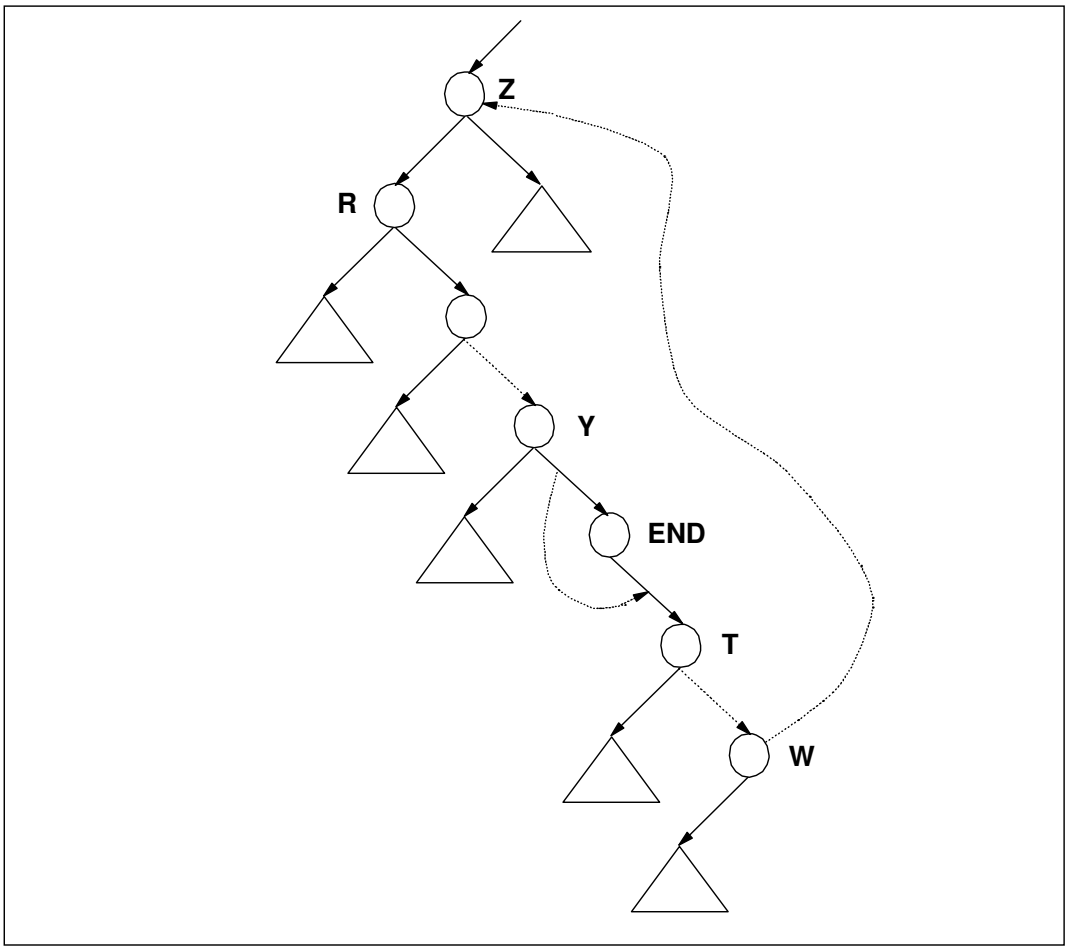

Figura 7.9 - Eliminando uma operação básica . Caso I

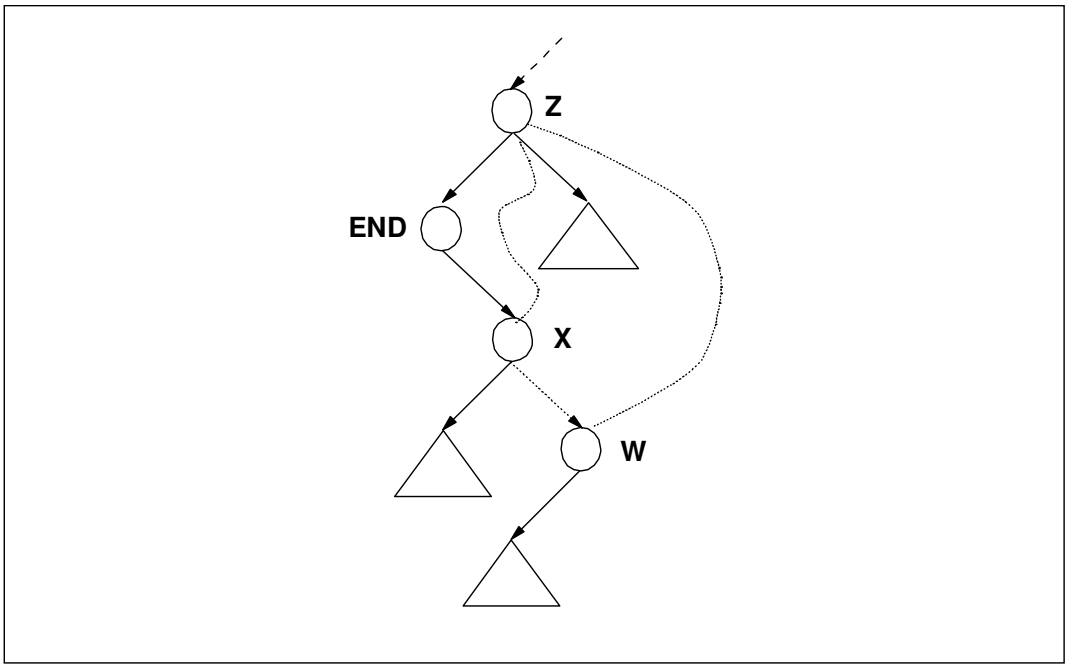

Figura 7.10 - Eliminando uma operação básica. Caso II

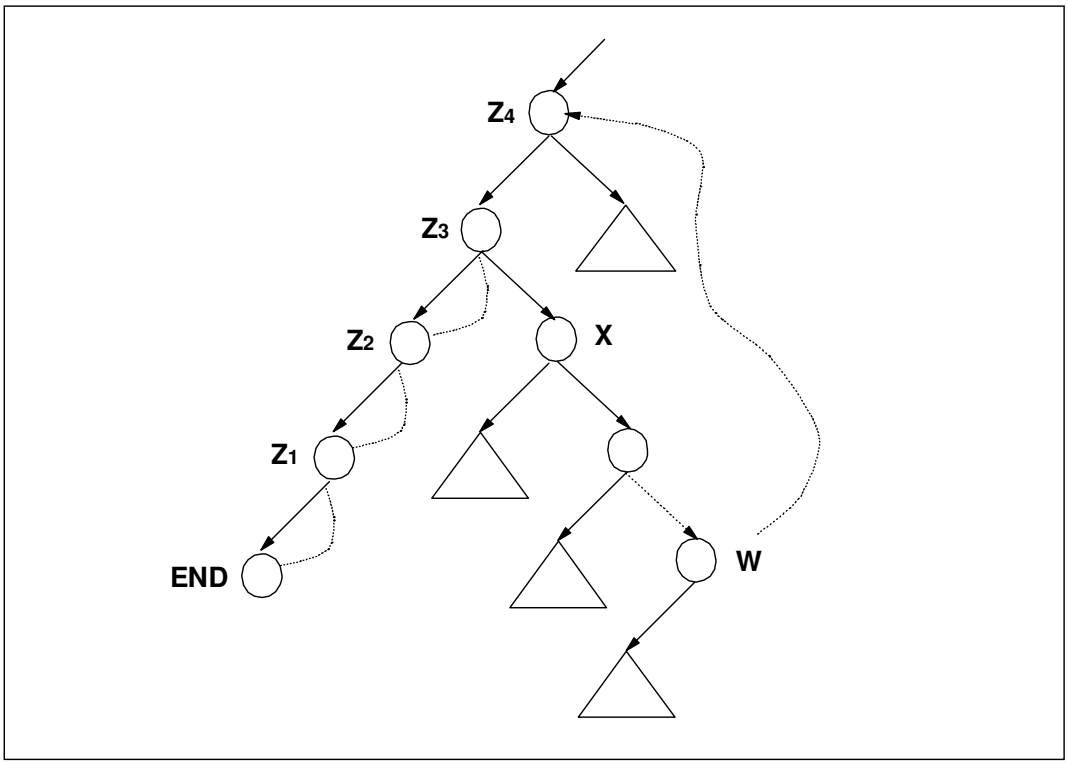

Figura 7.11 - Eliminando uma operação básica. Caso III

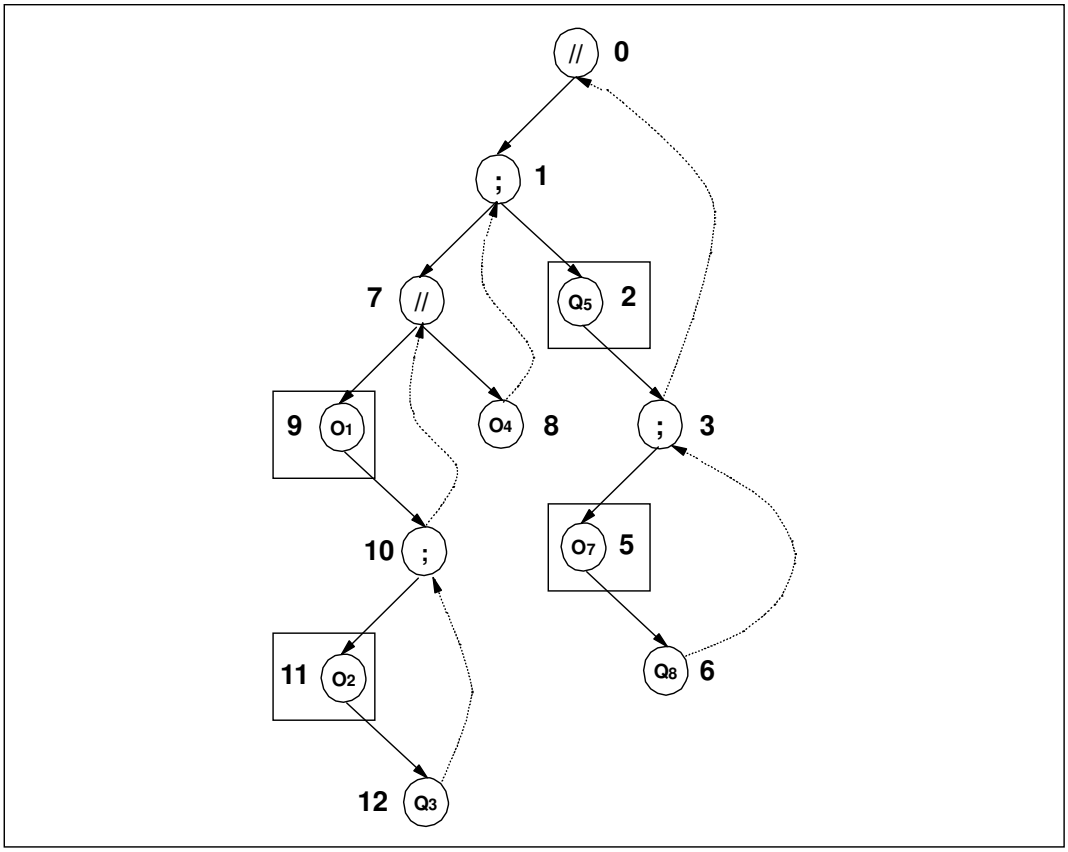

Figura 7.12 - Migração de uma Transação - I

A mesma figura serve ao caso 3a. Nesta situação, o nó *X* é alcançado imediatamente. Para eliminar *END*, basta executar o passo 3ai, conforme indica a linha tracejada. Agora devemos examinar o rótulo de *Z*, o ancestral imediato de *END*. Podemos encontrar tanto "//" como ";". Se for encontrado "//", os "irmãos" de *END* já foram inspecionados por ENVIAR_MENSAGENS e não há mais nada a fazer. Caso contrário, o rótulo é ";" e devemos examinar a próxima subárvore da sequência. Sua raiz é o novo descendente imediato de *Z*, à esquerda. O passo 3aiii se encarrega disto. Finalmente, quando *END* for o descendente imediato esquerdo de *Z* e, além disto, seu ponteiro direito for um alinhavo, chegamos ao caso 3b. Estamos num ponto interessante. Veja a Figura 7.11. Se o rótulo de Z_1 for "//", então todos os descendentes imediatos de Z_1 já executaram suas tarefas, responderam e foram eliminados da árvore de execução. Ou então o rótulo de Z_1 é ";" e *END* é o último nó da sequência de descendentes imediatos de Z_1. Em qualquer caso, toda a subárvore com raiz em Z_1 pode ser agora eliminada. Note que a mesma situação pode estar acontecendo com Z_1 e seu ancestral imediato Z_2, e assim por diante. O efeito do passo 3bi é subir na árvore até encontrar aquele ancestral de *END* que ainda tenha outros descendentes imediatos que não terminaram de executar suas tarefas. É claro que se tal nó não é encontrado chegaremos a raiz da árvore e, então, todos os nós já terminaram suas tarefas e o processo de migração está completado. Na figura, o passo 3bi é repetido 3 vezes até chegarmos a Z_3. A partir daí as próximas ações já são conhecidas.

Reconsidere a árvore de execução mostrada na Figura 7.8. Já vimos que numa primeira invocação de ENVIAR_MENSAGENS seriam despachadas as operações básicas O_1 O_2, O_5 e O_6. Indicaremos isto com $OP = \{ 1, 2, 5, 6 \}$. As mensagens de resposta chegarão em ordem imprevisível. Suponha que a mensagem de resposta do nó 4 chegue primeiro. Executando RECONFIGURAR(4), passaríamos pelo passo 3a, obtendo a árvore da Figura 7.12 e despacharíamos O_7, resultando em $OP = \{ 1, 2, 5, 7 \}$. Em seguida chegam as mensagens de resposta relativas aos nós 2 e 11. A invocação de RECONFIGURAR(2) e RECONFIGURAR(11) resultaria na árvore da Figura 7.13 e em $OP = \{ 1, 3, 7 \}$. Chegando a mensagem de resposta relativa ao nó 12 e, em seguida, àquela correspondente ao nó 5, a situação evolui para a Figura 7.14 com $OP = \{ 1, 8 \}$. A mensagem de resposta relativa ao nó 9 produziria a árvore da Figura 7.15 e $OP = \{ 4, 8\}$. O processo é completado com as mensagens de resposta vindas dos nós 8 e 6. O leitor está convidado a repetir a análise supondo outras ordens de chegada para as mensagens de resposta.

Os procedimentos ENVIAR_MENSAGENS e RECONFIGURAR completam uma primeira versão do metodo básico usado pelo nó de origem para controlar a migração de transações, como havíamos posto logo nó início desta subseção. Uma das ineficiências que havíamos então mencionado pode ser agora apreciada.

Considere a árvore de execução *A* mostrada na Figura 7.16. Nesta figura Q_m representa uma consulta que deve ser dirigida ao nó *m* e T_{nm} uma transferência de tabelas do nó *n* para o nó *m.* Suponha que o nó de origem seja *k.* Ao invocar ENVIAR_MENSAGENS o executor de transações em *k* vai, ao longo do tempo, mandar 4 mensagens para o nó *i*: uma para cada um dentre Q_{i1} , Q_{i2} , T_{ij} e T_{ik} . Desta forma, os custos inerentes a cada mensagem que é enviada, tais como espaço para endereços, para códigos corretores / detetores de erros, múltiplas confirmações (do ingles "akcs") exigidas pelos protocolos de comunicação, seriam duplicados 4 vezes neste exemplo simples. Sem falar na degradação do tempo de resposta do sistema em consequência de mais mensagens enviadas. Seria bem mais racional se toda a subárvore de execução com raiz sobre o nó 2 fosse enviada ao nó *i* numa única mensagem de trabalho. Em paralelo, a subárvore com raiz em 1 seria enviada ao nó *j*.

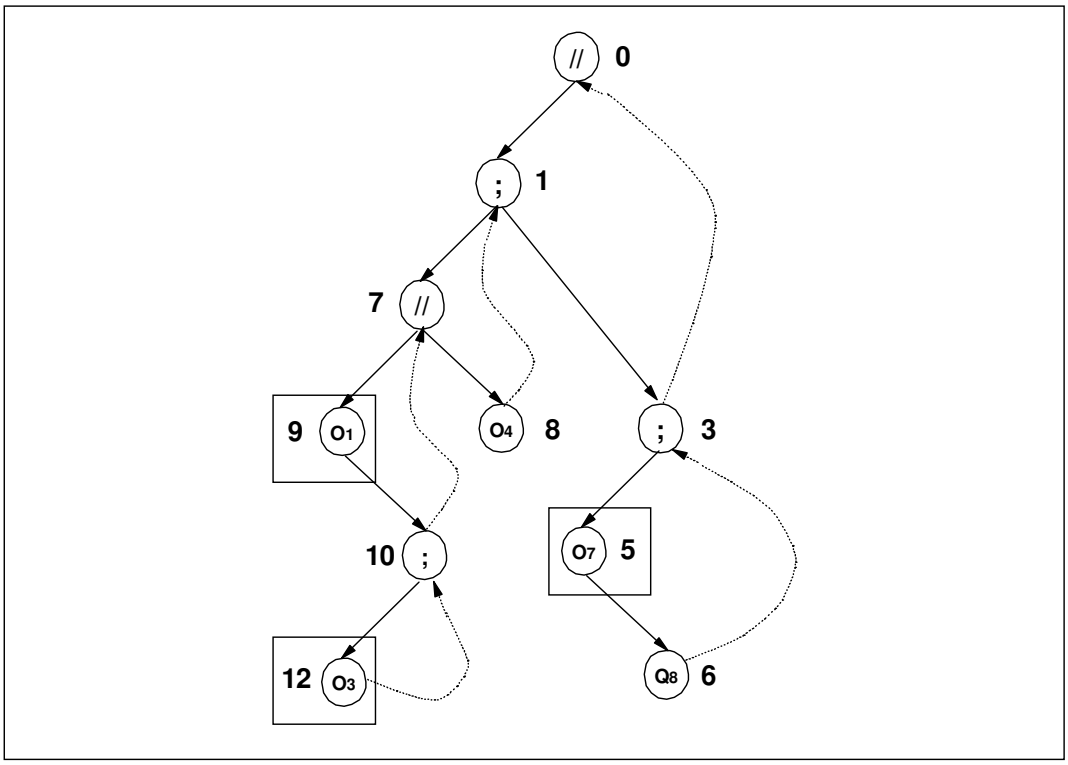

Figura 7.13 - Migração de uma Transação - II

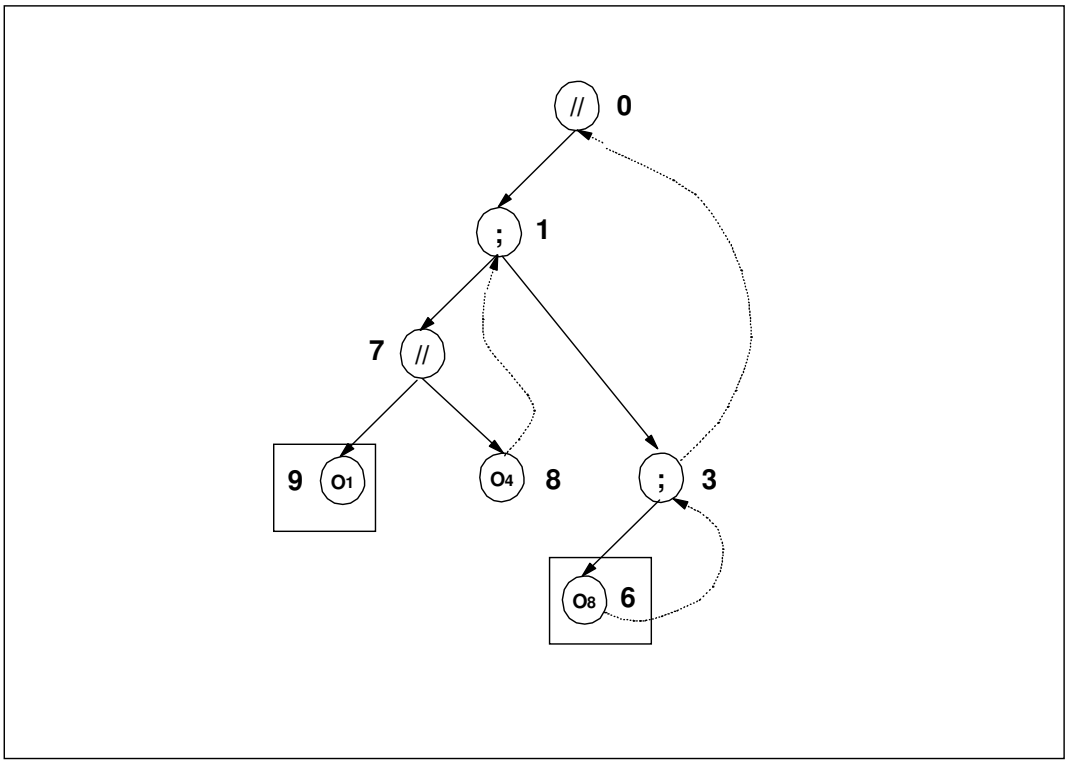

Figura 7.14 - Migração de uma Transação - III

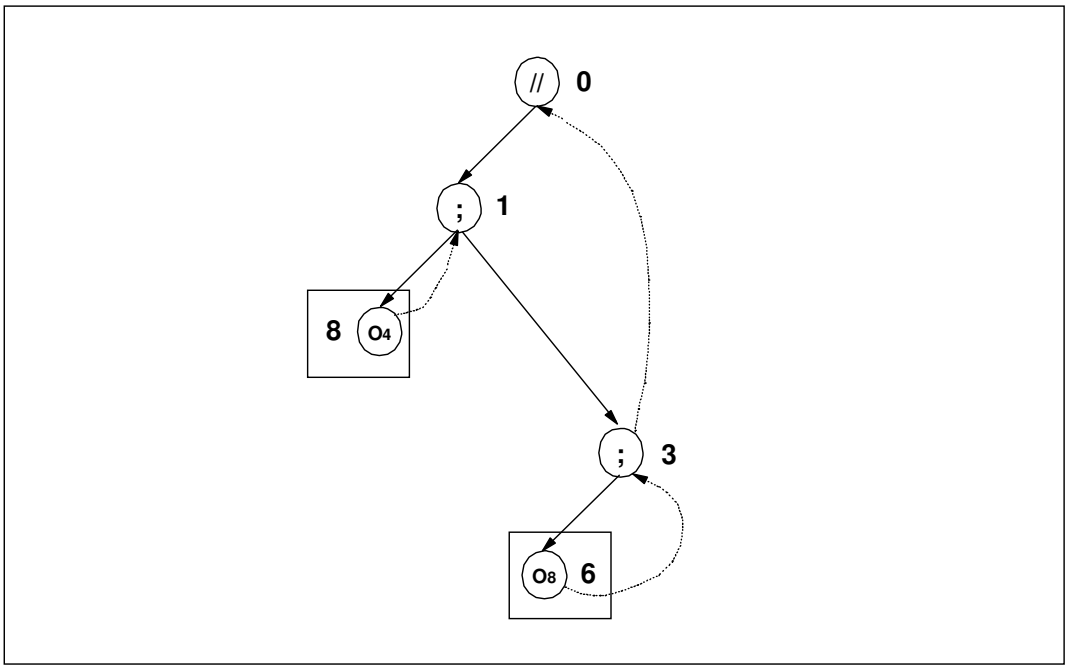

Figura 7.15 - Migração de uma Transação - Conclusão

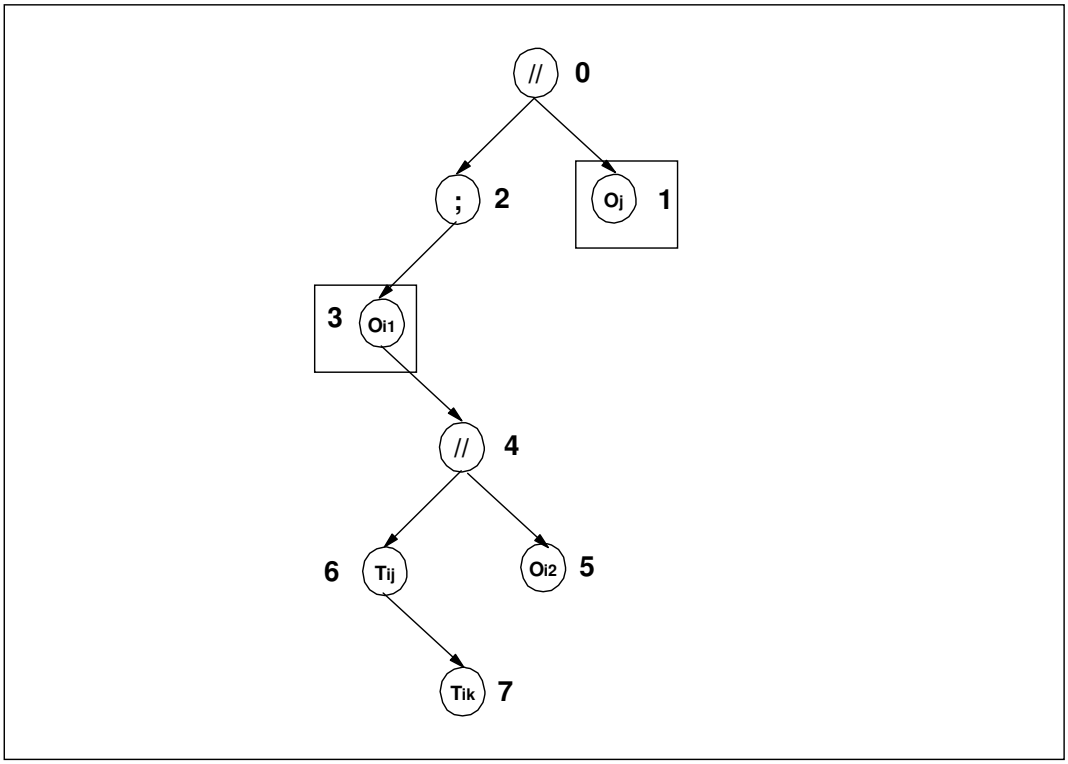

Figura 7.16 - Enviando Subárvores Completas a Nós Remotos

Note que este esquema também economiza trabalho local em *i* pois este teria que iniciar e instalar apenas um agente local para esta transação. Gostaríamos, portanto, de poder determinar não apenas quais as operações básicas que podem ser enviadas em paralelo a cada instante, mas quais sub-árvores poderão ser despachadas em paralelo a cada momento.

Vamos agora descrever uma maneira simples de se identificar sub-árvores completas a serem despachadas a nós remotos. Para tal, o executor de transações do nó de origem deve, logo antes de iniciar a migração da transação, invocar o procedimento recursivo MARCAR que será descrito em seguida. Este procedimento incorpora um algoritmo para marcar cada nó da árvore de execução com a identidade de um nó remoto. A idéia é que se um nó *i* é marcado com a identidade de outro nó *j* então toda a subárvore da qual *i* é a raiz deve ser enviada ao nó *j* durante a fase de migração. Lembre-se, porém, que estamos interessados em identificar subárvores máximas no sentido de que não enviaremos ao nó *j* a subárvore com raiz em *i* se o ancestral imediato de *i* tiver todos os seus descendentes imediatos, além de *i*, também endereçados a *j*. Neste caso, a subárvore "maior" com raiz no ancestral imediato de *i* será enviada a *j*. Ou talvez uma subárvore ainda mais abrangente, da qual o ancestral imediato de *i* é apenas um nó interno, deverá ser encaminhada ao nó *j*. Observe também que certos nós terão parte de seus descendentes imediatos despachados a determinados nós e parte a outros nós. Nestes casos, tais nós não serão marcados com a identidade de nenhum outro nó, pois que a subárvore de que são raizes sera desmembrada e remetida a vários nós distintos. O procedimento MARCAR identifica tais nós com um "*".

Além deste passo de identificação, o procedimento MARCAR altera localmente a estrutura da árvore de execução. A razão para estas alterações está ilustrada na Figura 7.17.

À esquerda temos parte de uma árvore de execução. A raiz é um nó rotulado "//", o qual detém 5 descendentes imediatos, cada um deles, possivelmente, terá toda uma subárvore ligada a seu ponteiro esquerdo. Como mostra a figura, os nós receberam os números de 1 a 6 à guisa de identificação e as subárvores foram chamadas de T_1 a T_5. Suponha que, por algum processo, chegou-se a conclusão que o nó 2 é a raiz de uma subárvore que deve ser executada remotamente pelo nó *i*. Idem, o nó 5 identifica uma outra subárvore que deve ser enviada ao nó *k*, enquanto que 3, 4 e 6 são raizes de subárvores que devem ir para o nó *j*. Isto é mostrado na Figura 7.17, indicando-se o nó de destino logo acima da raiz da subárvore. Note que o nó 1 recebeu a marca "*" pois seus descendentes imediatos são raizes de subárvores que devem seguir para nós remotos distintos. Para minimizar o número de mensagens enviadas, a árvore de execução é transformada como indicado na mesma figura, à direita. Foi criado um novo nó, *n*, e os nós 3, 4 e 6, originalmente endereçados a *j*, foram agrupados como descendentes imediatos de *n*. Note que a marca de *n* deve ser, obviamente, *j*. O ponto importante a observar é que esta manobra se torna possível porque o nó 1 é rotulado com "//". Isto indica que a ordem de execução de seus descendentes imediatos é imaterial.

Dado este fato, a subárvore à direita na figura é operacionalmente equivalente à original, à esquerda. Agora, a subárvore maior com raiz em 1 pode ser executada enviando-se apenas 3 mensagens: uma para o nó *i* descrevendo a subárvore com raiz em 1, outra para *k* descrevendo a subárvore com raiz em 5 e uma terceira para *j*, descrevendo a subárvore com raiz em *n*. Antes da transformação, estaríamos enviando 5 mensagens para executar a as mesmas operações. A Figura 7.18 ilustra a mesma idéia aplicada a um nó rotulado ";". O mesmo mecanismo ainda é usado apenas que devemos agora respeitar a ordem de execução das subárvores, conforme especificado inicialmente. Por esta razão, o novo nó agrupa apenas as subárvores que tinham os nós 2 e 3 como raizes originais.

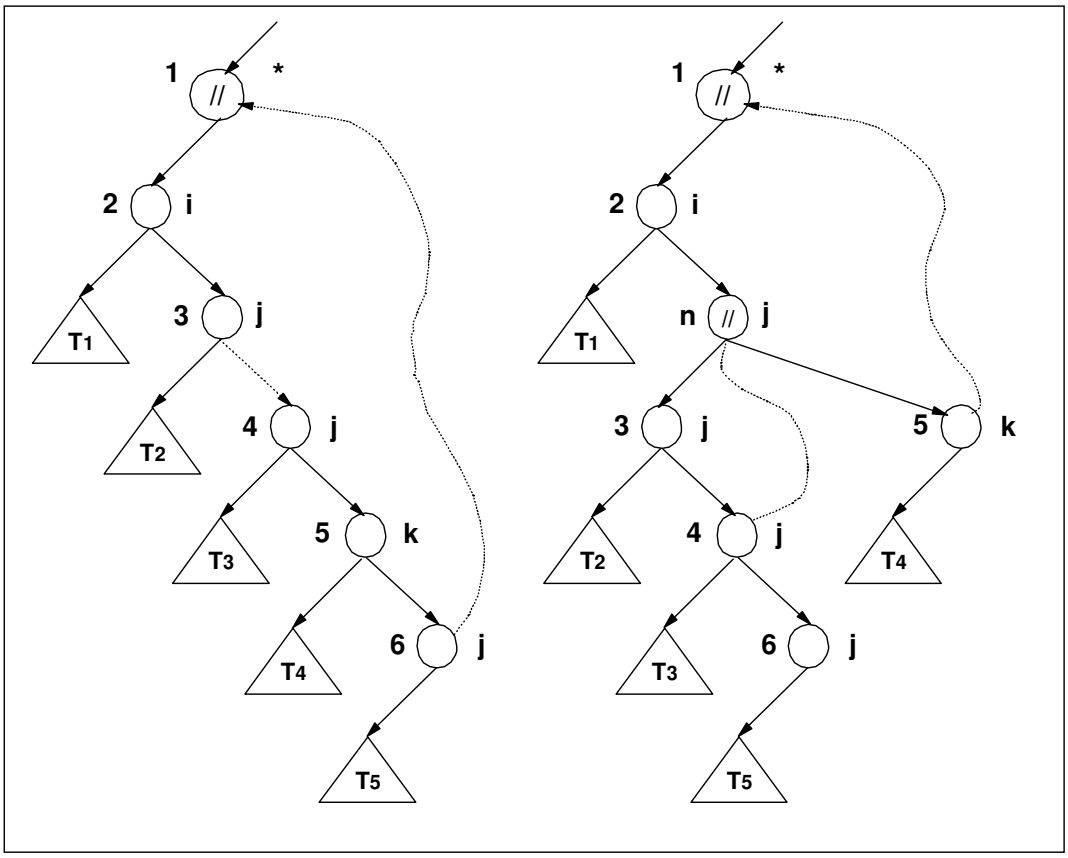

Figura 7.17 - A transformação local operada por MARCAR - I

Na descrição do procedimento MARCAR, em seguida, *A* representa a raiz da árvore de execução e *O* é uma operação básica.

1) se *A* está rotulada com *O*: marque este nó com a identidade do nó remoto onde esta operação básica deve executar e retorne;
2) se *A* está rotulada com "//":
   a) recursivamente, invoque MARCAR sobre cada um dos descendentes imediatos de *A;*
   b) se todos os descendentes imediatos de A tem a mesma marca, então marque *A* de forma idêntica e retorne;
   c) caso contrário:
      i) execute GRUPARP(*A*);
      ii) marque *A* com "*" e retorne.
3) se *A* está rotulada com ";":
   a) recursivamente, invoque MARCAR sobre cada um dos descendentes imediatos de *A* ;
   b) se todos os descendentes imediatos de *A* tem a mesma marca, então marque *A* de forma idêntica e retorne;
   c) caso contrário:
      i) execute GRUPARS(*A*);
      ii) marque A com "*" e retorne.

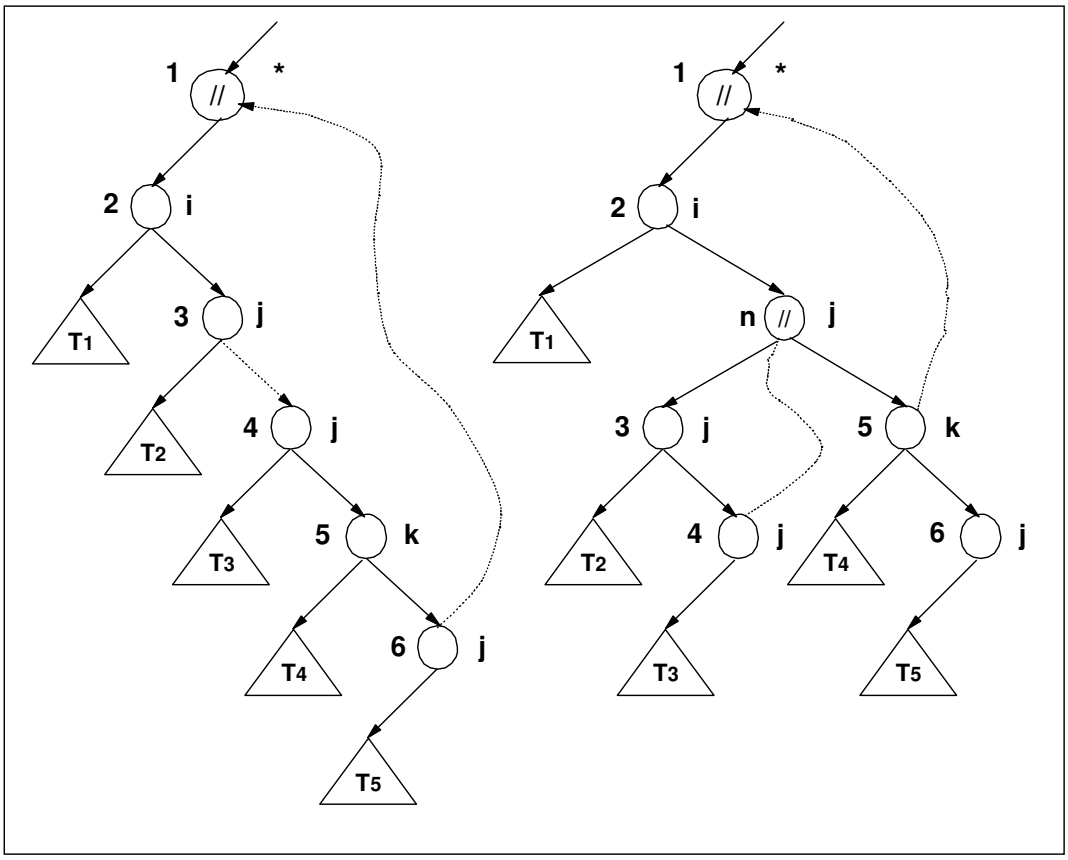

Figura 7.18 - A transformação local operada por MARCAR - II

A lógica do procedimento é transparente e dispensa maiores comentários. Está claro que GRUPARP tem como função agrupar os descendentes imediatos de um nó interno rotulado "//". Já GRUPARS perfaz a mesma tarefa quando o rótulo do nó em questão é ";". A descrição dos procedimentos GRUPARP e GRUPARS constitui-se num bom exercício em estruturas de dados. O leitor deveria tentar esquematizá-los antes de ler a solução proposta abaixo.

Seja *R* a raiz de uma árvore de execução. Uma solução simples para GRUPARP usaria uma tabela auxiliar onde cada linha indica um grupo distinto de nós que será construído. Cada linha é formada por 4 campos: *CONT*, *IDENT*, *INIC* e *FIM*. Os campos *INIC* e *FIM* são ponteiros para o começo e fim de uma lista encadeada de nós que formarão um grupo. O campo de *CONT* é um simples contador que indicará o número de nós já arrebanhados para um mesmo grupo. O campo *IDENT* conterá o endereço do nó remoto para onde este grupo será despachado. Finalmente, *x*, *y*, *Z* e *i* são variáveis auxiliares.

1) desça pela esquerda de *R*, achando o nó *Z*.
2) enquanto *Z* for diferente de *x* faça:
   a) seja *i* a marca de *x;*
   b) não existe na tabela uma linha cujo campo *IDENT* é *i* :
      i) em uma linha vaga, marque o campo *IDENT* como *i* e o campo *CONT* como 1. Os campos *INIC* e *FIM* devem apontar para *Z* ;

c) existe na tabela uma linha cujo campo *IDENT* seja *i* :

i) incremente *CONT* de 1 ;

ii) seja *x* o nó apontado por *FIM*. O ponteiro direito de *x* e também de *FIM* devem apontar para *Z* ;

d) siga pelo ponteiro direito de *Z*. Chame de *Z* o nó encontrado.

3) para cada linha da tabela cujo campo de *CONT* é maior que 1 :

a) crie um novo nó rotulado com "//". Seja *y* este nó.

b) o ponteiro esquerdo de *y* recebe o valor de *INIC* ;

c) o ponteiro direito do nó apontado por *FIM* recebe um alinhavo apontando para *y* ;

d) *INIC* recebe um ponteiro para *y* .

4) seja *n* o número de linhas utilizadas na tabela auxiliar.

5) percorra, em sequência, as linhas da tabela, desde a linha 1 até a linha *n-1*, e faça o seguinte:

a) seja *i* a presente linha ;

b) o ponteiro direito do nó apontado pelo campo *INIC* da linha *i* deve receber o valor do campo *INIC* da linha *i+1* .

6) o ponteiro direito do nó apontado pelo campo *INIC* da linha *n* deve receber o alinhavo apontando para *R* .

7) o ponteiro esquerdo de *R* deve apontar para o nó indicado no campo *INIC* da linha 1

A execução da rotina GRUPARP aplicada à raiz da árvore de execução indicada à esquerda na Figura 7.17 deverá produzir a árvore indicada à direita na mesma figura. Deixamos a cargo do leitor convencer-se de que GRUPARP produz as transformações desejadas.

A rotina GRUPARS poderia ser bastante semelhante a GRUPARP. Como agora estamos forçados a observar a ordem sequencial especificada na árvore original, pode-se, entre outras melhorias, dispensar a tabela auxiliar e construir os grupos à medida que os nós são visitados. Por uma questão de completude, GRUPARS está detalhada abaixo. Aqui, *R* ainda é a raiz da árvore original e *X*, *Y*, *T* e *N* são variáveis auxiliares:

1) posicione *T* = *R* ;

2) desça pelo ponteiro esquerdo de *R*. Seja *X* o nó encontrado .

3) posicione *Y* = *X* ;

4) enquanto o ponteiro direito de *y* não for um alinhavo e a marca de *x* for idêntica a marca de *Y,* siga pelo ponteiro direito de *Y*. Chame de *Y* o nó encontrado.

5) se *X* for distinto de *Y*, então :

a) crie um novo nó, *N*, rotulado com ";" e cuja marca é idêntica a de *X* ;

b) o ponteiro esquerdo de *N* deve apontar para *X* ;

c) *X* é posicionado sobre o nó indicado pelo ponteiro direito de *Y* ;

d) o ponteiro direito de *Y* deve receber um alinhavo apontando para *N* ;

e) se *T* for idêntico a *R*, então o ponteiro esquerdo de *T* deve apontar para *N*, caso contrário, o ponteiro direito de *T* aponta para *N*.

f) se *X* for

i) idêntico a *R*, então *N* recebe um alinhavo para *R*; retorne;

ii) distinto de *R*, então:

(1) o ponteiro direito de *N* aponta para *X* ;

(2) posicione *T* em *N* e *Y* em *X* ;

(3) volte ao passo 4 .

6) se *X* for idêntico a *Y* então:

a) se o ponteiro direito de *X* indicar *R*, então retorne.

b) caso contrário:

i) posicione *T* sobre *X* ;

ii) siga pelo ponteiro direito de *X*. Chame de *X* o nó encontrado.

iii) posicione *Y* sobre *X* ;

iv) volte ao passo 4.

É também deixada ao leitor a tarefa de familiarizar-se com a lógica de GRUPARS. Em particular, a Figura 7.18 pode servir como teste em uma simulação manual das instruções deste procedimento.

Note que o nó de origem invoca o procedimento MARCAR apensa uma vez, quando do início da migração. Isto feito, aciona a rotina de ENVIAR_MENSAGENS a fim de despachar mensagens de trabalho para nós remotos. As mensagens contêm, em geral, a descrição de partes da árvore de execução. Ao receber mensagens de resposta, o nó de origem invoca o procedimento RECONFIGURAR para eliminar partes da árvore de execução. Note que RECONFIGURAR, ao eliminar a raiz de uma subárvore está, na realidade, retirando toda esta subárvore da árvore original. O próprio procedimento RECONFIGURAR invoca ENVIAR_MENSAGENS novamente, quando então novas mensagens são despachadas e o ciclo se repete. ENVIAR_MENSAGENS e RECONFIGURAR precisam ser levemente modificadas.

No caso de ENVIAR_MENSAGENS, agora não mais precisamos descer recursivamente a nível de uma operação básica, mas paramos quando é encontrado um nó intermediário cujo rótulo seja diferente de "*". Esta modificação é fácil de ser introduzida e o resultado é mostrado abaixo. Para evitar ambiguidades, chamaremos o procedimento modificado de N_ENVIAR_MENSAGENS. Supomos que o procedimento MARCAR já foi executado e todos os nós da árvore detêm uma marca, indicando o endereço remoto onde a subárvore com raiz neste nó deve executar, quando for o caso. Na descrição do procedimento, *P* é o programa *PR* a ser analisado, *A* é a raiz da árvore de execução de *P*, *O* é uma operação básica e *R* é uma variável auxiliar

1) *A* é rotulada *O*: despache *O* a seu nó remoto, junto com seu endereço relativo à árvore de execução, e retorne.

2) *A* é rotulada "//":

a) se a marca de *A* for "*", então:

i) siga pelo ponteiro esquerdo de *A.* Seja *R* a raiz da subárvore encontrada;

ii) invoque N_ENVIAR_MENSAGENS(*R*) recursivamente;

iii) se o ponteiro direito de *R* for um alinhavo, retorne;

iv) caso contrário, siga por este ponteiro direito. Chame de *R* o nó encontrado;

v) retorne ao passo 2a.ii;

b) caso contrário, envie *A* completa para o nó remoto cuja identidade é a marca de *A*.

3) *A* é rotulada ";":

a) se a marca de *A* for "*", então invoque N_ENVIAR_MENSAGENS(*R*) recursivamente, onde *R* é a subárvore cuja raiz é dada pelo ponteiro esquerdo de *A*; caso contrário, envie *A* completa para o nó remoto cuja identidade é a marca de *A* .

A modificação sobre RECONFIGURAR se deve ao fato de que precisa invocar N_ENVIAR_ MENSAGENS e não mais ENVIAR_MENSAGENS. Chamando de N_RECONFIGURAR a nova versão, teremos a rotina abaixo onde *A* é a árvore de execução e *END* é uma variável que aponta para o nó cujo endereço veio com a mensagem de resposta. O rótulo *O* indica uma operação básica enquanto *X, Y* e *Z* são variaveis auxiliares:

1) se *END* for a raiz da árvore, então N_RECONFIGURAR e a migração desta transação estão terminadas. Caso contrário, partindo de *END* siga pela direita até encontrar um nó com alinhavo. Seja *Z* o nó para onde o alinhavo aponta ;

2) *END* não é o descendente imediato esquerdo de *Z* :

a) a partir de *Z*, siga pela esquerda e depois sempre pela direita até encontrar um nó cujo ponteiro direito aponte para *END*. Seja *y* este nó;

b) elimine *END* da árvore. Basta atribuir ao ponteiro direito de *y* o valor do ponteiro direito de *END*, mesmo que este seja um alinhavo.

3) *END* é o descendente imediato esquerdo de *Z* :

a) O ponteiro direito de *END* não é um alinhavo:

i) caminhe uma vez pela direita a partir de *END*. Seja *x* o nó encontrado.

ii) elimine *END* da árvore atribuindo *x* ao valor do ponteiro esquerdo de *Z*;

iii) se o rótulo de *Z* for ";", execute N_ENVIAR_MENSAGEM(*x*)

b) O ponteiro direito de *END* é um alinhavo:

i) enquanto o ponteiro direito de *END* for um alinhavo e *END* for distinto da raiz da árvore; caminhe pelo alinhavo. Chame de *END* o nó encontrado.

ii) se *END* for a raiz da árvore, N_RECONFIGURAR e também a migração da transação estão terminadas.

iii) se *END* não for a raiz da árvore:

(1) siga sempre pela direita até encontrar um nó com alinhavo. Seja *Z* o nó para onde o alinhavo aponta;

(2) seja *x* o nó para onde o ponteiro direito de *END* aponta;

(3) elimine *END* da árvore. Basta atribuir *x* ao valor do ponteiro esquerdo de *Z* ;

(4) se *Z* for rotulado ";", execute N_ENVIAR_MENSAGENS(*x*) .

Observe que cada agente local instalado em nó remoto deve, agora, perfazer uma tarefa algo mais complexa do que simplesmente executar uma operação básica. Precisa executar uma árvore de execução em princípio tão complexa quanto aquela do nó de origem. A única diferença é que todas as operações básicas serão executadas localmente. Antes de consolidar todo o processo, porém, vamos examinar um outro ponto muito importante.

Em relação a situação simplificada inicial, resta um tipo de evento que merece atenção: são as falhas que podem se abater tanto sobre o nó original quanto sobre os nós remotos. Ha vários aspectos a serem mencionados. Em primeiro lugar, uma falha primária em qualquer nó remoto destroi o conteúdo da memória principal, levando consigo todos os agentes locais e respectivos descritores que se encontravam ativos quando do instante da falha. Em segundo lugar, e necessário que o nó de origem seja informado a respeito do fato que um nó remoto falhou. Para compreender este ponto, suponha que tal não fosse o caso. Neste cenário, o nó remoto falha e se recupera. A seguir, chega a próxima mensagem de trabalho endereçada a este nó remoto. Este último, simplesmente, responderia com a instalação de um agente local para operar em favor da transação, como faria se não tivesse falhado. Apenas que, como já perdeu os descritores e agentes locais relativos a esta transação, tudo se passaria no local remoto como se esta fosse a primeira mensagem de trabalho para esta transação. Tanto o nó de origem quanto este nó remoto prosseguiriam executando normalmente, embora parte do trabalho desta transação no local remoto já tivesse sido perdido quando da falha. Em terceiro lugar, note que também podem ocorrer falhas em cascata. Neste caso, o sistema volta a falhar no exato instante em que está a se recuperar de uma falha anterior. É preciso que os algoritmos levem este fato em conta e que, ao serem invocados, não provoquem inconsistências. Em quarto lugar, lembre que a cada nó, remoto ou não, é facultado cancelar uma transção, unilateralmente e a qualquer instante, desde que antes de iniciada a segunda fase do protocolo bifásico para confirmar intenções, como veremos na próxima subseção. Transações podem ser canceladas unilateralmente devido a um número de causas, entre elas a necessidade de quebra de bloqueios globais, ou mesmo locais. Finalmente, vale ressaltar que estaremos preocupados apenas com falhas primárias. Falhas secundárias e terciárias afetam a memória secundária do sistema e serão alvo de estudo no Capítulo 9.

A seguir vamos descrever a rotina completa que deve ser seguida pelo executor de transações em cada nó. Além de introduzir o tratamento contra falhas, o algoritmo não faz mais distinção entre nó remoto ou nó de origem, descrevendo o processo de execução de transações de maneira indistinta quanto a função original de cada nó.

EM OPERAÇÃO NORMAL: assincronamente,

1) os agentes locais estão operando em favor de várias transações, sob o controle do SGBD e do do executor de transações locais;
2) ao receber uma árvore de execução do processador de comandos da LMD presente neste nó, o executor de transações local:
    a) computa a lista de nós participantes;
    b) aciona os mecanismos de controle de integridade para por a lista de nós a salvo, junto com a identificação da transação;
    c) invoca o procedimento MARCAR sobre a árvore de execução;
    d) invoca o procedimento N_ENVIAR_MENSAGENS sobre a árvore de execução.
3) ao receber uma mensagem de resposta *m* referente a uma transação *T*, o executor de transações local investiga o campo de veredito de *M* :

a) se constar EFETUADO, invoca o procedimento N_RECONFIGURAR tendo como parâmetros a árvore de execução de *T* e o endereço de um nó interno da árvore, endereço este que veio codificado no campo de remetente de *M* ;

b) se constar CANCELADO e o protocolo bifásico ainda não foi invocado em favor de *T*

i) põe a salvo um registro indicando que o protocolo bifásico deve ser invocado, forçando o cancelamento de *T* ;

ii) inicia a execução do protocolo bifásico para confirmar intenções;

iii) após execução do protocolo bifásico, *T* está completa e pode desaparecer do BD ;

4) quando o executor de transações deteta que a fase de migração de uma transação *T* está completa:

a) coloca a salvo um registro indicando que iniciou a execução do protocolo bifásico com vistas a *T* ;

b) embarca na execução do protocolo bifásico;

c) após a execução do protocolo, *T* está terminada e desaparece do BD;

5) para toda mensagem de trabalho *M* recebida por este nó, o executor de transações local :

a) verifica se esta transação já consta como ativa neste nó consultando sua tabela de transações ativas. Em caso negativo, registra em lugar seguro o fato de que agora esta transação já está ativa neste nó, adicionando também sua identidade à tabela de transações ativas;

b) inicia e instala um agente local para servir a esta requisição;

c) extrai a árvore de execução do campo de descrição de *M* ;

d) invoca o procedimento EXTRAIOP sobre esta árvore;

6) cada vez que o executor de transações identifica que as tarefas requisitadas em uma mensagem de trabalho estão completas:

a) constrói uma mensagem de resposta, preenchendo o campo remetente com a identidade deste nó remoto mais o endereço, relativo a árvore de execução original, da raiz da subárvore que recebeu para executar. O campo identificação é preenchido com a identificação da transação e, finalmente, o campo de veredito com EFETUADO;

b) envia a mensagem de resposta ao nó de origem;

c) extingue o agente local correspondente;

7) se algum agente externo válido decide pelo cancelamento de uma transação *T* originária neste nó, então:

a) registra em lugar seguro que o protocolo bifásico será invocado em favor de *T* ;
b) aciona o executor de transações para que este invoque o protocolo bifásico com intenção de forçar o cancelamento da transação;

8) se algum agente externo válido decide pelo cancelamento de uma transação *T/* que está executando remotamente nó, então:

a) constrói uma mensagem de resposta com campo de remetente preenchido com a identidade deste nó, campo de identifição refletindo a identidade da transação e campo de veredito CANCELADO;
b) envia a mensagem ao nó de origem.

EM CASO DE FALHA: ao se recuperar de uma falha, o mecanismo de controle de integridade é acionado e assume o controle. Os registros de que dispõe são recuperados. Para cada registro encontrado, indicando que trabalho foi iniciado em favor de uma transação *T* se:

1) encontra um outro registro indicando que o protocolo bifásico para esta transação já foi iniciado, então o controle de integridade repassa esta informação ao executor de transações para que este último complete a estratégia prescrita pelo protocolo bifásico;
2) se o protocolo bifásico em favor de *T* ainda não foi iniciado e *T* estava executando remotamente neste nó, então:
   a) constrói uma mensagem de resposta com campo de remetente preenchido com a identidade deste nó, campo de identifição refletindo a identidade da transação e campo de veredito CANCELADO;
   b) envia a mensagem ao nó de origem;
3) se o protocolo bifásico em favor de *T* ainda não foi iniciado e *T* tinha neste nó seu nó de origem, então:
   a) registra em lugar seguro que o protocolo bifásico será invocado em favor de *T*;
   b) aciona o executor de transações para que este invoque o protocolo bifásico com intenção de forçar o cancelamento da transação.

O procedimento EXTRAIOP é uma versão simplificada do procedimento ENVIAR_MENSAGENS. Note que, agora, sabemos que as operações básicas recebidas através de mensagens serão todas executadas no presente nó remoto. Se *A* representar a raiz da árvore de execução e *O* for uma das operações básicas, EXTRAIOP procede como indicado abaixo.

1) *A* é rotulada *O*: instale e inicie um agente local para esta transação. Repasse *O* para este agente e retorne;
2) *A* é rotulada "//":
   a) siga pelo ponteiro esquerdo de *A*. Seja *R* a raiz da subárvore encontrada.
   b) invoque EXTRAIOP(*R*) recursivamente.
   c) se o ponteiro direito de *R* for um alinhavo, retorne;
   d) caso contrário, atribua a *R* a raiz da subárvore indicada por este ponteiro;
   e) retorne ao passo 2b.
3) *A* é rotulada ";":
   a) siga pelo ponteiro esquerdo de *A*. Seja *R* a raiz da subárvore encontrada;
   b) invoque EXTRAIOP(*R*) recursivamente;
   c) retorne.

O resto desta subseção será dedicado a uma discussão do algoritmo que acabamos de apresentar. Em primeiro lugar, vamos examinar o processo de execução de transações em operação normal. É uma consolidação das observações que fizemos até agora a respeito do início, migração e término de transações tanto para nós remotos quanto para nós de origem de novas transações. A única rotina ainda não especificada por completo é o protocolo bifásico

para confirmação de intenções. Este último será detalhado na seção seguinte. Os ítens listados tratam dos vários tipos de eventos que podem ocorrer, assincronamente, em cada nó.

Na ausência de novas transações, se mensagens de trabalho não chegam e ainda se as tarefas locais não se completaram, dispensando mensagens de resposta, temos apenas os agentes locais a executarem suas funções. É o cenário do passo 1. Parte do trabalho de cada agente local, a nível de ação elementar, deve ser resguardado de danos causados por eventuais falhas. Lembre que cada nó deve ser capaz de desfazer/refazer os efeitos das ações elementares de cada transação que execute localmente. Como isto pode ser conseguido será descrito no Capítulo 9. Os mecanismos de controle de concorrência, a serem estudados no Capítulo 8, provêem a ilusão de que só esta transação está a executar no BD, neste instante. Podemos, portanto, raciocinar sob esta hipótese.

Os três próximos ítens, 2, 3 e 4, descrevem o comportamento deste nó quando uma nova transação é submetida localmente. Os ítens 5 e 6 especificam o comportamento deste nó quando solicitado por outro a operar remotamente em favor deste último. O procedimento adotado quando um agente externo decide cancelar a transação é descrito nos passos 7 e 8. Agentes externos válidos podem ser, por exemplo, o próprio usuário ou o módulo responsável por levantar bloqueios.

O passo 2 especifica que a lista de nós participantes deve ser protegida. Em seguida, o procedimento MARCAR é acionado para determinar as subárvores máximas que poderão ser despachadas a cada nó, conseguindo o máximo de paralelismo possível. Esta rotina chama, internamente, as rotinas GRUPARP e GRUPARS que modificam localmente a estrutura da árvore de execução. Isto permitirá que o próximo módulo a ser invocado pelo passo 2, N_ENVIAR_MENSAGENS, despache as subárvores a seus respectivos destinatários remotos. Note que uma das subárvores selecionadas pode ser destinada a executar localmente, no próprio nó de origem. Após enviadas as mensagens de trabalho, o cenário do passo 1 volta a se apresentar: os agentes locais, em cada nó ativado, se desincumbem de suas respectivas tarefas. Quando mensagens de resposta começam a chegar, o código do passo 3 entra em ação. Como estamos supondo que falhas não ocorrem, apenas o caso "a" será ativado. O campo de veredito de cada mensagem de resposta conterá EFETUADO. A seguir, o procedimento N_RECONFIGURAR é invocado. Este último, então, elimina da árvore de execução toda a subárvore que estava a executar remotamente e cuja mensagem de resposta, indicando sucesso na execução, estamos a analisar. Finalmente, a própria rotina N_RECONFIGURAR já aciona novamente a rotina N_ENVIAR_MENSAGENS que se encarrega de despachar novas subárvores para execução remota.

Sob requisição de trabalho vinda de outro local, cada nó se comporta conforme especificado nos passos 5 e 6. No primeiro, são descritas as ações correspondentes a recepção de mensagens de trabalho. O executor de transações local instala um agente local e inicializa seu descritor. Passa, a seguir, a analisar as especificações da árvore que veio com a mensagem de trabalho. Aqui, há duas estratégias básicas a seguir, a escolha dependendo de fatores locais de cada nó. No procedimento acima, optou-se por invocar a rotina EXTRAIOP sobre esta árvore. Note que, assim procedendo, poderemos estar criando vários agentes locais para operarem em benefício desta transação, caso EXTRAIOP detete a chance de executar mais de uma operação básica em paralelo. O custo envolvido com a criação e coordenação destes processos locais deve ser ponderado contra a opção de instalarmos apenas um agente local para esta solicitação. Neste último caso, as operações básicas prescritas na árvore de execução recebida seriam executadas uma por vez, mesmo que houvesse oportunidade para explorarmos paralelismos inerentes a natureza do problema. Finalmente, chegamos a situação

onde o executor de transações deteta que o trabalho solicitado por uma mensagem anterior já está completo. Devemos executar o passo 6. Uma mensagem de resposta é construida e encaminhada ao nó que solicitou o trabalho. O campo de veredito desta mensagem deve conter EFETUADO. Claro, se o presente nó é, também, o nó de origem da transação, então a mensagem pode ser dispensada. Sob estas condições, o executor de transações deve, agora, executar os passos previstos no ítem 3.

Para terminar esta subseção, vamos discutir os pontos mais relevantes no combate a falhas. É responsabilidade do controle de integridade invocar ações especificas quando o sistema está a se recuperar de falhas. Embora deixemos os detalhes para o Capítulo 9, cabe, neste ponto, uma visão simplificada das operações que serão executadas. Ao retornar de uma falha cada nó tem condições de:

1) recuperar a identidade de cada transação que migrou para este nó e ainda não foi confirmada ou cancelada;
2) recuperar a identidade e a lista de nós participantes de cada transação que originou neste nó e ainda não terminou sua execução;
3) refazer/desfazer cada ação elementar de toda transação que foi ativada sobre este nó;
4) recuperar o texto de mensagens que ainda não processou e para as quais já enviou as correspondentes confirmações de recebimento.

Os passos 2 e 5 garantem os dois primeiros ítens. Note que o protocolo de comunicações só deve liberar a confirmação de recebimento da mensagem (o "ack") após o nó recipiente ter salvo a própria mensagem. Do contrário, a confirmação poderia ser enviada e, antes da mensagem ser analisada, o sistema local pode cair, pondo a perder o texto da mensagem. A confirmação funciona como um recibo, assinado pelo recipiente, de que a mensagem chegou e está a salvo. O terceiro ítem é garantido pelo modo de operar de cada agente local, os quais tomam cuidados especiais para preservar a capacidade de desfazer/refazer ações elementares. Isto é feito usando os mecanismos de controle de integridade, conforme detalhado no próximo capítulo.

Tome agora um nó que está se recuperando de uma falha. Abstraindo detalhes, o processo de recuperação envolveria, a grosso modo e além de outras medidas, as operações indicadas anteriormente. Cada transação que executou remotamente neste nó terá seu nó de origem avisado deste fato, pois receberá uma mensagem de resposta cujo campo de descrição contém CANCELADO. A identidade do nó de origem pode ser obtida analisando-se a própria sequência de caracteres que formam a identificação da transação. A execeção feita àquelas transações que já tenham embarcado na execução do protocolo bifásico se justifica na medida em que este protocolo deve operar de forma robusta com relação a possíveis falhas. Desta forma, uma vez iniciado, o protocolo ira até o fim, garantindo a confirmação ou cancelamento da transação, conforme o caso. O efeito das mensagens de CANCELADO junto a cada nó de origem será, primeiro, registrar em lugar seguro que esta transação esta sendo cancelada. Em seguida, o nó de origem inicia a execução do protocolo bifásico para cancelar a transação globalmente. Isto é previsto no passo 3. Ao se verificar uma falha remota, portanto, estamos elegendo por cancelar a transação globalmente. Note que, se um nó remoto voltar a cair durante esta fase de recuperação, tornará a enviar mensagens de resposta com veredito de CANCELADO ao nó de origem, causando com que múltiplas mensagens referentes a "mesma" falha sejam recebidas por este último. Conforme estipulado no passo 3, porém, antes de iniciar a execução do protocolo bifásico com o intuito de cancelar esta transação, um registro deste fato é posto a salvo no local de origem. Quando o nó de origem

receber múltiplas mensagens causadas por este tipo de falhas em cascata, o passo 3b detetará que o protocolo bifásico já iníciou, evitando sua reexecução. Observe que este esquema também funciona se mensagens múltiplas forem causadas por falhas em vários nós remotos que participaram da execução de uma mesma transação. Para transações originárias no local da falha, o passo 3 estipula a imediata execução do protocolo bifásico com o intuito de cancelar a transação. Novamente, antes de iniciar a execução do protocolo, um registro é salvo indicando este fato. Se o nó volta a falhar durante este período de recuperação, apenas as transações ali originárias para as quais o protocolo ainda não tinha sido ativado é que passarão por este processo. A robustez do próprio protocolo bifásico garante os resultados deste ponto em diante.

7.4 TÉRMINO DE TRANSAÇÕES

Esta última seção discute o protocolo bifásico para confirmar intenções. A importância do protocolo reside no fato de que se coloca como o último módulo que é invocado durante a vida de uma transação. É responsável por garantir a propriedade de *atomicidade* inerente ao conceito de transação. Não fosse a eventualidade de falhas que podem vir a perturbar a operação normal do sistema, obter o consenso de todos os nós participantes quanto a cancelar ou confirmar determinada transação seria tarefa muito simples. A imprevisibilidade das falhas, aliada ao efeito danoso de destruirem o conteúdo da memória principal, torna o problema de se chegar a uma decisão consensual bem mais complexo.

Nesta seção estamos assumindo que os mecanismos de controle de integridade funcionem a toda prova, protegendo todo registro que lhe seja confiado contra quaisquer tipos de falhas. Mais ainda, estes registros poderão ser recuperados intactos quando o sistema que falhou estiver no processo de recuperação para voltar à operação normal. Não importando o tipo ou a severidade da catástrofe, suporemos que sempre poderemos recuperar qualquer informação vital que se fizer necessária. Para isto, é necessário que este tipo de informação seja *reconhecida* pelos algorítmos que descreveremos e, também, que seja expressamente passada aos mecanismos de controle de integridade *antes* do instante crucial quando assumimos que sobrevivam a falhas. Só desta maneira poderemos estar seguros de que estes registros estarão efetivamente protegidos. Note-se que, muitas vezes, a informação a resguardar está localizada em um *nó remoto*, forçando troca de mensagens para que o nó em questão tenha como certo que o nó remoto já tomou as devidas providências. Este fato é um dos mais sérios complicadores que se apresentam quando tentamos desenvolver um protocolo bifásico, visto que dá margem a que os múltiplos nós envolvidos falhem, talvez várias vezes, permitindo que situações complexas se desenvolvam. Seguindo este racional, o princípio básico que norteia as ações preventivas que serão tomadas para garantir proteção contra falhas pode ser resumido no seguinte paradigma.

Ao nos depararmos com informação vital

1. tentamos salvar tudo que for necessário, repassando a informação aos mecanismos de controle de integridade;
2. só após o sucesso desta operação, podemos continuar do ponto onde havíamos parado anteriormente.

Como veremos adiante, uma das inconveniências que pode resultar deste tipo de procedimento é que os vários nós envolvidos no processo estarão sujeitos a receber múltiplas mensagens descrevendo o "mesmo" evento. Tipicamente, o remetente falhou e, ao se recuperar da falha, torna a falhar pela segunda vez. Ao retornar da segunda falha, refaz uma

série de ações, entre as quais o envio de mensagens que já tinham sido despachadas quando se recuperava da primeira falha. É o próprio processo de recuperação que leva um nó a repetir mensagens, e não um mau funcionamento da rede de comunicação de dados. Tendo em mente o paradigma posto acima, podemos recolocar o esquema básico para término de transações, descrito no Capítulo 6, como abaixo. Nesta descrição, e em todo o resto desta seção, assumiremos que o nó de origem da transação é estipulado como o *coordenador* da operação. A razão é simples: o nó de origem é o único que dispõe da lista de todos os nós participantes. Mais ainda, esta lista está a salvo neste nó.

NUMA PRIMEIRA FASE

1) coordenador:
    a) registra em lugar seguro que iniciará o envio de mensagens exploratórias;
    b) envia, a todos os nós participantes, mensagens de consulta na tentativa de saber se todos estão aptos a executar o protocolo;
    c) espera pelas mensagens de resposta dos nós remotos;
2) cada nó participante:
    a) ao receber a mensagem tenta registrar os efeitos locais da transação em lugar seguro;
    b) põe a salvo um registro indicando o sucesso ou insucesso da operação;
    c) se foi bem sucedido, responde ao coordenador que está preparado para prosseguir. Caso contrário, responde que não poderá continuar;
    d) espera pelo veredito final do coordenador.

NUMA SEGUNDA FASE

1) coordenador:
    a) baseado nas respostas encaminhadas pelos nós participantes, determina o veredito final com relação ao destino da transação;
    b) registra o veredito final em lugar seguro;
    c) envia a todos os nós participantes uma mensagem descrevendo seu veredito final;
    d) espera por mensagens, uma de cada nó participante, confirmando que tomou as medidas estipuladas de acordo com o veredito final;
    e) após a chegada de todas as mensagens confirmatórias, o executor de transações local termina o processamento em favor desta transação, a qual desaparece do BD;
2) cada nó participante:
    a) ao receber o veredito do coordenador, age de acordo com este;
    b) coloca a salvo um registro, indicando o destino desta transação;
    c) envia ao coordenador uma mensagem de resposta, confirmando que agiu de acordo com o veredito que recebeu;
    d) cancela os agentes locais que operavam em benefício desta transação.

Repare como é feito um esforço para observar o paradigma, exposto anteriormente, que força o registro da intenção antes de executar qualquer ação correlata.

Vamos agora embarcar na descrição do protocolo bifásico em detalhe. Antes, porém,

lembraremos os pontos principais que serão importantes para a compreensão do mesmo. Com relação aos protocolos responsáveis pela comunicação entre os nós da rede, relembrando, suporemos que são confiáveis ao ponto de podermos assumir que toda mensagem despachada:

- chega a seu destino correto em tempo finito;
- não é entregue em endereço errado;
- não é deturpada ao ser transmitida;
- não tem sua ordem trocada, isto é, mensagens partindo de um mesmo emitente para um mesmo recipiente chegarão na mesma ordem que foram enviadas;
- não é duplicada.

Outro aspecto importante com relação aos protocolos de comunicação de dados diz respeito ao envio de confirmações ("acks") pelo nó recipiente. A confirmação só é enviada após a mensagem estar a salvo, em lugar seguro, no destinatário. Suponha que uma mensagem *M* é enviada do nó *x* para o nó *y*. Se, quando *M* chegar, *y* não está em operação normal e o conteúdo de *M* é perdido, então o protocolo de comunicação se encarregará de reenviar *M*, visto que *x* não recebe qualquer confirmação em tempo hábil com relação a *M*. O mesmo se passa se, quando *M* chega a *y*, este está operando normalmente mas falha antes de ter conseguido salvar *M* em lugar seguro. Ainda neste caso, *x* não recebe a confirmação e o protocolo de comunicação de dados tomará a iniciativa de reenviar *M* a *y*. Este esquema garante que, se *y* falhar, toda mensagem *M* para a qual já tenha enviado a confirmação de recebimento estará a sua disposição para análise quando retornar a operação normal, ou mesmo durante o processo de recuperação, só desaparecendo quando *y* a lê e descarta. Na realidade, assim procedendo, estamos evitando incorporar ao protocolo bifásico os mecanismos de contagem de tempo (relógios e "timeouts" associados), necessários para garantir os efeitos estipulados. Estamos relegando esta tarefa ao protocolo de comunicação, visto que este já implementa estes mecanismos como parte de sua operação normal, pois deles necessita para viabilizar outras tarefas de sua responsabilidade.

No que tange ao cenário original, os processos de iniciação e migração de transações, descritos nas seções anteriores, garantem os seguintes fatos quando do instante em que o protocolo bifásico inicia sua execução:

- um registro foi salvo indicando a intenção de iniciar a execução do protocolo bifásico;
- o coordenador salvou a lista de nós participantes, associada à identificação da transação;
- cada nó participante, inclusive o nó de origem, registrou os efeitos locais de cada transação de forma tal que tem capacidade para refazer/desfazer quaisquer destes efeitos, sempre que julgar necessário

Outro aspecto da questão diz respeito à liberdade delegada a cada nó, individualmente, de cancelar, unilateralmente, qualquer transação. Até o momento, este direito poderia ser exercido a qualquer instante, desde que *antes* do início do protocolo bifásico. Ocorre, porém, que há o perigo de bloqueios globais no sistema. Em tais casos, o módulo que deteta bloqueios deve selecionar um certo número de transações que seriam canceladas e posteriormente automaticamente resubmetidas pelo sistema, número este suficiente para levantar todos os bloqueios existentes naquele instante. Se insistirmos em que transações não podem ser canceladas posteriormente ao início do protocolo bifásico, poderíamos,

potencialmente, atingir um cenário onde teríamos bloqueios globais envolvendo apenas transações nestas condições. Isto geraria um impasse. Felizmente, a situação pode ser contornada exigindo-se que transações não sejam canceladas após terem registrado a intenção de entrar na *segunda fase* do protocolo bifásico. Esta alteração não afeta fundamentalmente nenhum dos ítens discutidos nas seções anteriores, pois não se envolviam com detalhes de implementação do protocolo bifásico. A razão de passarmos a esta nova condição de irreversibilidade quanto ao cancelamento unilateral de transações ficara mais clara no Capítulo 8 que trata do problema de controle de concorrência. Basicamente, após passar pela primeira fase do protocolo bifásico, a transação já requisitou todos os recursos de que necessitará. Assim, não mais solicitará recursos e, portanto, não poderemos ter bloqueios constituídos apenas por transações nestas condições.

Finalmente, é importante relembrar que o protocolo deve apresentar robustez contra a eventualidade de falhas múltiplas. Ou seja, as propriedades básicas inerentes ao conceito de transação devem prevalecer mesmo que nós falhem quando estão a se recuperar de falhas anteriores, não importando quantas vezes esta situação seja cascateada em um mesmo nó ou em múltiplos nós. Está implícita, porém, a hipótese de que, embora qualquer nó possa ser prejudicado por falhas, este nó retornará a operação normal em algum ponto no futuro. Do contrário, o protocolo poderia entrar em compasso de espera sem fim, o que negaria a atomicidade da transação, além de bloquear recursos do sistema eternamente. Note também que o protocolo deverá incorporar código para tratamento de falhas e é executado pelo executor de transações local.

Após esta discussão preparatória, estamos prontos a iniciar a descrição do protocolo em datalhe. Faremos uso do campo de estado de retorno que faz parte do descritor de cada transação, conforme discutido em seção anterior. Para facilitar a leitura do que segue, relembramos que o campo de descrição das mensagens de trabalho e o campo de veredito das mensagens de resposta podem conter as seguintes informações, pertinentes no momento.

O campo de descrição da mensagem de trabalho pode conter:

- descrição da árvore de execução
- PREPARE_SE
- CONFIRME
- CANCELE

O campo de veredito da mensagem de resposta pode conter

- EFETUADO
- CANCELADO
- PREPARADO
- IMPOSSIBILITADO
- COMPLETADO

O campo de estado de retorno do descritor pode conter

- DESCONHECIDO
- ATIVO
- COLETANDO
- CONFIRMANDO
- CANCELANDO
- PRONTO_SIM
- PRONTO_NAO

O protocolo bifásico é obtido detalhando-se o esquema básico apresentado há pouco. Estamos supondo que qualquer nó que ative o protocolo, na qualidade de coordenador, deve registrar, em lugar seguro e *antes de invocar o protocolo*, que a transação em questão está passando do estado de DESCONHECIDO para COLETANDO. Esta mudança também é efetivada no campo de estado de retorno do descritor da transação. Na descrição que segue, optamos por separar as ações de acordo com a natureza de cada nó ao invés de agrupá-las por fases. Em cada caso, a atitude de cada nó é guiada pelo estado de retorno em que se encontra a transação. Ao entrar em determinado estado de retorno, o sistema executa, em primeiro lugar, os passos relacionados com a computação local. Em seguida, fica à espera de mensagens. Estas podem chegar de maneira assíncrona e em qualquer ordem. Pode ser o caso, inclusive, de já haver mensagens pendentes após executarmos os passos relativos à computação local. Em qualquer circunstância, após executar a computação local, o sistema colhe mensagens e, em cada caso, segue os passos correspondentes, segundo previsto no código do estado de retorno em que se encontra. Também foram incluídos os passos que devem ser observados se o sistema é colhido por uma falha quando já havia invocado o protocolo em favor de uma certa transação. Neste caso, o controle de integridade, após detetar que o protocolo já estava sendo executado, delega ao executor de transações local a tarefa de completar sua execução. Este último verifica o estado de retorno em que se encontrava a transação imediatamente antes da falha, agindo como se estivesse a mudar, neste instante, para este mesmo estado de retorno. Isto significa que efetuará a computação local antes de se voltar para as mensagens que porventura tenham sobrevivido à falha. Segue a especificação do protocolo bifásico.

O COORDENADOR, NO ESTADO DE

1) COLETANDO:
   a) efetua computação local:
      i) obtém a lista de nós participantes;
      ii) envia, a cada nó participante, uma mensagem de trabalho onde o campo de descrição contém PREPARE_SE;
      iii) inicia uma tabela com cada um dos nós participantes marcados como "sem_resposta";
   b) processa mensagens de resposta recebidas. Se o campo de veredito for
      i) PREPARADO:
         (1) marca o nó remetente como "com_resposta";

(2) verifica se todo nó da tabela já está marcado como "com_resposta". Em caso afirmativo, registra, em lugar seguro e no campo de estado de retorno do descritor da transação, que está passando ao estado de CONFIRMANDO;

ii) IMPOSSIBILITADO ou CANCELADO: registra, em lugar seguro e no campo de estado de retorno do descritor da transação, que está passando ao estado de CANCELANDO;

2) CONFIRMANDO:

a) efetua computação local:

i) obtém a lista de nós participantes;

ii) envia, a cada nó participante, uma mensagem de trabalho onde o campo de descrição contém CONFIRME;

iii) inicia uma tabela onde aparece cada um dos nós participantes marcados como "sem_confirmação";

b) processa mensagens de resposta recebidas. Se o campo de veredito for:

i) COMPLETADO:

(1) marca o nó remetente como "com_confirmação";

(2) verifica se todo nó da tabela já está marcado como "com_confirmação". Em caso afirmativo, termina o protocolo neste nó de origem. A transação sai do BD.

ii) PREPARADO: envia ao nó perticipante uma mensagem de trabalho com campo de descrição contendo CONFIRME

3) CANCELANDO:

a) efetua computação local:

i) obtém a lista de nós participantes;

ii) envia, a cada nó participante, uma mensagem de trabalho onde o campo de descrição contém CANCELE;

iii) inicia uma tabela onde aparece cada um dos nós participantes marcados como "sem_confirmação";

b) processa mensagens de resposta recebidas. Se o campo de veredito for

i) COMPLETADO:

(1) marca o nó remetente como "com_confirmação";

(2) verifica se todo nó da tabela já está marcado como "com_confirmação". Em caso afirmativo, termina o protocolo neste nó de origem. A transação sai do BD.

ii) PREPARADO ou IMPOSSIBLILITADO: envia ao nó participante uma mensagem de trabalho com campo de descrição contendo CANCELE;

CADA PARTICIPANTE, NO ESTADO DE

1) DESCONHECIDO: ao receber a mensagem de trabalho com campo de descrição contendo PREPARE_SE, passa ao estado de "ATIVO";

2) ATIVO: efetua computação local:

a) verifica se tem condições locais de confirmar a transação. Isto implica em:

   i) resguardar em lugar seguro todos os efeitos locais da transação, garantindo que poderão ser refeitos mais tarde, caso necessário;

   ii) obter todos os recursos do sistema de que vai necessitar para completar a transação;

   iii) verificar se esta transação não foi cancelada localmente;

b) se está em posição de confirmar a transação localmente, registra em lugar seguro e no campo de estado de retorno do descritor que está a ingressar no estado de PRONTO_SIM;

c) se não está em posição de confirmar a transação localmente, registra em lugar seguro e no campo de estado de retorno do descritor que está a ingressar no estado de PRONTO_NAO;

3) PRONTO_SIM:

a) efetua computação local: envia ao coordenador uma mensagem de resposta com campo de veredito contendo PREPARADO;

b) processa mensagens de trabalho recebidas. Se o campo de descrição contiver:

   i) CONFIRME:

      (1) coloca a salvo um registro indicando que a transação foi confirmada;

      (2) envia uma mensagem de resposta ao nó coordenador cujo campo de veredito contém COMPLETADO;

      (3) termina o protocolo neste nó remoto. A transação completou sua execução local;

   ii) CANCELE:

      (1) remove os efeitos locais da transação;

      (2) coloca a salvo um registro indicando que a transação foi cancelada;

      (3) envia uma mensagem de resposta ao nó coordenador cujo campo de veredito contém COMPLETADO;

      (4) termina o protocolo neste nó remoto. A transação completou sua execução local;

   iii) PREPARE_SE: envia ao coordenador uma mensagem de resposta com campo de veredito contendo PREPARADO;

4) PRONTO_NAO:

a) efetua computação local: envia ao coordenador uma mensagem de resposta com campo de veredito contendo IMPOSSIBILITADO;

b) processa mensagens de trabalho recebidas. Se o campo de descrição contiver:

   i) PREPARE_SE: envia ao coordenador uma mensagem de resposta com campo de veredito contendo IMPOSSIBILITADO;

   ii) CANCELE:

      (1) remove os efeitos locais da transação:

      (2) coloca a salvo um registro indicando que a transação foi cancelada;

      (3) envia uma mensagem de resposta ao nó coordenador cujo campo de veredito

contém COMPLETADO;

(4) termina o protocolo neste nó remoto. A transação completou sua execução local;

5) se a transação não mais consta da tabela de transações ativas neste nó, tabela esta em poder do executor de transações: processa as mensagens de trabalho recebidas de acordo com seu campo de descrição:

   a) CONFIRME: envia uma mensagem de trabalho ao coordenador cujo campo de veredito contém COMPLETADO;

   b) CANCELE: envia uma mensagem de trabalho ao coordenador cujo campo de veredito contém COMPLETADO;

   c) CASO CONTRÁRIO: ignora a mensagem

EM CASO DE FALHA

1) os mecanismos de controle de integridade recuperam o sistema local. Se estes descobrem que o último registro de estado de retorno referente a uma transação *T* é:

   a) DESCONHECIDO: os mecanismos de controle de integridade tomam as medidas apropriadas em relação a *T* ;

   b) *R*, diferente de DESCONHECIDO: os mecanismos de controle de integridade repassam ao executor de transações a tarefa de concluir a execução do protocolo bifásico em favor de *T*. Após a volta a normalidade, o executor de transações reexecuta as tarefas previstas no estado *R* como se estivesse entrando neste estado pela primeira vez.

A descrição acima distingue entre os comportamentos afetos a um nó coordenador e a um nó participante. Na realidade, o algorítmo é executado de forma integrada, pois um mesmo nó pode abrigar tanto transações aí originárias como transações que para aí migraram. A distinção é feita a nível de mensagens. Assim, um nó nunca receberá uma mensagem de trabalho identificada com uma transação que tem aí seu nó de origem e cujo campo de descrição contém PREPARE_SE. Embora não explicitamente declarado, estamos assumindo que um nó coordenador também está sujeito a executar localmente parte das ações referentes a uma transação. Desta forma, também se obrigará a tentar registrar em lugar seguro todos os efeitos locais desta transação antes de tomar a decisão quanto ao veredito final. Seu voto, obviamente, também deve ser levado em conta no passo 1b do grupo de ações que executa como coordenador.

Assumindo primeiro um cenário onde falhas não ocorressem, apenas os grupos do protocolo referentes a um nó coordenador e a um nó participante seriam ativados. Neste caso o coordenador nada mais faz que consultar cada participante quanto as suas condições locais com vistas a confirmar a transação. Este comportamento está descrito no passo 1 da parte que cabe ao coordenador no algorítmo acima. Cada nó participante remete ao coordenador seu voto de acordo com suas condições locais. São os passos 3a e 4a da parte que diz respeito a cada nó participante. Se todos votam a favor, o coordenador decide pala confirmação e comunica este fato a cada participante. Se alguém votou contra, o coordenador será forçado a decidir pelo cancelamento, o que é comunicado em seguida a todos os outros nós envolvidos. É o passo 3 do código relativo ao coordenador. Finalmente, cada nó participante age de acordo com o veredito recebido do coordenador, segundo os passos 3 e 4 do grupo de ações

que cabe a cada participante. Em qualquer caso, uma mensagem de resposta, com campo de veredito contendo COMPLETADO, é encaminhada por cada nó participante ao coordenador. Após certificar-se de que todos os nós participantes liqüidaram a transação, o nó coordenador faz o mesmo. Neste cenário simplificado onde não há falhas, o protocolo poderia ser otimizado dispensando-se o registro de cada estado em lugares de onde pudessem ser recuperados em caso de falha. No entanto, a consulta a cada nó ainda é necessária, mesmo na ausência de falhas. Para compreender isto, suponha que a transação migrou sem problemas. O executor de transações no local de origem inicia a execução do protocolo bifásico enviando PREPARE_SE a cada participante. Em paralelo com o envio das mensagens, porém antes que uma delas chegue a determinado nó remoto, a transação é cancelada neste nó remoto. Lembre que o sistema pode forçar isto a ocorrer para evitar bloqueios. Nesta situação, não teríamos outra alternativa que cancelar a transação em todos os demais nós onde executou. A consulta iniciada pelo coordenador é, portanto, necessária. É exatamente no instante quando entra no estado de PRONTO_SIM ou PRONTO_NAO que não é mais permitido a um nó cancelar, unilateralmente, a correspondente transação. Focalizando sobre outro ângulo, pode-se dizer que a consulta efetivada pelo coordenador funciona como um "sinal" que percorre todos os nós remotos a alertá-los que o período de cancelamentos individuais está encerrado, pelo menos no que tange à presente transação.

Passando a admitir falhas, vamos encarar o que ocorre quando um nó participante é acometido por um destes eventos. Dependendo de quão adiantado esteja a execução do protocolo neste nó, podemos ter vários cursos de ação.

Tome, em primeiro lugar, o caso no qual o nó remoto, ao retornar, não se depare com qualquer registro de que o protocolo bifásico já foi iniciado em benefício desta transação. Isto é possível se o nó remoto falha antes que tenha chegado a mensagem de trabalho com campo de descrição PREPARE_SE que lhe é endereçada pelo coordenador. Imediatamente, o nó remoto envia ao nó de origem da transação uma mensagem de resposta com campo de veredito contendo CANCELADO. Isto era o previsto quando discutimos a migração de transações na presença de falhas. Note que, no que concerne ao sistema local, esta transação ainda *não* foi envolvida pelo protocolo bifásico, visto que seu estado de retorno ainda é DESCONHECIDO. De acordo com o passo 1b do protocolo especificado acima, o coordenador passa ao estado de CANCELANDO e, em seguida, distribui mensagens a todos os participantes desta transação para que removam seus efeitos locais. O campo de descrição destas mensagens contém CANCELE. Lembre-se que estamos supondo que os protocolos de comunicação são robustos com relação a falhas. As confirmações (ou "acks") necessários só são enviados quando as correspondentes mensagens foram "lidas", isto é, resguardadas de falhas. Por conseguinte, podemos tomar como garantido que a mensagem de PREPARE_SE eventualmente será analisada pelo nó remoto. Mais ainda, a mensagem de CANCELE também já pode ter chegado quando este nó remoto começar a processar suas mensagens após se recuperar da falha. Ou então, antes de enviar a mensagem de CANCELE, o coordenador já falhou e se recuperou várias vezes. Neste último caso, várias mensagens de PREPARE_SE podem estar acumuladas junto ao nó remoto. De modo similar, o coordenador pode ter falhado várias vezes após ter entrado no estado de CANCELANDO, esta última mudança de estado tendo sido provocada pela primeira mensagem de CANCELADO que recebeu do nó remoto. Se isto ocorrer, poderemos ter, adicionalmente, várias mensagens de CANCELE junto com as mensagens de PREPARE_SE esperando para serem processadas pelo nó remoto.

O que ocorre após este nó remoto voltar à operação normal depende da ordem com que estas mensagens são analisadas. O resultado final, porém, será sempre o mesmo: o cancelamento

da transação neste nó remoto. Se a mensagem com campo de descrição PREPARE_SE é analisada primeiro, o nó passa ao estado de ATIVO. Lembre-se que o controle de integridade não modifica o estado de retorno da transação e, antes da falha, este continha DESCONHECIDO. O passo 1 do protocolo bifásico, portanto, dita que devemos efetuar a troca de estados conforme indicado. No estado de ATIVO, descobrimos imediatamente que a transação foi cancelada localmente. Isto força a passagem ao estado de retorno de PRONTO_NAO. Neste estado, uma mensagem de IMPOSSIBILITADO é enviada ao coordenador. Se, neste ponto, o nó remoto processa algumas das mensagens de PREPARE_SE, caso haja tais mensagens pendentes, reenviará ao coordenador múltiplas cópias da mensagem de IMPOSSIBILITADO. Em algum ponto, porém, uma das mensagens de CANCELE, ainda pendentes, será processada. Isto dará ao nó remoto uma chance de cancelar localmente a transação, enviando ao coordenador uma mensagem de COMPLETADO. A partir deste ponto, as demais mensagens de PREPARE_SE, se houver, serão ignoradas enquanto que as mensagens de CANCELE darão origem a novas mensagens de COMPLETADO que são encaminhadas ao coordenador. Lembre-se que, após enviar a mensagem de COMPLETADO, o protocolo bifásico é terminado. O passo 5, portanto, dita o comportamento acima, pois esta transação já foi liqüidada neste nó remoto. Deste ponto em diante, o comportamento do nó remoto será sempre ditado, no que se refere a esta transação, pelo passo 5 do grupo de ações que diz respeito a um nó participante. O efeito desejado, qual seja, o cancelamento local da transação, já foi conseguido pelo nó remoto. Resta um problema no que tange ao nó remoto. Este pode falhar antes de conseguir completar o passo 4, após ter lido todas as mensagens de CANCELE de que dispunha originalmente. Entretanto, cada vez que se registrar um insucesso na execução das operações previstas no passo 4, este será repetido quando da recuperação do sistema local. Note que, se o nó remoto falhar durante a execução do passo 4, quando o sistema local se recuperar deverá gerar ainda outra mensagem de IMPOSSIBILITADO, a qual o coordenador, no estado de CANCELANDO, responderá com outra mensagem de CANCELE, dando nova oportunidade ao nó remoto de cancelar a transação localmente. Esta troca de mensagens perdura até que o nó remoto despache sua mensagem de COMPLETADO.

A outra possibilidade seria o nó remoto, de início, processar a mensagem de CANCELE antes da mensagem de PREPARE_SE. Como o estado de retorno é DESCONHECIDO, esta mensagem é ignorada. Esta situação perdura até que a primeira mensagem de PREPARE_SE, dentre aquelas ainda pendentes, é analisada pelo nó remoto. Deste ponto em diante, o ciclo descrito no parágrafo anterior é repetido resultando, de novo, no cancelamento local da transação. Em qualquer circunstância, a transação é cancelada neste nó remoto, o que está em consonância com o fato de que este nó falhou quando operava em favor da transação. Lembre-se que esta falha poderá ter acarretado perdas críticas com relação a esta transação e, por conseguinte, o seu cancelamento global é a única alternativa.

Suponha agora que o nó participante, após falhar, encontre um registro indicando o estado de retorno como ATIVO. Segundo o passo 2 do mecanismo de proteção contra falhas, deve verificar se os efeitos locais da transação sobreviveram intatos e, mais ainda, se tem condições de requisitar todos os recursos locais porventura necessários para levar a cabo a execução da transação. Quando houver conseguido, e dependendo do caso, o nó remoto muda o estado de retorno para PRONTO_SIM ou PRONTO_NAO conforme o caso, resguardando tal mudança em local seguro. Note que o nó remoto pode falhar várias vezes antes de conseguir este intento. Seu estado de retorno, porém, será sempre recuperado como ATIVO, forçando a repetição destes passos. Uma vez que tenha conseguido passar ao estado de PRONTO_SIM ou PRONTO_NAO, o nó remoto envia, em seguida, mensagens de resposta ao coordenador com campo de veredito contendo PREPARADO ou IMPOSSIBILITADO,

respectivamente. Falhas repetidas quando nestes estados causam o reenvio da mesma mensagem ao nó coordenador. Este último está em compasso de espera aguardando tais mensagens, no passo 1 da rotina correspondente aos nós coordenadores. Concluímos que o nó coordenador não deve apenas *contar* as mensagens de resposta que chegam e cujos campos de veredito contenham PREPARADO ou IMPOSSIBILITADO. Assim fazendo, ficaria exposto à situação onde um nó remoto remete múltiplas mensagens com campo de veredito PREPARADO enquanto que um outro nó remoto, por raz&ot.es de grande carga local, responde com atraso enviando uma mensagem de resposta com campo de veredito IMPOSSIBILITADO. Neste caso, o coordenador poderia ser enganado decidindo-se por confirmar a transação visto que poderia inteirar número suficiente de mensagens com campo de veredito PREPARADO antes que o nó sobrecarregado enviasse sua mensagem decisiva. É necessário que o coordenador *identifique* que nós remotos já enviaram suas respostas, ignorando mensagens duplicadas. De forma análoga, é possível que o coordenador também despache várias cópias de mensagens indicando PREPARE_SE a um mesmo nó remoto se vier a falhar, talvez várias vezes, quando está no estado de COLETANDO à espera das mensagens de resposta de todos os nós participantes. Sempre que o coordenador envia múltiplas mensagens de PREPARE_SE a um nó remoto, a partir da segunda cópia que analise o nó remoto só poderá estar no estado de PRONTO_SIM ou PRONTO_NAO. Isto porque a primeira versão já forçou este nó a passar do estado de DESCONHECIDO para o estado de ATIVO e daí para um destes últimos estados. Em qualquer caso, é necessário que o nó remoto retransmita a correspondente mensagem de PREPARADO ou IMPOSSIBILITADO, pois a falha no coordenador p&ot.e a perder a relação dos nós participantes que já haviam sido marcados como "com_resposta". É preciso que o coordenador recomece todo o processo de remarcar. O reenvio de tais mensagens, porém, é previsto nos passos 2 e 3 do grupo de ações correspondentes a um nó participante.

Ao concluir esta seção, gostaríamos de comentar o que se passa quando o coordenador falha durante a execução do protocolo bifásico. A leitura das instruções pertinentes a um nó coordenador no algorítmo acima deixam claro que o racional é bastante semelhante àquele empregado para nós participantes, e que acabamos de discutir. Se falhar quando o estado de retorno registrado é COLETANDO, o nó coordenador forçará que novas mensagens de trabalho, com campo de descrição contendo PREPARE_SE, sejam repetidamente enviadas aos participantes. Esta multiplicidade de mensagens, entretanto, não é prejudicial, conforme já vimos. Em algum ponto, o coordenador consegue despachar uma mensagem a cada participante e acumula as respostas correspondentes, forçando uma mudança para o estado de CONFIRMANDO ou CANCELANDO, conforme os votos que tenha recebido. Uma vez atingidos estes estados, a transação já tem seu destino selado. O coordenador envia mensagens de trabalho a cada participante com campo de descrição contendo CONFIRME ou CANCELE, conforme apropriado. Em seguida, fica à espera das mensagens de resposta dos nós participantes indicando que já liqüidaram a transação localmente. Esperar por confirmações dos participantes é importante pois estes podem falhar antes de liqüidarem a transação localmente. Nesta situação, reverteriam aos estado de PRONTO_SIM ou PRONTO_NAO. No primeiro caso, pelo menos, ficariam sem saber o que fazer, visto que podem tanto ser chamados a confirmar ou a cancelar a transação. Uma vez completadas todas as confirmações, o coordenador pode liqüidar a transação no local de origem. Falhas repetidas do coordenador, quando no estado de CONFIRMANDO causarão a repetição do envio de mensagens CONFIRME a todos os participantes. Reforçamos, novamente, que as hipóteses a respeito dos protocolos de comunicação de dados garantem que as mensagens de CONFIRME ou CANCELE não só chegarão a seus destinatários como estes terão a chance de agir de acordo com o caso, isto é, eventualmente as mensagens serão "lidas" por estes nós

remotos. Agora nos deparamos com uma situação semelhante àquela quando o coordenador estava no estado de COLETANDO, isto é, pode vir a falhar antes que tenha obtido mensagens de COMPLETADO de todos os participantes. Se isto ocorrer, após a falha o coordenador deverá repetir o passo 2, reenviando mensagens de CONFIRME a todos os nós participantes. Observe que, se o nó remoto já liqüidou a transação localmente, então deverá responder ao coordenador com uma mensagem de resposta contendo COMPLETADO em seu campo de veredito, conforme previsto no passo 5. Se o nó remoto ainda não liqüidou a transação, então poderá tentar novamente ao receber a mensagem de CONFIRME vinda do coordenador, para, em seguida, enviar a mensagem de COMPLETADO ao coordenador. Este ciclo pode ser repetido quantas vezes for necessário até que o coordenador consiga obter confirmações de todos os participantes e, então, liqüidar a transação localmente. O mesmo pode ser dito com respeito a falhas repetidas quando no estado de CANCELANDO. Repare, portanto, que toda mensagem que chegue a um nó qualquer e referindo-se a uma transação que já foi liqüidada localmente, será simplesmente ignorada, *exceto* quando se tratrar de uma mensagem de trabalho com campo de descrição contendo CONFIRME ou CANCELE endereçada a um nó participante. Neste caso, este último deve enviar uma mensagem de resposta ao nó remetente em cujo campo de veredito deve constar COMPLETADO. Este esquema evita que entremos numa situação onde teríamos confirmações, confirmações de confirmações, etc., num ciclo sem fim.

Existem outras variações que implemetam este mesmo estratagema básico, por vezes chamado de protocolo bifásico centralizado, que acabamos de apresentar. Referências neste sentido serão incluídas nas notas bibliográficas ao fim deste capítulo. Melhorias, principalmente no que se refere à liberação de determinados recursos bloqueados por transações, podem ser introduzidas no esquema que acabamos de discutir. Em vez de mencioná-las agora, porém, será mais instrutivo esperarmos o momento oportuno quando os mecanismos de controle de concorrência estiverem em pauta. O leitor está convidado a tentar incluir algumas destas otimizações no esquema básico do protocolo bifásico apresentado acima. Caso o protocolo de comunicação de dados não apresente as características que assumimos, mormente no que diz respeito ao envio de confirmações ("acks") somente após a mensagem recebida estar a salvo, mecanismos apropriados teriam que ser enxertados na descrição do protocolo bifásico apresentada acima. Estes enxertos preencheriam aquelas funções vitais que não são implementadas a contento pelo protocolo de comunicação.

NOTAS BIBLIOGRÁFICAS

Draffan e Poole [1980] formam uma boa coletânea de artigos. Procuram cobrir todos os aspectos relativos a banco de dados distribuídos, desde a fase de projeto lógico até os mecanismos mais internos de controle de integridade. Lindsay [1980] contém uma descrição mais ou menos detalhada das fases de início, migração e término de transações, especialmente desta última. O modelo de migração é um tanto geral e mais voltado a processos distribuídos, o que causa alguma diferença com aquele discutido no texto. A parte de término de transações é bem descrita. Também inclui uma boa descrição do mecanismo controle de integridade envolvendo atas e balizadores. Gray [1978] descreve algumas das estratégias usadas na implementação do Sistema R, desenvolvido no laboratório de pesquisas da IBM, em San Jose, California. Discute com algum detalhe o mecanismo de atas, bem como o protocolo bifásico. Lindsay et all. [1979] proporciona uma descrição detalhada do protocolo bifásico. Uma análise de vários métodos usados para terminar transações aparece em Cooper [1982]. Outros protocolos de término de transações são propostos em Skeen [1981] e Skeen [1982]. Alguns dentre estes relaxam a condição de que devemos manter o bloqueio de todos os recursos requisitados pela transação, só liberando-os após esta ter

completado. Em contrapartida, introduzem outros problemas peculiares. Já Mohan, Strong e Finkelstein [1983] descrevem tentativas recentes de se introduzir protocolos mais robustos, por outro lado menos eficientes, para se implementar certos aspectos críticos do término de transações. São os chamados "algorítmos Bizantinos". Hammer e Shipman [1979] descrevem um ambiente distribuído alternativo em relação àquele que adotamos e aborda os problemas de migração e término de transações neste novo cenário. Um modelo de transações e falhas aparece em Lampson e Sturgis [1976]. Rosenkrantz, Stearns e Lewis [1978] discutem modelos formais envolvendo controle de concorrência. No que tange a estruturas de dados em geral e árvores em particular, Knuth [1968] e também Horowitz e Sahni [1976] são execelentes referências. Tanembaum [1981] contém uma descrição bem estruturada de protocolos usados em redes de comunicação de dados, bem como da arquitetura destas redes. O texto de Menascé e Schwabe [1894], em português e mais recente, cobre estes mesmos tópicos.

CAPÍTULO 8. CONTROLE DE CONCORRÊNCIA

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

8.1 INTRODUÇÃO

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

8.1.1 Anomalias de Sincronização

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

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

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

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

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

Há três critérios de consistência para este banco de dados:

C1. o CODIGO de cada curso é unico.

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

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

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

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

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

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

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

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

Assim a seqüência

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

indica uma execução seqüencial das transações MATRICULE_1(m,c), LISTE(m), CANCELE(c) e MATRICULE_2(m,c), nesta ordem.

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

M1 C1 C2 M2 M3

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

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

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

L1 C1 C2 L2

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

Para concluir esta seqüência de exemplos, suponha novamente que o curso INF2045 exista inicialmente. Considere uma execução concorrente de MATRICULE_1(82.5782,INF2045) e MATRICULE_2(82.4920,INF2045) representada pela seguinte seqüência:

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

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

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

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

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

8.1.2 Modelagem do Sistema

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

T_1 lê o objeto físico y_1

T_2 lê o objeto físico x_1

T_2 escreve no objeto físico x_1

T_1 escreve no objeto físico x_1

Para o segundo nó, L_2 representa a seguinte seqüência:

T_1 lê o objeto físico x_2

T_1 escreve no objeto físico x_2

T_1 escreve no objeto físico x_1

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

8.1.3 Critérios de Correção

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

Os critérios para transações são simples:

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

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

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

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

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

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

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

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

C1. Cada transação submetida ao sistema deve eventualmente terminar.

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

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

*8.2 TEORIA DA SERIALIZAÇÃO

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

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

8.2.1 Execuções Seriais

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

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

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

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

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

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

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

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

Como exemplos de escalonamentos não seriais teríamos:

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

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

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

e

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

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

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

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

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

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

8.2.2 Equivalência de Execuções

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

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

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

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

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

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

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

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

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

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

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

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

8.2.3 Execuções Serializáveis

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

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

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

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

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

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

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

Há apenas dois possíveis escalonamentos seriais neste caso:

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

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

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

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

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

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

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

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

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

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

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

Como exemplos de escalonamentos não serializáveis teríamos

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

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

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

e

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

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

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

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

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

8.2.4 Uma Condição Suficiente para Serialização

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

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

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

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

Podemos, então, mostrar o seguinte:

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

Demostração

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3 MÉTODOS BASEADOS EM BLOQUEIOS - PARTE I

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

8.3.1 Protocolo de Bloqueio de Objetos

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

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

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

livre não permite acesso ao objeto por nenhuma transação.

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

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

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

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

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

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

x é o nome de um objeto

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

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

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

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

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

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

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

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

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

livre: não permite acesso ao objeto por nenhuma transação;

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

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

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

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

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

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

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

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

Com isto encerra-se a discussão preliminar sobre bloqueios.

8.3.2 Tratamento de Bloqueios Mútuos

8.3.2.1 Caracterização de Bloqueios Mútuos

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

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

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

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

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

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

8.3.2.2 Detecção / Resolução de Bloqueios Mútuos

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

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

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

8.3.2.3 Prevenção de Bloqueios Mútuos

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

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

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

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

O teste escolhido deverá sempre garantir que bloqueios mútuos não irão ser criados ao adicionar T_i à fila de espera de x. Em termos do grafo de espera, isto significa que a adição do arco (T_i,T_j) ao grafo não irá criar ciclos. Há vários testes possíveis com esta propriedade. O mais simples seria sempre reiniciar T_i ao solicitar o bloqueio, que geraria um desperdício grande de recursos. Dois testes mais razoáveis, baseados em prioridades dadas às transações, seriam

Versão não-preemptiva: deixe T_i esperar por T_j se e somente se T_j tiver prioridade menor do que T_i; caso contrário cancele T_i.

Versão preemptiva: deixe T_i esperar por T_j se e somente se T_j tiver prioridade maior do que T_i; caso contrário cancele T_j.

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

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

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

Técnicas preemptivas requerem um cuidado adicional. Caso a transação já tenha sido

confirmada, ou seja, caso uma decisão já foi tomada para instalar as modifiçãoes produzidas pela transação no banco de dados, a transação não poderá ser cancelada. Para evitar este problema, deve-se garantir que, ao atingir esta fase, a transação detenha todos os bloqueios que precisa. Assim, não esperará por nenhuma outra transação, o que implica em que não participa de nenhuma seqüência de impasse e, portanto, irá terminar normalmente.

8.3.3 Protocolo de Bloqueio em Duas Fases

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

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

2) *E* é serializável.

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

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

1) bloqueie cada objeto antes de acessá-lo;

2) libere cada objeto imediatamente após acessá-lo.

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

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

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

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

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

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

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

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

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

Note que, em *M*, a transação T_1 libera *x* após $R_1(x)$, voltando à bloqueá-lo antes de $W_1(x)$, o que constitui uma violação da condição 2 do protocolo. Note ainda que *L* é serializável, mas não *F*.

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

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

Este escalonamento é serializável por ser equivalente a

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

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

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

*8.3.4 Correção do Protocolo de Bloqueio em Duas Fases

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

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

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

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

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

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

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

Suponha que $T_i < T_j$. Então existem ações $O_i(X)$ e $O_i(X)$ de T_i e T_j, respectivamente, tais que $O_i(X)$ e $O_j(Y)$ conflitam e $O_i(X)$ precede $O_j(Y)$. Logo, existe um objeto $x \in X \cap Y$. Pela condição 1 do protocolo de bloqueio em duas fases, uma ação $B_i(x)$ de T_i terá que preceder $O_i(X)$ e, da mesma forma, uma ação $B_j(x)$ de T_j terá que preceder $O_i(X)$. Pelo protocolo de bloqueio / liberação de objetos, uma ação $L_i(Z)$ de T_i tal que $x \in Z$ terá que suceder $O_i(X)$ e

preceder $B_j(x)$. Por definição, o ponto de bloqueio de T_i terá que preceder ou coincidir com $L_i(Z)$. Mas, pela condição 2 do protocolo de bloqueio em duas fases, o ponto de bloqueio de T_j terá que suceder $B_j(x)$. Logo, por transitividade, o ponto de bloqueio de T_i precede o ponto de bloqueio de T_j em L. Assim, $T_i \rightarrow T_j$.

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

8.3.5 Uma Implementação Centralizada do Protocolo de Bloqueio em Duas Fases

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

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

2) se a transação foi processada normalmente:

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

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

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

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

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

8.4 MÉTODOS BASEADOS EM BLOQUEIOS - PARTE II

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

Em todas as implementações assume-se que as transações são processadas conforme descrito na Seção 8.1.2. Em particular, todas as modificações são armazenadas em uma área de trabalho até que a transação execute um comando FIM-DE-TRANSAÇÃO, quando o protocolo bifásico para confirmar intensões é invocado para instalar as modificações criadas pela transação, ou rejeitá-las, cancelando a transação.

8.4.1 Implementação Básica

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

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

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

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

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

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

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

8.4.2 Implementação por Cópias Primárias

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

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

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

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

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

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

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

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

8.4.3 Implementação por Bloqueio Centralizado

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

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

Esta implementação oferece como vantagem, conforme já mencionado, a facilidade de deteção/resolução de bloqueios mútuos pois toda a tabela de bloqueios reside em um único

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

8.4.4 Tratamento de Bloqueios Mútuos no Caso Distribuído

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

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

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

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

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

a *f*, baseando-se na tabela de bloqueios local, e tenta detetar e resolver bloqueios mútuos locais a *f*; em seguida envia o subgrafo para o seu pai. Um nó interior *n*, ao receber os subgrafos dos seus filhos, e após construir o seu próprio subgrafo local, faz a união de todos estes subgrafos e tenta detetar bloqueios mútuos locais a *N*, onde *N* é o conjunto dos nós da subárvore cuja raiz é *n*; em seguida, envia o subgrafo consolidado para seu pai e assim por diante até a raiz.

8.5 MÉTODOS BASEADOS EM PRÉ-ORDENAÇÃO

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

8.5.1 Protocolo de Pré-Ordenação

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

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

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

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

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

É interessante fazer também uma analogia entre o protocolo de pré-ordenação e uma disciplina às vezes usada em estabelecimentos (bem organizados) em que o cliente, ao chegar, apanha uma senha e aguarda a vez para ser atendido. Se houver apenas um

empregado e o cliente for atendido sem interrupção, a situação se torna equivalente a permitir apenas execuções seriais. O caso interessante acontece quando vários empregados atendem a vários pedidos de vários clientes ao mesmo tempo. A disciplina de senhas (isto é, o próprio protocolo de pré-ordenação) deve necessariamente harmonizar todo o trabalho de tal forma que o cliente com senha *i* tenha sempre a ilusão de que foi atendido antes do cliente com senha *j*, se $i<j$ (e, portanto, não reclame do aparente caos reinante).

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

8.5.2 Implementação Básica

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

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

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

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

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

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

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

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

Esta implementação corretamente processa ações conflitantes em ordem de senha. De fato, seja *E* uma execução seguindo esta implementação e modelada por um escalonamento global *L*. Sejam O_i e O_j ações em um escalonamento local de *L* com senhas s_i e s_j. Suponha que O_i precede O_j e que O_i conflita com O_j. Seja *x* um objeto acessado por O_i e O_j (*x* existe pois estas

ações conflitam). Suponha que O_i é uma leitura (logo O_j tem que ser uma atualização). Como O_i precede O_j, temos como resultado dos testes que $s_i < R\_senha(x) < s_j$. Logo, O_i e O_j foram processadas em ordem de senha, como se queria demonstrar. O caso em que O_i é uma atualização é análogo.

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

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

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

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

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

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

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

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

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

para armazenar *R_senha* e *W_senha* ou reiniciar transações com mais freqüência.

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

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

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

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

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

senhas quando reiniciar transações. Este esquema simples naturalmente não evita completamente o problema de reinício cíclico, mas o torna pouco provável.

8.5.3 Implementação Conservativa

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

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

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

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

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

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

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

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

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

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

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

Esta solução, por sua vez, cria a necessidade de comunicação adicional entre os gerentes de transações e nós onde o banco está armazenado, que pode se tornar proibitiva em uma rede de grandes proporções. Para solucionar este último problema, um gerente de transações *g* poderá enviar uma mensagem de controle a um nó *i* contendo uma senha *n* maior do que a sua corrente indicando que não pretende mandar para *i* nenhum subcomando com senha menor do que *n*. Esta mensagem de controle não precisa ser repetida até que as senhas dadas por *g* cheguem a *n*. Naturalmente, deverá haver um mecanismo adicional para revogar esta decisão, caso *g* tenha necessidade de enviar a *i* um subcomando com senha menor do que *n*.

8.5.4 Implementação Baseada em Versões Múltiplas

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

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

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

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

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

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

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

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

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

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

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

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

$R\_seq(x) = (5,8,15,...,92)$

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

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

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

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

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

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

8.6 COMPARAÇÃO ENTRE OS MÉTODOS

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

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

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

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

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

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

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

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

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

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

pessimista: caracterizado por uma elevada percentagem de ações conflitantes

otimista: caracterizado por uma baixa percentagem de ações conflitantes

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

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

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

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

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

NOTAS BIBLIOGRÁFICAS

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

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

CAPÍTULO 9. CONTROLE DE INTEGRIDADE

Os tipos de falhas que podem acometer cada nó do sistema foram descritos e classificados no Capítulo 6. Este mesmo capítulo contém uma breve descrição dos principais mecanismos que são usados para se implementar a função de controle de integridade. O sucesso desta operação é vital para assegurar o próprio princípio de atomicidade em relação a cada transação submetida ao SGBD. Alguns dentre aqueles métodos serão agora focalizados em mais detalhe.

No Capítulo 7 foi apresentada a integração funcional de todos os módulos que compõem o SGBD em cada nó da rede. Os mecanismos de controle de integridade lidam apenas com ações elementares estando, portanto, posicionados nas camadas inferiores que compõem a hierarquia local do SGBD. Relembrando, o SGBD local em cada nó tem sob seu controle os agentes locais que operam em benefício das várias transações que aí se originaram, ou que vieram a migrar para este nó. Cada agente local já recebe do SGBD local uma seqüência de ações elementares que, por sua vez, formam parte de uma transação submetida pelo usuário em algum nó da rede. Os agentes locais devem zelar para que as ações elementares que recebem sejam executadas na ordem especificada. Operando no próximo nível mais interior está o mecanismo de controle de concorrência. Este último é responsável por serializar as ações elementares de todas as transações que estão ativas neste nó de modo a evitar inconsistências. Os mecanismos de controle de integridade vêem logo a seguir na hierarquia local. À medida que o controle de concorrência libera ações elementares, estas passam pelo controle de integridade antes de acionarem o sistema operacional local quando, então, materializam o acesso ao BD local.

Com este cenário em mente, podemos passar à descrição de alguns métodos empregados para efetivar a função de controle de integridade.

9.1 ATAS

Dentre todas as maneiras de se proporcionar segurança contra os efeitos maléficos de eventuais falhas, o seguinte paradigma expõe uma idéia bastante singela:

- sempre que a invocação de uma ação elementar implicar numa mudança de estado do BD, os estados anterior e posterior à invocação da ação são postos a salvo em lugar seguro;
- ocorrendo uma falha, uma busca junto aos registros que foram salvos permitirá reconstruir um estado consistente do BD.

O mecanismo de atas (ou, em inglês, "logs", "audit trails", "journals") se propõe a implementar este paradigma. Conquanto a idéia básica seja simples, devido ao fato que operam em ambientes complexos, atas enfrentam problemas de sincronismo, de eficiência e de eficácia que geram mecanismos de controle de integridade bastante sofisticados. Atas garantem, em primeira instância, proteção contra falhas primárias mesmo porque, à exceção de pseudo-falhas, estas são, de longe, os tipos mais freqüentes de falhas com que deve se preocupar um SGBD usual. Dependendo da gravidade e de que áreas de memória secundária ativa foram atingidas por uma falha secundária, o mecanismo de atas pode também recuperar o BD a um estado consistente após a ocorrência de uma falha secundária. Nesta seção, porém, estaremos primordialmente

preocupados em combater falhas primárias. A menos que expressamente declarado em contrário, o termo falha será reservado, nesta seção, a designar falhas primárias. Quanto à organização, iniciaremos com comentários gerais, seguidos de um mecanismo simplificado. À medida que prosseguirmos, aprimoramentos serão introduzidos e discutidos.

9.1.1 Considerações Gerais

Nesta subseção levantaremos pontos importantes que devem ser tidos em mente quando o controle de integridade for descrito em detalhe. Muitos deles já foram abordados antes, de modo que a discussão abaixo servirá para relembrá-los, agora focalizando-os sob os aspectos que tocam mais de perto os mecanismos de controle de integridade.

9.1.1.1 O Sistema de Paginação Local

Obviamente não se pretende salvar o estado completo do BD cada vez que uma ação elementar é executada e provoca mudança de estado no BD. Mesmo em sistemas pequenos, isto implicaria em custos altos demais e, pior, para resguardar informação de forma muito ineficiente. Antes, registros são anotados na ata descrevendo os valores inicial e final de cada objeto modificado, catalogando as mudanças de forma incremental.

Podemos conceber o BD a nível lógico como um conjunto de objetos de alguma forma inter-relacionados. Da mesma forma, podemos visualizar a ata, logicamente, como uma seqüência semi-infinita de registros, cujos conteúdos serão especificados oportunamente. Do ponto de vista físico, cada BD local será entendido como composto por uma série de *páginas de memória*. Cada página é identificada por um *endereço* único e todas as páginas são do mesmo tamanho. Os valores dos vários objetos físicos são codificados em campos internos a estas páginas. Suporemos que há espaço suficiente em cada página para acomodar o valor de qualquer objeto físico do BD, a qualquer tempo. Assim, a cada objeto físico podemos também associar um único *endereço*. Este endereço poderia ser tomado como um par ordenado contendo o endereço da página onde se localiza o objeto seguido de um endereço relativo ao começo desta página, especificando completamente a posição deste particular objeto. Em acréscimo, outras informações, tais como o número de "bits" reservado a este objeto, podem também fazer parte de seu endereço. Este esquema está em consonância com o fato de que estamos operando a nível de ações elementares, manipulando apenas objetos físicos do BD. Mais ainda, esta visão forma uma interface natural com o sistema operacional local que, provavelmente, já implementa o conceito de paginação. Em vista desta última observação, nada mais natural que acomodar a própria ata, fisicamente, também em páginas de memória.

Imaginamos que as páginas de memória onde residem o BD e a ata localizem-se em memória secundária a salvo, portanto, de quaisquer danos causados por falhas primárias. Ocorre que o conteúdo de memória secundária não pode ser modificado "in loco". Para modificar o conteúdo de uma página, quer referente ao BD local ou à ata, o sistema operacional local deve

1. ler a página em questão para a memória principal;
2. modificar seu conteúdo como desejado, na memória principal;
3. reescrever a página de volta na memória secundária, destruindo a cópia original

Durante instantes críticos, partes do BD ou da ata devem ser trazidos a memória principal para modificações. Não seria uma boa idéia reescrever cada página de volta à memória secundária após cada modificação. Isto geraria tráfego excessivo entre a memória principal e a memória secundária, o que poderia degradar sensivelmente o tempo de resposta do sistema, visto que acessos a memória secundária são relativamente lentos quando comparados a operações em memória principal. É desejável, portanto, manter certas páginas em memória principal por períodos mais prolongados. Em conseqüência, páginas de memória principal poderão sofrer muitas alterações antes de ser reescritas em memória secundária, tanto por transações em andamento como por outras transações que já confirmaram ou cancelaram e mesmo já desapareceram do BD. Falhas primárias poderão destruir o conteúdo destas páginas antes que tenham uma chance de serem reenviadas a memória secundária. Mesmo estando dispostos a pagar o preço de ter maior tráfego entre memória principal e memória secundária, ainda não esta afastado o risco do sistema falhar em momentos críticos, destruindo o conteúdo de páginas que não puderam ser reescritas em memória secundária. Note que as páginas que descrevem a própria ata são afetadas por este tipo de problema. O mecanismo de controle de integridade deve proteger não apenas o BD local como o conteúdo da própria ata que usa para garantir a segurança do sistema todo. Enfatizamos que o processo de reescrita em memória secundária vai obliterar o conteúdo anterior da correspondente página em memória secundária. Há mecanismos de controle de integridade que reescrevem a nova cópia em áreas de trabalho na memória secundária de forma a não destruir a cópia original. Tais mecanismos serão alvo de discussão em seções posteriores.

A memória secundária dispõe de capacidade de armazenamento muito superior àquela da memória principal. Na realidade, neste estágio inicial, estamos supondo que a memória secundária é ilimitada, pois definimos que a própria ata é capaz de armazenar incontáveis registros. De qualquer forma, é necessário que páginas sejam recolocadas em memória secundária, de tempos em tempos, para abrir espaço para que novas páginas possam ser lidas para a memória principal. Suporemos que a memória principal contém uma área específica, que chamaremos de *área de trabalho*, a qual pode conter um número finito, porém não especificado, de páginas de memória. O trânsito de páginas entre a área de trabalho e a memória secundária é controlado por um *gerente de paginação*. Este último implementa protocolos para, por exemplo, escolher qual a próxima página que deve sair da área de trabalho para que outra página, cuja leitura foi requisitada, seja colocada na área de trabalho e posta a disposição do processo requisitante. O particular algoritmo de substituição de páginas na memória principal quando a área de trabalho encontra-se saturada é de responsabilidade do sistema operacional local. O mecanismo de controle de integridade poderá interferir no comportamento do algoritmo de duas formas. Em primeiro lugar, terá poderes para *forçar* com que páginas sejam recolocadas na memória secundária quando assim o determinar. Em segundo lugar, poderá também decretar que páginas da área de trabalho sejam *mantidas* na memória principal até segunda ordem. A menos destas interferências, necessárias para que as ações de ambos os módulos sejam sincronizadas apropriadamente, o mecanismo de atas permite que o gerenciador de paginação siga suas próprias estratégias de controle do tráfego de páginas. Em particular, a *ordem* de entrada e saída de páginas na memória principal ou o tempo de permanência de cada página na área de trabalho, são de livre escolha do gerenciador de paginação local. Deve ficar claro, porém, que há dois mecanismos decidindo a respeito de quando e quais páginas são trazidas da memória secundária ou para lá são enviadas: o gerenciador de paginação e o controle de integridade locais.

Em geral, teremos parte da área de trabalho ocupada por páginas que representam objetos do BD e outro tanto tomado por páginas que descrevem partes da ata. Da mesma forma, na memória secundária haverá páginas correspondentes ao BD e outras que completam a ata. O desafio encarado pelo controle de integridade está em orquestrar as idas e vindas de páginas entre a memória principal e a memória secundária, de tal sorte a garantir que, na eventualidade de uma falha primária, a parte da ata que não foi afetada contenha informação suficiente para reconstruir um estado consistente do BD e também recuperar a própria ata. Lembre que falhas primárias destróem o conteúdo da memória principal e, em conseqüência, inutilizam também o conteúdo da área de trabalho. A seqüência dos acontecimentos pode ser de tal ordem que páginas com modificações relativas a transações que já confirmaram, desaparecendo do BD, ainda se encontravam na área de trabalho no instante da falha. Também, apenas parte das páginas envolvidas com transações que cancelaram podem ter sido reenviadas para a memória secundária, consolidando a operação de desfazer os efeitos da transação. Entretanto, outras páginas afetadas por esta transação podem ainda não ter sido reescritas em memória secundária no intervalo compreendido entre o instante em que o efeito das ações elementares foram invertidos até o momento da falha. Assim, a operação de desfazer os efeitos da transação ainda não teve chance de ser registrada a salvo, estando aquela página inconsistente. Há também a preocupação em torno da eficiência do processo de recuperação e, ainda, cuidados devem ser tomados para que a manipulação da ata não degrade o sistema de forma significativa.

9.1.1.2 Início, Migração e Término de Transações

Quando do início de uma transação, antes de entrar na execução de sua primeira ação elementar, o agente local da transação instala em lugar seguro, isto é *na ata*, um registro descrevendo este fato. Chamaremos este registro de *registro de início de transação* ou simplesmente registro de início. Conforme exposto no Capítulo 7, o conteúdo deste registro varia segundo este é o nó de origem da transação ou não. Esta diferença, no entanto, é explorada apenas na fase de migração da transação, sendo, neste ponto, imaterial para a discussão que segue. O importante é que cada transação ativa num dado nó dispõe de um único registro de início instalado na ata.

Durante a fase de migração, um *registro de execução* é lançado na ata para cada ação elementar invocada em favor desta transação e cuja invocação resulte na modificação do valor do objeto. Além do endereço do objeto modificado, o registro de execução contém também os valores iniciais e finais do objeto. Se a transação for forçada a cancelar localmente, a ata deve receber um *registro de cancelamento local da transação* ou registro de cancelamento local.

Na fase de término, há todo um ritual que deve ser observado para confirmar e cancelar transações, já descrito no Capítulo 7. Transações só são consideradas *confirmadas* em um dado nó quando, ao executar a segunda fase do protocolo bifásico, o sistema local instala na ata um *registro de confirmação da transação* ou simplesmente registro de confirmação. Por outro lado, transações são *canceladas* num particular nó quando o sistema lança na ata um *registro de cancelamento da transação* ou, de forma curta, um registro de cancelamento. É importante relembrar que *antes* de escrever o registro de cancelamento, o protocolo bifásico cuida para que todas as ações elementares desta transação sejam invertidas, isto é, seus efeitos são removidos. Só após o sucesso desta operação o registro de cancelamento é instalado. Observe que as ações elementares invocadas para inverter os efeitos das ações normais da transação *não* deixam traços de sua invocação na ata. Esta contém apenas os registros referentes as ações elementares invocadas antes do ponto em que o cancelamento foi decidido.

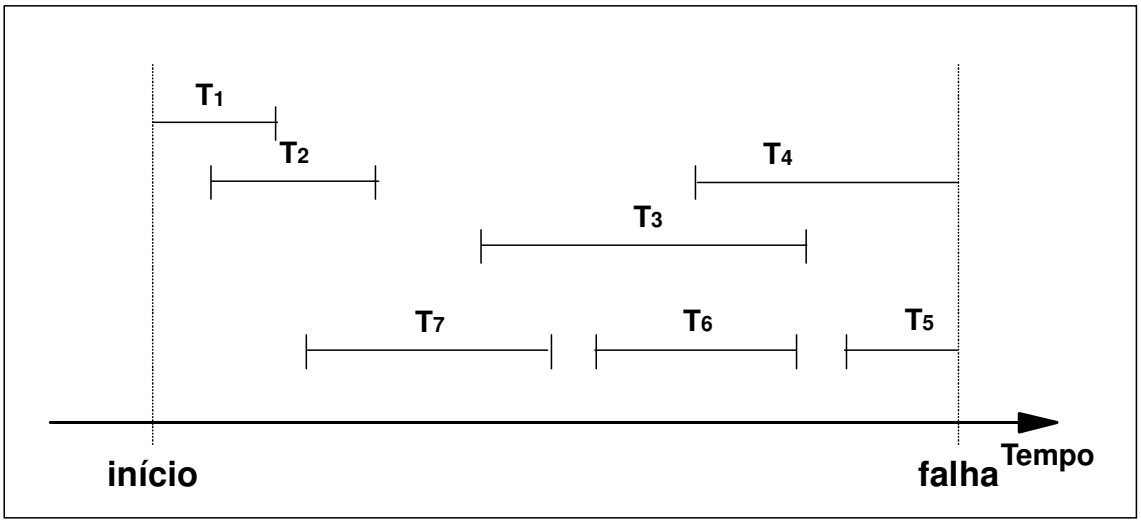

Figura 9.1 - Transações Executando

O último tipo de registro de que necessitamos são *registros de término* de transação, ou simplesmente registros de término. São precisamente aqueles registros lançados na ata pelo protocolo bifásico durante a fase de término da transação, conforme previsto na descrição deste protocolo no Capítulo 7. Descrevem os estados por que passa o protocolo ao executar localmente em favor de determinada transação, quer seja este o nó coordenador ou apenas um nó participante.

9.1.1.3 O Controle de Concorrência

Lembre que as ações elementares de todas as transações ativas em um dado momento devem ser serializadas pelo mecanismo de controle de concorrência. Esta serialização é obtida sem se levar em conta possíveis ações elementares que venham a anular os efeitos de ações anteriores, como requerido quando devemos desfazer os efeitos de transações. Refazer e desfazer os efeitos de ações elementares para que transações tenham seus efeitos reconfirmados ou recancelados, cria a expectativa de que certas ações elementares serão executadas contra o BD à revelia dos mecanismos de controle de concorrência. Em princípio isto poderia configurar cenários onde houvesse violação da consistência do BD devido a serializações estranhas. Lembre, porém, que a tarefa fundamental dos módulos de controle de concorrência é propiciar uma serialização das ações elementares de tal sorte que *uma transação não tem como acessar objetos modificados por outra transação que ainda não tenha completado*. Assim, num dado instante, o uso de cada objeto do BD é privativo da transação ativa naquele instante que irá modificar o seu valor. Um mesmo objeto pode ser *lido* por várias transações, desde que nenhuma outra transação em andamento venha a *modificar* seu valor. Neste último caso, claro, não há mal em assim se proceder pois que a operação de leitura não altera o valor do objeto. Tome como exemplo a situação ilustrada na Figura 9.1, onde T_4 e T_5 estariam incompletas no instante da falha.

Digamos que T_2 e T_6 hajam cancelado enquanto que as demais, T_1, T_3 e T_7 tenham confirmado. Neste exemplo, T_3 e T_6 não poderiam estar modificando o valor de um mesmo objeto. O mesmo pode ser dito quanto a T_1 e T_7, e assim por diante. Já T_1, T_3 e T_5 poderiam ter sido executadas envolvendo modificações sobre o valor de um mesmo objeto.

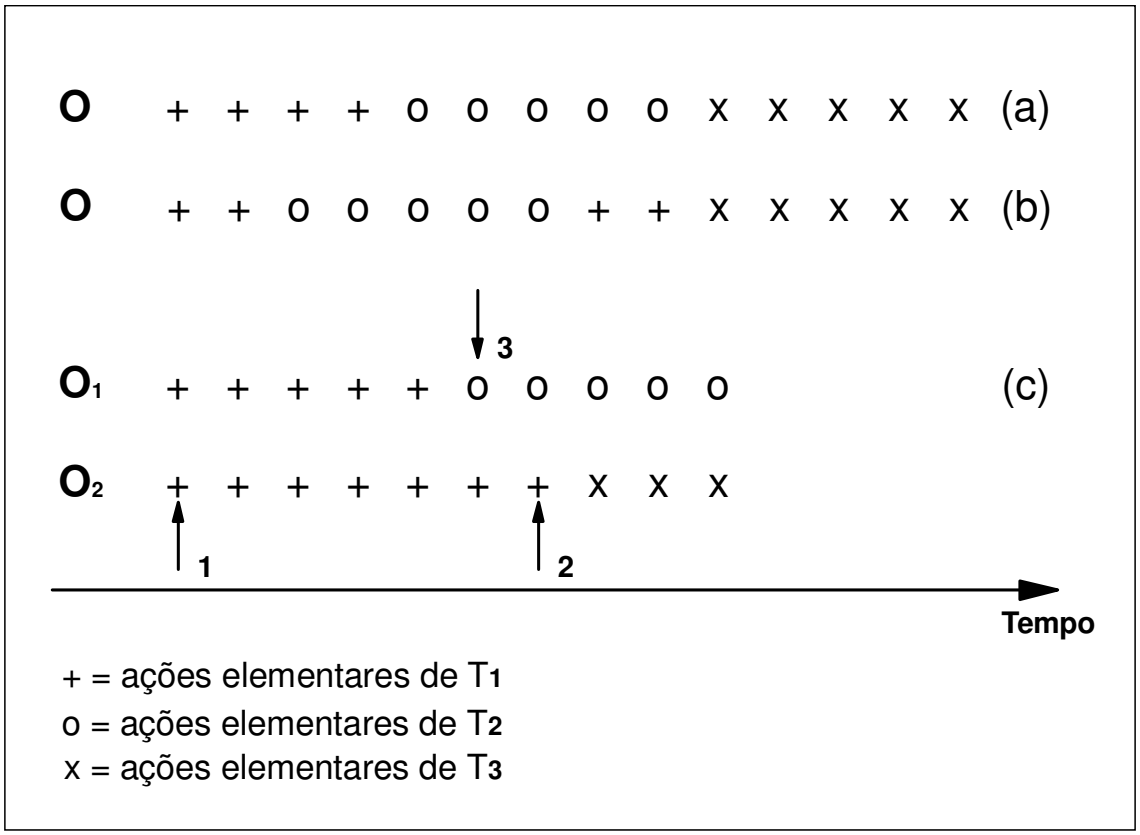

Figura 9.2 - O Princípio da Não Interferência

Suponha que agrupemos as ações elementares em classes, uma para cada objeto do BD, de tal sorte que ações pertencentes a mesma classe afetam o mesmo objeto. Então podemos dizer que, se agrupadas por transações, a seqüência de ações de uma mesma classe devem ocorrer em blocos completos de ações referentes a uma mesma transação. A Figura 9.2 ilustra este fato, onde as transações T_1, T_2 e T_3 são supostas pertencerem a uma mesma classe.

A seqüência (a) seria válida, pois as ações elementares não estão misturadas. Uma seqüência de ações de uma transação não é interrompida por seqüências de ações de outras transações. No caso da seqüência (b), o controle de concorrência não operou de forma correta, pois ações da transação T_2 foram enxertadas entre as ações de T_1, o que pode levar a toda sorte de problemas, conforme discutido no Capítulo 8. Mesmo que não haja problemas quando tomamos cada classe isoladamente, isto não é suficiente para garantir que a máxima da não interferência enunciada acima é observada no conjunto. Na parte (c) da mesma figura, temos as seqüências correspondentes as classes de ações elementares que modificam os objetos O_1 e O_2. Tomadas separadamente, as seqüências relativas a O_1 e O_2 apresentam os blocos completos que desejamos. As setas 1 e 2 indicam um intervalo de tempo quando a transação T_1 estava certamente ativa. A seta 3, porém, indica que T_2 modificou o objeto O_1, também modificado por T_1, antes que T_1 completasse. Este cenário viola o princípio de que a uma transação não é permitido modificar o valor de objetos que são modificados por uma segunda transação que ainda não completou.

Desta discussão podemos concluir que a ação de desfazer os efeitos locais de uma transação não requer que, em conseqüência deste ato, tenhamos que desfazer também os efeitos de outras transações. A razão para isto se fixa exatamente no fato de que objetos modificados por uma transação são invisíveis a outras transações durante o período em que a primeira está ativa. Se este princípio não fosse observado, ao desfazer os efeitos de uma transação poderíamos nos ver envolvidos em um processo em cascata, sendo forçados a desfazer efeitos de outras transações. Dentre estas poderiam estar transações já confirmadas e eliminadas do BD. Mais ainda, sendo o BD distribuído, este processo de cascata poderia se alastrar para outros nós, com as conseqüências maléficas usuais no tempo de resposta, entre outras. Estando os mecanismos de controle de concorrência a operarem corretamente, porém, cada ação de desfazer os efeitos de ações elementares ficara contido ao nó onde é iniciado.

9.1.2 Um Mecanismo Simplificado

Vamos iniciar o desenvolvimento dos mecanismos de controle de integridade com um modelo ainda irreal cujo objetivo é apresentar as idéias básicas de forma simplificada. Subseções posteriores sofisticarão este modelo básico de forma a torná-lo real e, também, mais eficiente.

Suporemos nesta seção que as modificações no conteúdo da ata possam ser escritas diretamente em memória secundária e de forma atômica. Por conseguinte, todas as páginas da ata residem sempre em memória secundária, mesmo durante os instantes em que são atualizadas, estando a salvo de falhas. A área de trabalho em memória principal contém apenas páginas com dados do BD local.

Para o gasto imediato, podemos tomar cada registro da ata como formado pelos seguintes campos, de acordo com seu tipo.

Os registros de tipo *INÍCIO, CONFIRMAÇÃO, CANCELAMENTO* e *CANC_LOCAL* contêm os campos

identificação
tipo

Os registros de tipo *TÉRMINO* contêm os campos

identificação
tipo
estado de retorno

Os registros de tipo *EXECUÇÃO* contêm os campos

identificação
tipo
endereço
valor inicial
valor final

O significado dos campos identificação e estado de retorno é óbvio: refletem, respectivamente, a identificação da transação que lançou este registro na ata ou um estado de retorno do protocolo bifásico. Os campos de tipo classificam os registros de acordo com sua utilidade ou função. O tipo *INÍCIO* indica o ponto quando a transação foi, pela primeira vez, ativada neste nó. Os tipos *CANCELAMENTO* e *CONFIRMAÇÃO* indicam tanto o instante quando a execução do protocolo bifásico terminou, como também qual o destino global dado à transação: se cancelada ou confirmada, respectivamente. Neste instante, a transação desaparece do BD e, claro, também é eliminada da tabela de transações ativas em poder do executor de transações local. O tipo *CANC_LOCAL* reflete o fato de que esta transação foi cancelada localmente, porém o ritual previsto no protocolo bifásico ainda não foi, até este momento, completado no que diz respeito a esta transação. O executor de transações local ainda mantém a transação na sua tabela de transações ativas, indicando que sofreu cancelamento local. O tipo *TÉRMINO* indica registros que contêm estados de retorno, relativos ao protocolo bifásico. Finalmente, o tipo *EXECUÇÃO* refere-se a registros que revelam os efeitos de ações elementares invocadas em favor desta transação. Seu campo de endereço contém o endereço completo do objeto modificado. Seus campos de valor inicial e valor final contêm os valores deste objeto anterior e posterior à invocação da ação elementar que originou o registro, respectivamente. À medida que os algoritmos forem se tornando mais sofisticados, novos campos serão introduzidos e novos valores serão definidos para o conteúdo do campo de tipo. No presente nível de detalhe, esta descrição será satisfatória. Podemos observar desde já que, na prática, cada registro poderá conter mais informações, tais como o tamanho do registro, necessárias para uma eficiente implementação dos registros da ata a nível físico.

Para facilidade de discurso, vamos classificar as transações de acordo com o seguinte esquema. Seja *T* uma transação qualquer. Diremos que em dado momento e em um dado nó *T* é uma *transação cancelada* se constar na ata um registro de *CANCELAMENTO* ou *CANC_LOCAL* com a identificação de *T*. Chamaremos *T* de uma *transação confirmada* se pudermos localizar na ata um registro tipo *CONFIRMAÇÃO* referente a *T*. Note que, em qualquer momento, T não pode ser classificada como cancelada ou confirmada, simultaneamente. O termo *transação confirmando* será reservado àquelas transações que, não podendo ser classificadas como canceladas ou confirmadas num certo instante, contarem, neste instante, com um registro tipo *TÉRMINO* já lançado na ata em seu benefício. Mais ainda, o campo de estado de retorno deste registro deverá conter *PRONTO_SIM* ou *CONFIRMANDO*. Eventualmente, com o correr do tempo, transações podem mudar de classe. Finalmente, chamaremos de *transação incompleta* àquela transação que não se enquadra em nenhuma das três classes definidas há pouco.

Suponha que num certo instante inicial o BD está em um estado quiescente, isto é, todas as transações invocadas já confirmaram ou cancelaram globalmente. A partir deste momento, transações passam a executar contra o BD, algumas confirmando, outras cancelando. Num dado ponto, ocorre uma falha, colhendo certas transações antes que tenham cancelado ou confirmado. Seja *I* um ponteiro que indica, na ata, o registro correspondente ao início da primeira transação que modificou o conteúdo de qualquer página do BD, desde que o sistema esteve quiescente pela última vez. Mais tarde, o problema de se obter I será contornado. O mecanismo de controle de integridade deverá

desfazer os efeitos de transações incompletas

desfazer os efeitos de transações canceladas

refazer os efeitos de transações confirmadas

refazer os efeitos de transações confirmando

determinar que transações foram colhidas durante a execução do protocolo bifásico

Tomando a ata como uma seqüência que cresce da esquerda para a direita, *I* indica o registro tipo *INÍCIO*, situado mais à esquerda na ata, cuja transação alterou o conteúdo de uma página do BD desde que esteve quiescente pela última vez. Uma primeira idéia para se restaurar a consistência do sistema após uma falha primária é descrita abaixo. São usadas três listas, a saber: *TC*, *TA* e *TR*. A primeira acumulará as transações iniciadas após o ponto *I* que foram colhidas pela falha como confirmadas ou confirmando. A segunda lista será formada por transações iniciadas após *I* que, no momento da falha, seriam classificadas como canceladas ou incompletas. Cada transação em *TA* será marcada como "local" ou "global" segundo tenha sido cancelada local ou globalmente, respectivamente. No curso do algoritmo, como veremos, transações incompletas serão promovidas a canceladas localmente e, assim, também serão marcadas como "local". A última lista, *TR*, conterá a identificação e estado de retorno daquelas transações que são colhidas no instante da falha durante a execução do protocolo bifásico. As três listas agrupam aquelas classes de transações que esperam o mesmo tipo de atitude por parte do controle de integridade durante a fase de recuperação. Repare que a lista *TR* pode ter elementos em comum com as listas *TA* e TC. No primeiro caso, por exemplo, poderíamos ter uma transação que já iniciou a execução do protocolo bifásico, tendo sido cancelada localmente durante a execução da primeira fase. No segundo caso, a transação já lançou na ata um registro de tipo *TÉRMINO*, com campo de estado de retorno contendo *PRONTO_SIM*, porém ainda não finalizou a execução do protocolo bifásico. Podemos, agora, detalhar um pouco mais as operações do mecanismo de controle de integridade.

Numa primeira fase, removemos efeitos de ações elementares. Andando na ata para a esquerda, desde o instante da falha até o ponto *I*, examine cada registro encontrado. Seja *R* o próximo registro a ser examinado. Se o campo de tipo de *R* contiver

1) *CANCELAMENTO*:
   a) ponha a identificação da transação na lista *TA* e marque-a como "global" ;
2) *CONFIRMAÇÃO*:
   a) ponha a identificação da transação na lista *TC* ;
3) *CANC_LOCAL*:
   a) se a transação não está em *TA*,
      então ponha sua identificação em *TA* e marque-a como "local" ;

4) *TÉRMINO*:
   a) se campo de estado de retorno de *R* indicar *PRONTO_SIM* ou *CONFIRMAÇÃO*
      e a transação não está em *TC*
      então ponha a transação na lista *TC* e na lista *TR*, junto com seu estado de retorno ;
   b) se campo de estado de retorno de *R* não indicar *PRONTO_SIM* nem *CONFIRMAÇÃO*
      e a transação não está em *TC* nem em *TR*
      então ponha a transação na lista *TR*, junto com seu estado de retorno
5) *EXECUÇÃO*:
   a) se a transação está em *TA*, não importando sua marca,
      então remova os efeitos desta ação elementar
   b) se a transação não está nem em *TA* nem em *TC*,
      então remova os efeitos desta ação elementar
   c) caso contrário, ignore este registro
6) *INÍCIO*:
   a) se a transação não está em *TA* nem em *TC*,
      então
      i) envie uma mensagem de resposta, com campo de veredicto contendo *CANCELADO*, ao nó coordenador da transação
      ii) construa um registro de ata tipo *CANC_LOCAL* para esta transação
      iii) lance este registro na ata
      iv) acrescente esta transação à lista *TA*, marcando-a como "local"

Numa segunda fase, reinstalamos os efeitos de ações elementares. Andando para a direita, desde *I* até o fim da ata, examine cada registro que encontrar. Seja *R* o próximo registro a ser examinado. Se o campo de tipo de *R* contiver

1) *EXECUÇÃO*:
   a) se esta transação está na lista *TC*,
      então refaça os efeitos desta ação elementar
   b) caso contrário, ignore este registro
2) EM QUALQUER OUTRO CASO:
   a) ignore este registro

O algoritmo lê a ata duas vezes. Na primeira vez, partimos do fim da ata e a percorremos até o ponto indicado por *I*. Note que os registros indicando se uma particular transação confirmou, cancelou ou já foi envolvida na execução do protocolo bifásico só aparecem na ata após todos os

registros de execução da transação. Percorrendo a ata de trás para a frente passaremos pelos registros de tipo *CONFIRMAÇÃO*, *CANCELAMENTO*, *CANC_LOCAL* ou *TÉRMINO* antes de atingirmos qualquer registro de tipo *EXECUÇÃO* de uma mesma transação. Usando este fato, as listas *TC*, *TA* e *TR* são compiladas a medida que prosseguimos. Mantendo estas listas, teremos condições de determinar o estado de cada transação no instante da falha e, assim, tomar as medidas apropriadas quando os registros de *EXECUÇÃO* forem encontrados.

Durante a primeira fase, todas as ações elementares invocadas em benefício de transações que estavam incompletas no instante da falha são desfeitas. Observe que isto é feito na ordem inversa daquela em que estas mesmas ações elementares foram invocadas, como é desejável. O racional para se desfazer os efeitos de tais transações é baseado no fato de que o gerente de paginação local pode já ter enviado a memória secundária algumas das páginas modificadas por esta transação. As mudanças instaladas nestas páginas devem, portanto, ser removidas. Repare que, quando atingimos um registro tipo *EXECUÇÃO* para uma transação incompleta este fato pode ser determinado verificando-se se a transação não consta em *TA* nem em *TC*. Este trabalho de desfazer efeitos estará completo quando atingirmos o registro de tipo *INÍCIO* desta mesma transação. Neste ponto, devemos lançar na ata um registro de tipo *CANC_LOCAL* e, também, adicionar a transação, marcada como "local", na lista *TA*. O resultado provocado pela falha será idêntico a um cancelamento local da transação. Os efeitos de transações que já haviam cancelado, localmente ou globalmente, devem também ser desfeitos nesta primeira fase. Este tipo de transações são acumuladas na lista *TA*. A necessidade de remover os efeitos destas transações advém do fato de que o gerente de paginação pode ainda não ter reenviado algumas páginas modificadas pela transação à memória secundária desde o instante em que suas ações elementares foram desfeitas. Neste caso, a inversão destes efeitos ainda residia na memória principal no instante da falha e foram, portanto, perdidas. Lembre que sempre desfazemos os efeitos de todas as ações elementares de uma transação antes de lançar na ata registros tipo *CANCELAMENTO* ou *CANC_LOCAL*.

Observe que, quando o algoritmo terminar, não mais haverá transações incompletas no sistema, neste momento.

A segunda fase do algoritmo simplesmente percorre a ata desde o ponto indicado por *I* até seu fim. Ao examinar cada registro tipo *EXECUÇÃO*, a correspondente ação elementar terá seus efeitos reinstalados caso a respectiva transação conste da lista *TC*, de transações que devem confirmar. A ação elementar que será refeita, claro, está descrita pelos campos de valor inicial, valor final e endereço presentes no registro ora em exame.

Em seguida, o controle de integridade informa ao executor de transações local quais transações estavam no estado de canceladas localmente. Esta informação será necessária quando o executor de transações local participar do ritual prescrito pelo protocolo bifásico, quando este executar em favor desta transação. De forma análoga, o executor de transações local precisa saber a identidade e estado de retorno correspondente a cada transação que foi colhida pela falha quando o protocolo bifásico executava em seu benefício. Neste último caso, conforme discutimos no Capítulo 7, o executor de transações local não esperará o início da execução do protocolo bifásico pelo nó de origem da transação. Em vez disso, retoma a execução do protocolo a partir do estado de retorno que lhe é informado pelo controle de integridade.

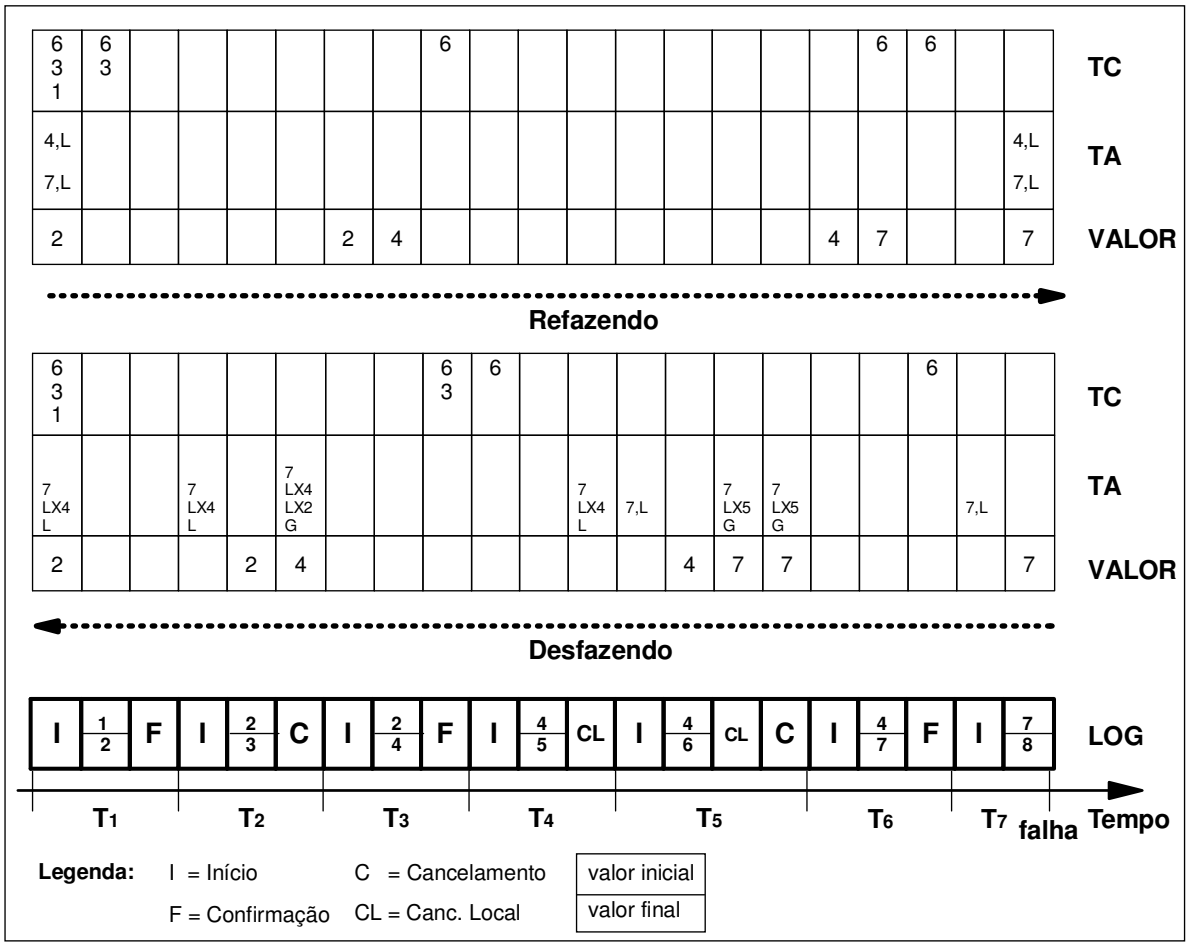

Figura 9.3 - Recuperando-se de Uma Falha Primária

Considere a Figura 9.3. onde a execução do algoritmo é ilustrada. Para simplificar, estamos mostrando apenas o que ocorre com o valor de um objeto, *O*, afetado pelas transações que detêm registros na ata à direita de *I*. A figura contém, na primeira coluna, uma fotografia do trecho da ata desde o registro apontado por *I* até o instante da falha. Esta coluna deve ser lida de cima para baixo. Cada transação foi limitada a um único registro tipo *EXECUÇÃO* para manter o exemplo sob controle. A segunda coluna, lida de baixo para cima, reflete a fase de desfazer ações elementares. Esta coluna é dividida em três sub-colunas. Estas últimas mostram o efeito do processamento de cada registro da ata sobre o valor do objeto e também sobre o conteúdo das listas *TA* e *TC*. Para não sobrecarregar a figura, adotamos a convenção de que, em cada coluna, um espaço em branco indica o mesmo valor dos extremos verticais que o contêm. Na coluna referente à lista *TA* os elementos são indicados por pares, onde o primeiro componente refere-se a transação em questão e o segundo indica "L" ou "G", conforme a correspondente transação esteja marcada "local" ou "global" em *TA*, respectivamente. Note que não podemos ter, no intervalo de tempo ilustrado na referida figura, outras transações que modificaram o valor deste mesmo objeto e que se encontrem, no instante da falha, envolvidas na execução do protocolo bifásico. Isto se deve ao fato de que estamos analisando apenas um objeto e a máxima da não interferência entre transações força este cenário. Lembre que os recursos do sistema requisitados

por cada transação só são liberados quando o protocolo bifásico termina ou quando a transação é cancelada localmente. A lista *TR* seria, portanto, iniciada como vazia e terminaria como tal. Devido a este fato, não a incluímos na figura. Quando mais de um objeto estão envolvidos poderemos ter, num dado momento, vários elementos em *TR*. A terceira coluna da Figura 9.3, lida de cima para baixo, mostra a fase quando ações elementares são refeitas. Valem os mesmos comentários colocados a respeito da segunda coluna. Dadas as descrições de cada registro da ata é fácil ver que todos os passo do algoritmo são exeqüíveis. Por exemplo, cada registro tipo *EXECUÇÃO* contém o endereço mais os valores iniciais e finais do objeto a que se refere seu campo de identificação. Assim, podemos refazer/desfazer os efeitos daquela ação elementar a que corresponde o registro.

Um aspecto importante a notar é que o algoritmo apresenta imunidade contra falhas em cascata, isto é, quando o sistema torna a falhar *durante* a execução do próprio algoritmo de recuperação. Observe que as operações de refazer/desfazer os efeitos de ações elementares podem ser repetidas à vontade, sem introduzirem valores estranhos por estarmos refazendo/desfazendo a mesma ação várias vezes. Isto porque a ação de desfazer limita-se a substituir o valor do objeto pelo conteúdo do campo de valor inicial obtido do registro da ata que estamos a examinar naquele instante. O mesmo se dá, em sentido inverso, para a ação de refazer. O que pode variar no caso de falhas em cascata é o conteúdo da lista *TA*, visto que a primeira tentativa de recuperação pode ter introduzido na ata registros de tipo *CANC_LOCAL* para transações que antes eram tidas como incompletas. Isto, porém, não influirá no resultado final do algoritmo. A hipótese de perdermos parte da ata, junto com o conteúdo da memória principal, está afastada neste modelo simplificado. Assim, o conteúdo da ata, tomado a partir do ponteiro *I*, é sempre suficiente para restaurar o BD local a um estado quase-quiescente, quer ocorram ou não falhas em cascata. Referimo-nos ao estado de retorno como quase-quiescente devido ao fato de que as transações em *TR* ainda não confirmaram ou cancelaram globalmente. Entretanto, após a recuperação do sistema local, não há qualquer transação, ativa localmente, que ainda esteja na fase de migração.

9.1.3 Um Mecanismo Mais Elaborado

O modelo simplificado da seção anterior apresentava um algoritmo para controle de integridade bastante ineficiente. Tudo se passava como se estivéssemos repetindo, um tanto às avessas, todas as ações elementares que foram invocadas desde o instante em que se deu a última falha. Como desejaríamos que os intervalos entre falhas fossem os mais longos possíveis, este particular esquema incorpora muita ineficiência. Nesta seção, vamos descrever novas técnicas que nos permitirão agilizar o método básico. Ainda manteremos a hipótese de que podemos escrever diretamente na ata em memória secundária.

Uma das fontes de ineficiência do algoritmo anterior reside no fato de que uma determinada ação elementar é refeita, ou desfeita, independentemente da página correspondente já ter sido reenviada à memória secundária após esta ação ter sido invocada. Para obtermos uma idéia, do tipo de economia que se pode conseguir, examinemos a Figura 9.4, onde são indicados três casos possíveis. Para cada caso, estão assinaladas na ata as posições de todos os registros referentes a uma mesma transação *T* que modificou o conteúdo de uma página *P* do BD. O ponto marcado por *Y* mostra o último instante em que *P* veio para a memória principal.

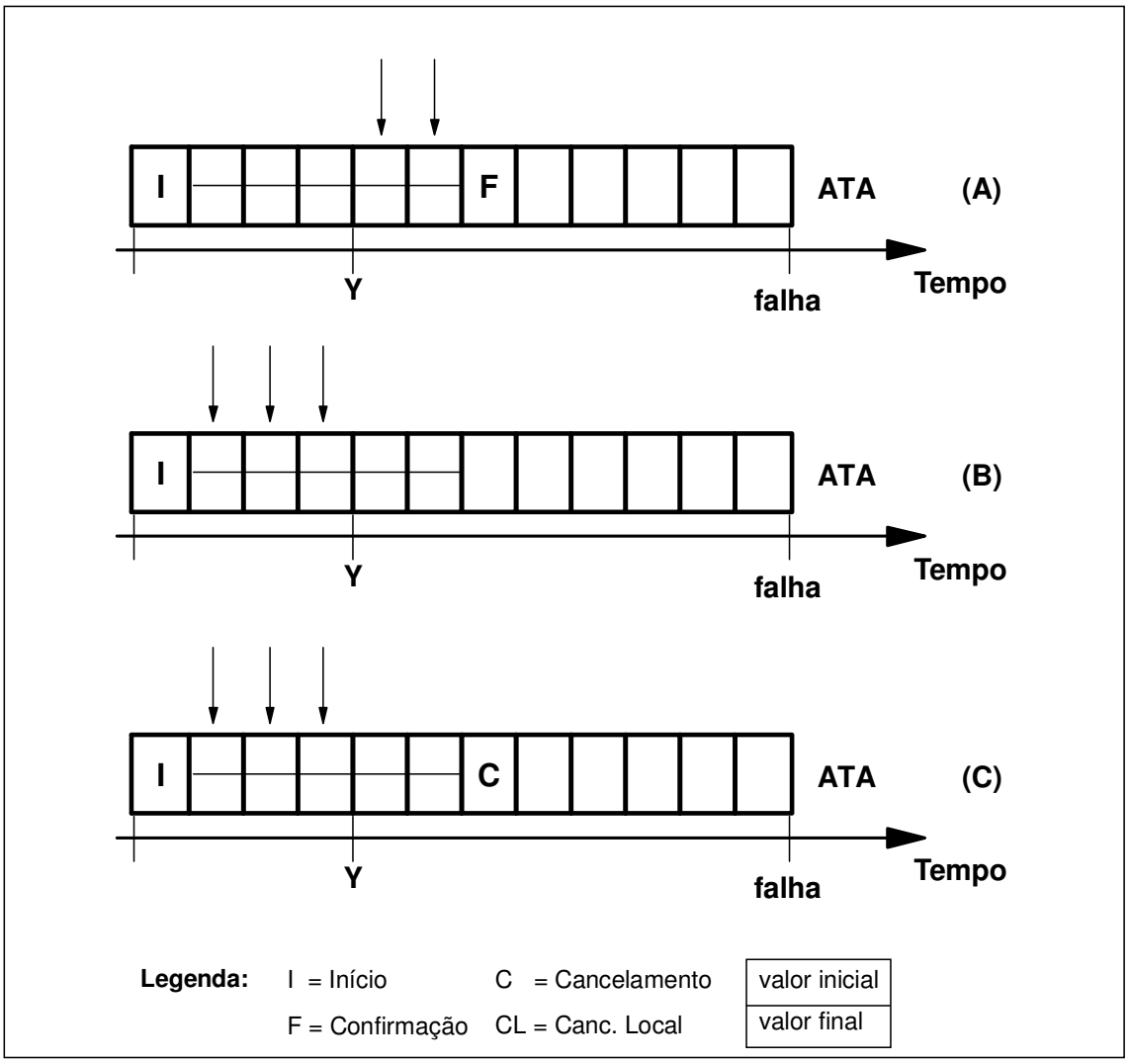

Figura 9.4 - Que Ações Desfazer/Refazer

No caso (a), *T* confirmou. À esquerda de *Y* está o registro de *INÍCIO* correspondente a *T*, seguido pelos registros de *EXECUÇÃO* de *T* e, à direita de *Y*, encontra-se o registro de *CONFIRMAÇÃO* de *T*. Neste caso, apenas as ações elementares de *T* cujos registros foram instalados na ata após o instante *Y* devem ter seus efeitos reinstalados no BD. São os registros assinalados na figura. Se *T* fosse classificada como confirmando em vez de confirmada, o esquema seria análogo, apenas que teríamos um registro tipo *PRONTO_SIM* ou *CONFIRMANDO* no lugar do registro de *CONFIRMAÇÃO*. O caso (b) indica uma transação incompleta. Neste último caso, todas as suas ações elementares correspondentes a registros de *T* que ocorrem após *Y* já foram desfeitos com a perda do conteúdo da memória principal. Logo, apenas os registros que ocorreram antes do ponto *Y* devem ser usados para desfazer efeitos já instalados no BD. Finalmente, a parte (c) ilustra uma transação cancelada. O raciocínio é análogo àquele para transações incompletas, ou seja, devemos desfazer os efeitos de ações cujos registros aparecem na ata antes do ponto *Y*.

Partimos agora para uma descrição mais precisa. Novos tipos de registros serão necessários e, também, alguns campos adicionais serão introduzidos em certos tipos de registros já definidos. Como ficou claro da discussão acima, temos que indicar não só os instantes em que páginas são trazidas para a área de trabalho, como também os instantes em que são enviadas para a memória secundária. Registros de tipos *LIDA* e *ESCRITA* vão assinalar estes pontos. Além disso, para melhorar a busca de registros na ata, estes formarão agora duas cadeias independentes. Uma delas ligará os registros que pertencem a uma mesma transação. Esta cadeia segue da direita para a esquerda, isto é, o ponteiro de um registro localiza a posição na ata do registro imediatamente anterior a este, referente a mesma transação. Os ponteiros que ligam os vários elementos desta cadeia serão chamados de *ponteiros de transação*. A segunda cadeia liga, de forma similar, todos os registros que afetam uma mesma página. Estes últimos serão conhecidos como *ponteiros de página*. Refletindo isto, os vários registros da ata podem ter, agora, os seguintes valores em seus campos de tipo

INÍCIO	*TÉRMINO*
CONFIRMAÇÃO	*CANCELAMENTO*
CANC_LOCAL	*EXECUÇÃO*
LIDA	*ESCRITA*

Um registro tipo *LIDA* ou *ESCRITA* é formado pelos campos de

tipo
identificação de página

Um registro tipo *INÍCIO, CONFIRMAÇÃO, CANCELAMENTO* ou *CANC_LOCAL* é formado pelos campos de

tipo
identificação de transação
ponteiro de transação
ponteiro de página

Um registro tipo *TÉRMINO* é formado pelos campos de

tipo
identificação de transação
ponteiro de transação
ponteiro de página
estado de retorno

Um registro tipo *EXECUÇÃO* é formado pelos campos de

tipo
identificação de transação
ponteiro de transação
ponteiro de página
endereço do objeto manipulado
valor inicial do objeto manipulado
valor final do objeto manipulado

Cada página do BD também conterá um ponteiro para a ata. Este ponteiro deverá indicar o último registro da ata que modificou valor de algum objeto desta página. Chamaremos este último ponteiro de *ponteiro de ata*. Antes de descrever o novo algoritmo, é preciso deixar claro quais os passos seguidos durante a fase de migração que terão influência no comportamento do algoritmo. Para este fim, as fases de início, migração e término de transações são revistas abaixo. Em cada caso são indicados os pontos que devem ser observados quando adotamos a nova estrutura de registros da ata. Também são descritas as condições que devem ser satisfeitas pelo gerente de paginação quando este instala uma página na área de trabalho da memória principal, ou remete-a à memória secundária.

QUANDO DO INÍCIO DA TRANSAÇÃO

1) o local de origem da transação inicia sua execução;
   a) o executor de transações instala na ata um registro tipo *INÍCIO* cujos campos de ponteiros de página e ponteiro de transação são nulos. O respectivo campo de identificação contém, claro, a identificação da transação;
   b) o executor de transações instala a transação em sua tabela de transações que estão ativas neste nó;
   c) o executor de transações também anota nesta tabela o endereço, na ata, onde foi instalado este registro de *INÍCIO*;
   d) o executor de transações instrui o SGBD local para que este crie um agente local para esta transação;
2) um nó remoto recebe uma mensagem requisitando trabalho em favor de uma transação
   a) o executor de transações verifica se a transação consta em sua tabela de transações ativas neste nó.
      i) em caso afirmativo, apenas instrui o SGBD local para criar outro agente local para esta transação.
      ii) em caso negativo, os mesmos passos previstos no ítem 1 são executados

QUANDO DA INVOCAÇÃO DE UMA AÇÃO ELEMENTAR

1) se a página que contém o objeto a ser modificado ainda não está na área de trabalho da memória principal, requisite-a ao sistema operacional local;
2) após satisfeita a requisição, a página é "fixada" na área de trabalho da memória principal;
3) a mudança desejada é efetivada nesta página;
4) lançando o registro na ata:
   a) um registro para ata, de tipo *EXECUÇÃO*, e construído na memória principal. Seus campos contêm
      i) identificação: recebe a identificação da transação;
      ii) endereço: recebe o endereço desta página seguido do endereço do objeto relativo a esta página;
      iii) valor inicial: conterá o valor do objeto antes de alterado pela ação em questão;

iv) valor final: conterá o valor do objeto depois de alterado por esta mesma ação elementar;

v) ponteiro de página: recebe o valor do ponteiro de ata presente nesta página;

vi) ponteiro da transação: recebe o endereço do último registro que foi instalado na ata em favor desta transação. Este endereço está em poder do executor de transações local.

b) registro assim construído é lançado na ata.

5) O valor do ponteiro de ata desta página que foi modificada recebe o endereço da ata onde este registro foi instalado;

6) em sua tabela de transações ativas neste nó, o executor de transações substitui o endereço relativo a esta transação pelo endereço onde este registro foi instalado na ata;

7) a condição de "fixada", relativa a esta página, é relaxada.

QUANDO DA CONFIRMAÇÃO DA TRANSAÇÃO

1) um registro tipo *CONFIRMAÇÃO* é lançado na ata. Os passos são os mesmos previstos no item 4 do grupo de ações referentes a execução de uma ação elementar, descritos acima. É claro que o tipo do registro será *CONFIRMAÇÃO* e os passos envolvidos na construção dos campos de valor inicial, valor final e endereço devem ser ignorados.

QUANDO DO CANCELAMENTO, GLOBAL OU LOCAL, DE UMA TRANSAÇÃO

1) obtenha do executor de transações o endereço, na ata, do último registro instalado até agora para esta transação. Este é o registro que o executor de transações mantinha, em sua tabela de transações ativas neste nó, junto a identidade desta transação. Seja *R* este endereço;

2) Desfazendo efeitos de ações elementares desta transação. Caminhe, a partir de *R*, através da cadeia formada pelos ponteiros de transação até encontrar um registro de tipo *INÍCIO*. Seja *Z* o próximo registro a ser examinado.

a) se este registro não é do tipo *EXECUÇÃO*, ignore-o;

b) caso contrário, seja P a página do BD indicada pelo ponteiro de página deste registro:

i) se *P* não está na área de trabalho da memória principal, requisite-a ao sistema operacional local;

ii) após satisfeita esta requisição, a página é "fixada" na área de trabalho da memória principal;

iii) desfaça o efeito desta ação elementar sobre esta página;

iv) o ponteiro de ata desta página deve receber o valor do ponteiro de página deste registro;

v) a condição de "fixada", relativa a esta página, é relaxada;

c) caminhe pelo ponteiro de transação deste registro. Seja *Z* o novo registro encontrado. Volte ao passo 2a.

3) Instalando um registro na ata. Também agora efetuamos as mesmas operações descritas sob o item 4 do grupo de ações expostas acima que cobrem os passos relativos a execução de ações elementares, ignorando-se os mesmos campos de valor inicial, valor final e endereço. O campo de tipo é preenchido com *CANC_LOCAL* ou *CANCELAMENTO* segundo estejamos, respectivamente, na presença de um cancelamento local imposto a este nó por agentes externos, ou estejamos a finalizar o protocolo bifásico, cancelando a transação globalmente.

QUANDO DA LEITURA DE PÁGINA PARA A ÁREA DE TRABALHO

1) gerente de paginação requisita a página ao sistema operacional local;
2) se a operação de leitura é bem sucedida, o gerente de paginação constrói um registro de tipo *LIDA*. Seu campo de identificação de página contém o endereço da página cuja leitura foi solicitada;
3) gerente de paginação lança o registro na ata.

QUANDO DA ESCRITA DE PÁGINA EM MEMÓRIA SECUNDÁRIA

1) gerente de paginação solicita ao sistema operacional local que esta página seja escrita na memória secundária;
2) se a operação é bem sucedida, o gerente de paginação constrói um registro de tipo *ESCRITA* em cujo campo de identificação de página coloca o endereço desta página;
3) gerente de paginação lança o registro na ata.

As operações acima são simples o suficiente para dispensarem maiores explicações no sentido de convencer o leitor de que constróem as cadeias desejadas. Alguns pontos, porém, devem ser observados. Em primeiro lugar, a operação de "fixar" a página na área de trabalho tem por objetivo proibir que o gerente de paginação, ao executar seu algoritmo de substituição de páginas, envie para memória secundária uma página que estava no processo de ter seu conteúdo modificado. Desta forma, evitamos que páginas inconsistentes sejam lançadas em memória secundária. Em segundo lugar, estamos adotando uma página como o objeto de granularidade básica para controle de integridade. Em outras palavras, a cadeia de ponteiros de páginas poderia ser refinada no sentido de que cada cadeia representasse ações que alteraram o valor de um simples objeto e não de uma página toda. Isto traz o inconveniente de que uma mesma página teria vários ponteiros para a ata, um para cada objeto que acomodasse. De outra forma, não teríamos acesso diretamente a partir da página modificada ao início da cadeia de registros relativos a cada objeto, individualmente. Além disto, sendo cada página a unidade básica usada pelo sistema operacional local para trafegar entre a memória principal e a memória secundária, ao tentarmos seguir a cadeia de um objeto particular estaríamos, na realidade, caminhando também pela cadeia de páginas correspondentes. Um terceiro ponto diz respeito a figura do executor de transações. Lembre que, sendo o banco distribuído, uma mesma transação pode dar origem, durante sua fase de migração, a vários agentes locais em um mesmo nó. Ao encadearem os registros de uma mesma transação, estes agentes devem ter um meio de se comunicar. Só desta forma um particular agente pode ter notícia de onde se encontra o último registro da ata instalado pelos demais, informação esta essencial se todos devem construir uma cadeia única

para os registros desta transação. Estando no controle de todos os agentes locais da transação, o executor de transações se apresenta como um veículo natural para distribuir esta informação. Um quarto aspecto diz respeito a operação de cancelar transações. Note que as ações invocadas para inverter efeitos já instalados no BD não geram qualquer registro que vai para a ata. Note também como os ponteiros de ata referentes a cada página onde os efeitos da transação são removidos regridem a seus antecessores até que, eventualmente, voltam ao estado em que estavam antes da transação ter sido iniciada. Também vale um comentário a respeito do protocolo bifásico para confirmar intenções. Quando um nó responde ao coordenador com a mensagem de *PREPARADO*, deve garantir que os efeitos locais da transação poderão ser confirmados ou cancelados de acordo com veredicto posterior que o coordenador enviará. Como estamos supondo que é permitido escrever diretamente na ata em memória secundária, esta garantia é automática, uma vez que a ata registra tudo que se passa com cada transação. Finalmente, observe o ritual seguido pelo gerente de paginação durante as operações de leitura e escrita de páginas. Ao executá-lo, este último garante que os intervalos em que cada página estava na área de trabalho estarão bem documentados na ata. Desnecessário dizer que, uma vez iniciadas as operações de escrita e leitura, o gerente de paginação proíbe que qualquer outro módulo tenha acesso a página em questão antes que o respectivo registro seja instalado na ata e a página liberada. Danos causados às páginas em memória secundária durante as operações de leitura e escrita configuram falhas secundárias e, portanto, estão fora da discussão, no momento.

De posse destas novas capacidades, podemos iniciar a exposição do algoritmo principal. Antes de descrevê-lo, porém, comentaremos a respeito das várias facetas e sutilezas afetas ao problema de efetivar-se o controle de integridade no presente cenário. A cada passo, a questão é analisada e as ações do algoritmo justificadas, preparando ao leitor para aceitar seu posterior detalhamento. A Figura 9.5 ilustra alguns dos casos possíveis. Os números entre *I* e o instante da falha, este último indicado por *F*, mostram a ordem temporal de ocorrência dos vários eventos. No instante *I* dispomos das seguintes informações:

1) a identidade de cada transação que estava classificada como incompleta;
2) a identidade de cada transação que havia cancelado localmente, porém não globalmente;
3) a identidade de todas as transações, junto com o respectivo estado de retorno, que estavam envolvidas na execução do protocolo bifásico;
4) o endereço de toda página que estava na área de trabalho da memória principal

O processo todo é, agora, precedido de uma nova fase preparatória. Em seguida embarcamos nas fases que já conhecemos, quais sejam: desfazer efeitos de ações elementares, refazer efeitos de ações elementares e, finalmente, instalar transações junto ao executor de transações local. Os parágrafos seguintes discutem cada uma destas fases em separado.

Iniciamos com a nova fase preparatória onde buscamos dois resultados:

1) obter as listas *TA*, *TC* e *TR*, definidas como na subseção anterior;
2) construir uma lista *PG* que reflita as páginas que estavam na área de trabalho da memória principal no instante da falha

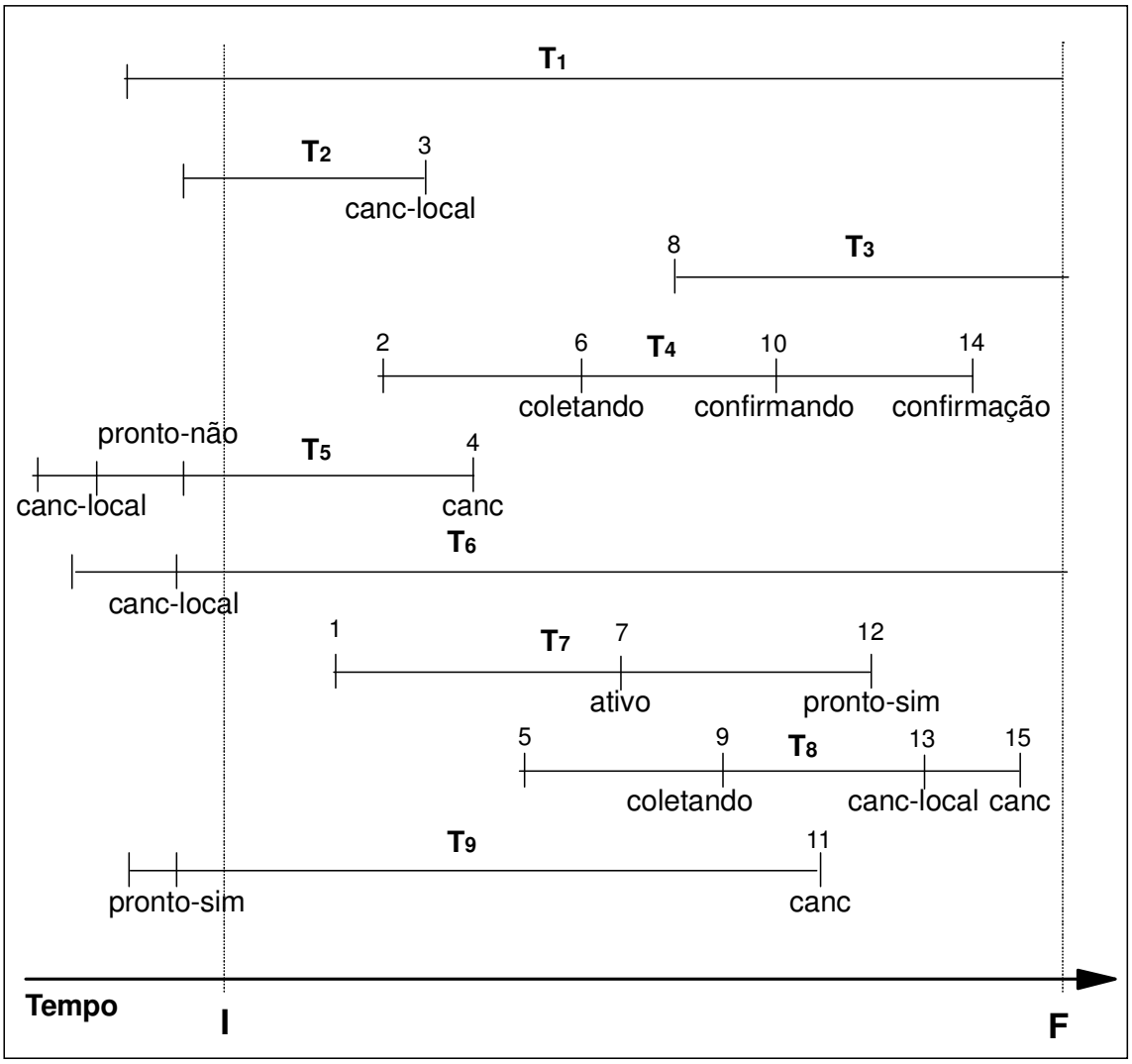

Figura 9.5 - Transações e o Instante da Falha

Nesta fase, as transações incompletas precisarão ser acumuladas em *TA*. No esquema da subseção anterior isto não era necessário, pois as transações incompletas eram detectadas usando-se o fato de que não pertenciam as listas *TA* ou *TC*. Para diferenciar este tipo de transação em *TA*, vamos marcá-las como "incompletas", em adição as marcas de "local" e "global" que já vínhamos usando.

O segundo item que desejamos construir nesta fase é muito fácil de ser obtido. Basta ler os registros da ata uma única vez, desde *I* até *F*. Durante esta mesma passada, podemos construir as listas *TA*, *TC* e *TR*. Note que agora estamos construindo as listas *TA*, *TC* e *TR* a medida que caminhamos na ata da esquerda para a direita. Para ilustrar o processo, tome a ref refid=9e.. Suponha que, de posse das informações iniciais a respeito das várias transações ativas no ponto *I*, as listas sejam iniciadas como segue:

$TA = \{ (T_1 , I) , (T_2 , I) , (T_5 , L) , (T_6 , L) \}$

$TC = \{ T_9 \}$

$TR = \{T_5 , T_9 \}$

Estamos adotando as seguintes simplificações de notação. Em relação aos itens da lista *TA*, as marcas de "incompleta", "local" e "global" serão indicadas por *I*, *L* e *G*, respectivamente. Dispensamos a indicação do estado de retorno nos elementos de *TR*, pois são irrelevantes nesta fase. Observe que, de início, não temos informação a respeito de quais transações confirmaram em pontos anteriores a *I*. Assim, a lista *TC* é iniciada apenas com transações que, no ponto *I*, pertencem a classe de transações confirmando.

Toda a mecânica desta fase se resume na idéia de que, à medida que formos encontrando os vários registros na ata, as listas mudam de "estado". Cada uma destas mudanças de "estado" acarretará alterações no conteúdo das três listas.

Os dois primeiros eventos a considerar estão indicados pelos pontos 1 e 2 na figura. Nestes instantes, as transações T_7 e T_4 iniciam suas execuções locais. Este fato é detectado quando os correspondentes registros tipo *INÍCIO* são encontrados na ata. A ação tomada é incluir ambas as transações em *TA*, marcadas como "incompletas", refletindo o novo "estado" das listas. Ficamos com

$TA = \{ (T_1 , I) , (T_2 , I) , (T_5 , L) , (T_6 , L) , (T_7 , I), (T_4 , I) \}$

$TC = \{ T_9 \}$

$TR = \{ T_5 , T_9 \}$

Note que, neste ponto da ata, T_4 e T_7 são classificadas como transações incompletas. Este fato é refletido na ação de colocá-las em *TA*. Em seguida, T_2 cancela localmente. Forçosamente, deve estar em *TA*. Em conseqüência, sua marca é mudada para "local". O próximo evento indica que T_5 cancela globalmente. Portanto estava em *TR* e ainda em *TA* ou *TC*. Basta retirar T_5 de *TR* e também de *TC*, se lá estiver. Em adição, T_5 deve ser colocada em *TA*. Se T_5 já estiver em *TA*, basta trocar sua marca para "global". Caso contrário, devemos inseri-la em *TA*, marcando-a como "global".

As lista agora tem a forma:

$TA = \{ (T_1 , I) , (T_2 , L) , (T_5 , G) , (T_6 , L) , (T_7 , I) , (T_4 , I) \}$

$TC = \{ T_9 \}$

$TR = \{ T_9 \}$

Na marca do ponto 5, T_8 é colocada em *TA*, marcada como "incompleta". Nos pontos 6 e 7, T_4 e T_7 são inseridas em *TR*, uma vez que lá não se encontravam. No ponto 8, T_3 é acrescentada a lista *TA* como "incompleta". Agora as listas seriam na forma:

$TA = \{ (T_1 , I) , (T_2 , L) , (T_5 , G) , (T_6 , L) , (T_7 , I) , (T_4 , I) , (T_3 , I) , (T_8 , I) \}$

$TC = \{ T_9 \}$

$TR = \{ T_9 , T_4 , T_7 \}$

No ponto 9, T_8 é inserida em *TR*. O registro no ponto 10 indica que o estado de retorno de T_4 deve ser alterado em *TR*. Mais ainda, T_4 deve agora ir para a lista *TC*. No ponto 11, T_9 é eliminada de *TC* e de *TR*, sendo colocada em *TA* como "global". Neste momento, teríamos :

$$TA=\{(T_1 , I) , (T_2 , L) , (T_5 , G) , (T_6 , L) , (T_7 , I) , (T_4 , I) , (T_3 , I) , (T_8 , I) , (T_9 ,G)\}$$
$$TC = \{ T_4 \}$$
$$TR = \{ T_9 , T_4 , T_7 , T_8 \}$$

Confiamos que o leitor possa efetuar as mudanças indicadas pelos pontos 12, 13, 14 e 15, obtendo o resultado final:

$$TA = \{ (T_1 , I) , (T_2 , L) , (T_5 , G) , (T_6 , L) , (T_3 , I) , (T_8 , G) , (T_9 , G) \}$$
$$TC = \{ T_4 , T_7 \}$$
$$TR = \{ T_7 \}$$

A fase seguinte deve remover os efeitos de todas as transações que no momento da falha eram classificadas como incompletas ou canceladas. Estas transações já foram identificadas na fase preparatória e estão acumulada na lista *TA*. Tudo que temos a fazer, portanto, é ler a ata do fim para o começo. Sempre que encontrarmos um registro tipo *EXECUÇÃO*, referente a uma transação que está em *TA*, os efeitos da correspondente ação elementar é invertido. O processo estará completo quando já analisamos os registros de todas as transações em *TA*. Para detectar esta condição, cada elemento de *TA* receberá uma segunda marca. Antes de iniciarmos, as transações em *TA* são marcadas "não_examinadas" sem prejuízo da marca que já detinham. Transações em *TA* são remarcadas "examinadas" quando atingimos os correspondentes registros tipo *INÍCIO*. Portanto, esta segunda fase estará completa quando todas as transações em *TA* estiverem marcadas "examinadas".

Vamos considerar, primeiro, o caso de transações que, como T_1 e T_8 na figura anterior, são classificadas como incompletas no instante da falha. Seja *T* uma transação deste tipo e seja *P* uma página qualquer do BD de tal sorte que *T* invocou ações que modificaram objetos em *P*. Tudo que podemos contar é com a cópia de *P* que residia na memória secundária no instante da falha e, por conseguinte, suporemos que *P* encarne esta cópia. Seja *Y* o endereço indicado pelo ponteiro de ata em *P*. Dependendo do instante em que *P* foi enviada para a memória secundária pela última vez, *Y* pode apontar para um ponto antes ou depois do marco *I*. A primeira situação é mostrada na Figura 9.6.

Os registros Z_1 a Z_7 representam os registros da ata que modificaram valores de objetos em *P*. O registro R_1 indica o início da transação *T*. A cadeia de ponteiros ligando os registros Z_i é aquela obtida a partir dos ponteiros de página. Formam, portanto, uma cadeia completa de registros que alteraram valores de objetos que residem em *P*.

Se assumirmos que durante a fase de execução não são bloqueados objetos internos a uma página então, devido ao principio de não interferência entre ações de transações que ainda não completaram, todos os registros entre Z_1 e Z_7 teriam sido gerados por *T*. Se este não for o caso, aqueles registros poderiam corresponder a várias transações. A cadeia de ponteiros de transação de *T* pode, evidentemente, passar por outros registros além daqueles indicados pelos Z_i's.

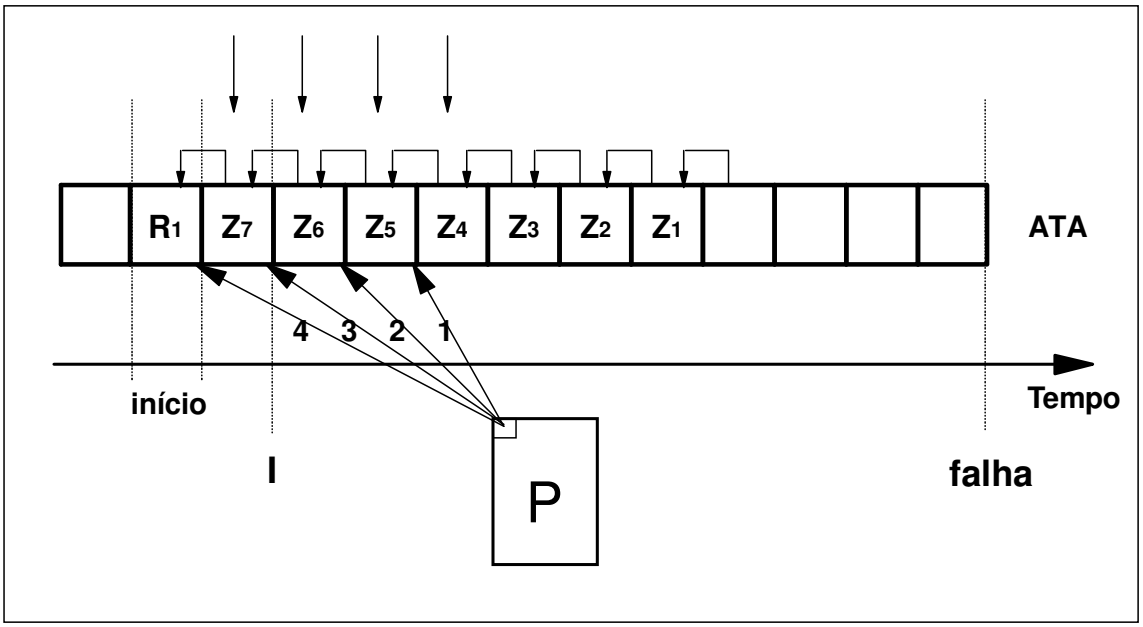

Figura 9.6 - Desfazendo Ações de uma Transação Incompleta

O valor inicial do ponteiro de ata de *P* aponta para Z_4. Isto significa que o gerente de paginação elegeu por reescrever *P* na memória secundária entre os instantes marcados por Z_4 e Z_3. Como a transação não chegou a completar, os efeitos de Z_1 a Z_3 foram automaticamente desfeitos quando o conteúdo da memória principal foi perdido. Os efeitos de Z_4 a Z_7, porém, têm que ser desfeitos pois já foram registrados na memória secundária. Este objetivo é fácil de ser alcançado percorrendo a ata da direita para a esquerda. Ao atingir Z_1, comparamos o valor do endereço deste registro com o ponteiro da ata de *P* e descobrimos que Z_1 está a direita daquele último. Isto indica que Z_1 só foi lançado na ata após a última vez que *P* foi reescrita na memória secundária. Podemos, portanto, ignorar Z_1 uma vez que a ação elementar que descreve já está, automaticamente, desfeita. Ignorando Z_1, passamos a analisar os registros entre Z_1 e Z_2. Estes dizem respeito a outras páginas, diferentes de *P*. Após completarmos suas análises, chegamos a Z_2. O mesmo ritual se repete, sendo Z_2 também ignorado. Prosseguindo, alcançamos Z_3, ignorando-o. Ao chegarmos a Z_4, porém, o ponteiro de ata de *P* indica o próprio Z_4. Neste caso, o efeito da ação correspondente a Z_4 é desfeito.

Em seguida, o ponteiro de ata de *P* é modificado para Z_5, o próximo registro que alterou valores em *P*, indicado pelo ponteiro de página do registro em Z_4. Isto é indicado pela seta na posição 2 na Figura 9.6. Após processar os registros entre Z_4 e Z_5, chegamos a este último. Os mesmos passos seriam repetidos, desfazendo-se os efeitos da ação correspondente a Z_5 e colocando-se o ponteiro de *P* a apontar para Z_6. Este último, e em seguida Z_7, sofreriam sorte igual, tendo os efeitos das respectivas ações elementares também removidos. Finalmente, quando alcançamos o ponto R_1, o registro de *INÍCIO* de *T* é analisado. Neste momento, o processamento relativo a transação *T* estaria completo. Em conseqüência, *T* é marcada como "examinada" em *TA*.

É claro que os registros que aparecem na ata entre aqueles mostrados na figura estariam guiando processos similares, dependendo do estado da respectiva transação no instante da falha. Observe também que a posição de *I* é imaterial para o desenrolar do algoritmo.

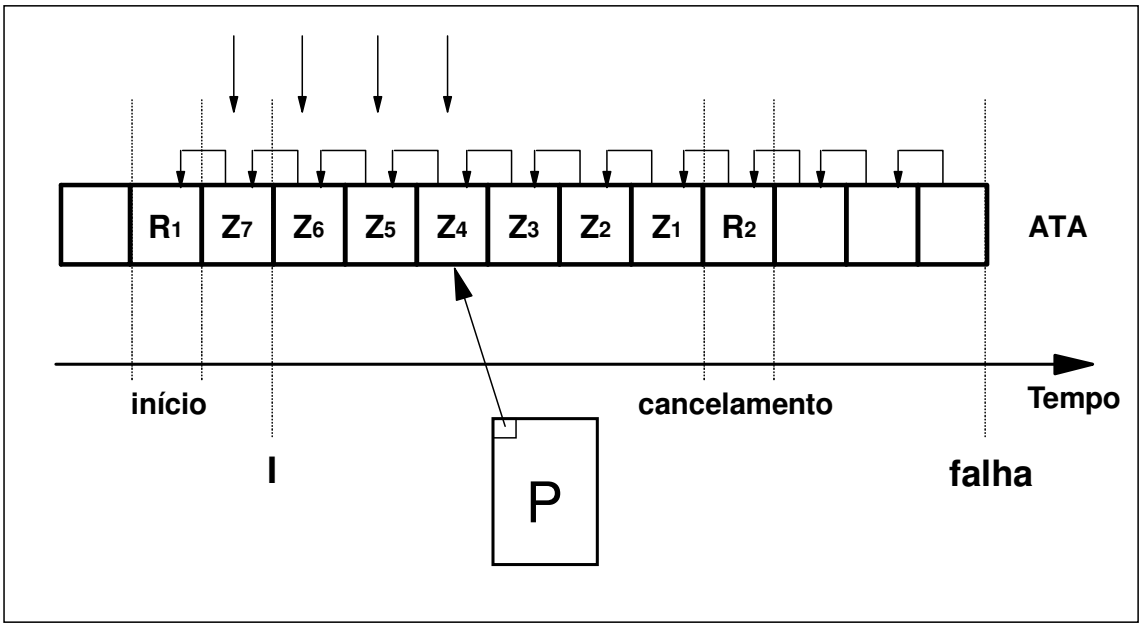

Figura 9.7 - Desfazendo Ações de uma Transação Cancelada - I

A mecânica usada para desfazer ações elementares consiste em substituirmos o valor do objeto presente em *P* por aquele indicado pelo campo de valor final do registro na ata. Em vista disto, poderíamos ser tentados a percorrer a ata e apenas inverter o efeito do último registro de cada transação. Note, porém, que registros diferentes podem estar se referindo a objetos de *P* que são distintos. Desta forma, não obteríamos o mesmo resultado se ignorássemos Z_4, Z_5 e Z_6, desfazendo apenas os efeitos relativos a ação de Z_7. Isto seria satisfatório unicamente no caso em que a noção de página e objeto do BD coincidissem. Mas, nesta hipótese, seríamos forçados a manter campos de valor inicial e valor final de tamanhos avantajados em todo registro tipo *EXECUÇÃO*. Sendo este tipo de registro o mais comum, o estratagema poderia custar bastante caro em termos de espaço disponível na ata. A alternativa, já mencionada, de termos um ponteiro de ata para cada objeto da página implica em menos espaço disponível para dados em cada página do BD, em troca de alguma economia durante a fase de recuperação de falhas. Em contrapartida, esperamos, falhas devem ocorrer apenas raramente. Em resumo, o esquema de um ponteiro por página parece acomodar uma boa solução de compromisso.

Ainda nesta fase de remover efeitos, vamos considerar o caso de transações canceladas, tanto local quanto globalmente. Não faremos distinção entre estes dois tipos de transações, pois ambas apresentam a característica comum de já terem tido seus efeitos removidos no passado. A Figura 9.7 repete o cenário para este tipo de transação. As únicas diferenças residem nos fatos de que há um registro de tipo *CANCELAMENTO*, ou *CANC_LOCAL*, indicado pelo ponto R_2 e podem haver, também, outros registros que manipulam objetos de *P* à direita de R_2. Estamos assumindo que *P* foi reescrita para a memória secundária antes de ser lançado em ata o registro de cancelamento da transação. Assim, o cenário ilustrado na última figura pode ter ocorrido de dois modos:

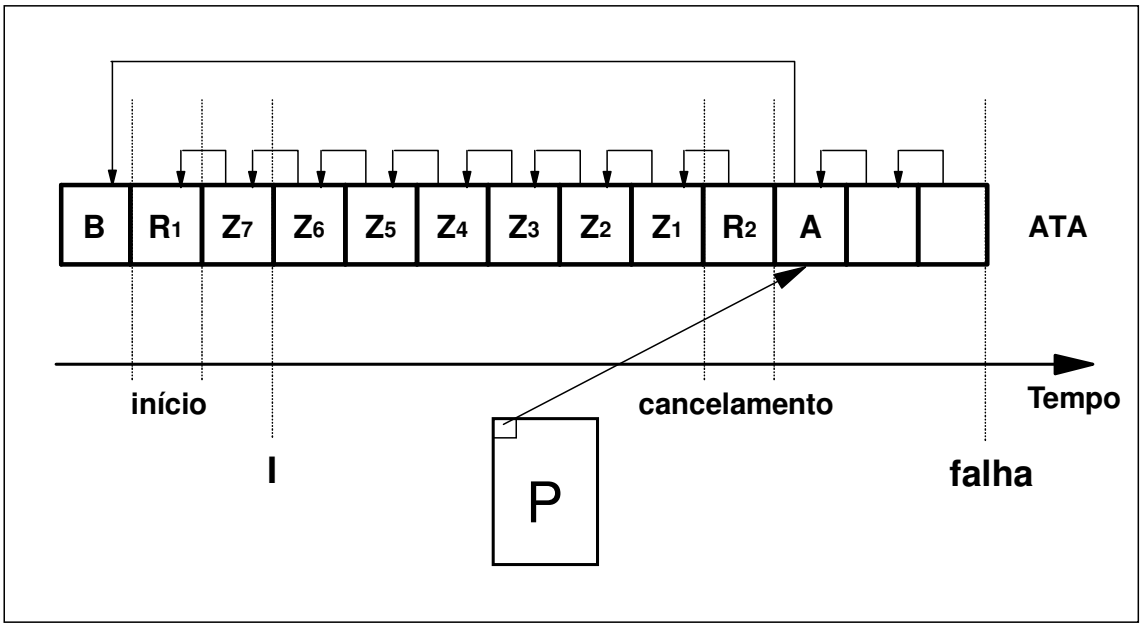

Figura 9.8 - Desfazendo Ações de uma Transação Cancelada - II

- gerente de paginação elegeu por reescrever *P* para a memória secundária em algum instante entre o lançamento na ata dos registros Z_4 e Z_3 isto é, antes que a decisão de cancelar *T* fosse tomada;
- o gerente de paginação resolveu reenviar *P* para a memória secundária após a decisão de cancelamento ter sido adotada, isto é, quando o agente local de *T* estava desfazendo os efeitos de suas ações elementares. Mais precisamente, no instante em que estávamos lendo a ata, da direita para a esquerda, entre os registros Z_3 e Z_4, no afã de percorrer a cadeia de ponteiros de transações de *T* e desfazer seus efeitos.

Em ambos os casos, o gerente de paginação enviou *P* para a memória secundária pela última vez antes do lançamento na ata do registro tipo *CANCELAMENTO* ou *CANC_LOCAL*. Na primeira hipótese, Z_1 a Z_3 caracterizariam modificações referentes a uma transação que foi cancelada e que não foram tornadas permanentes em memória secundária. Podem, portanto, ser ignoradas. Já Z_4 a Z_7 indicariam ações cujos efeitos teriam que ser removidos. Na segunda hipótese, temos situação semelhante. Apenas que o trabalho de instalar e remover os efeitos das ações indicadas por Z_1 a Z_3 foi efetuado em memória principal, sendo, portanto, desperdiçado. De qualquer forma, os efeitos dos registros correspondentes de Z_4 até Z_7 são os únicos que precisam ser desfeitos, exatamente como no caso de transações incompletas. O procedimento, portanto, é inteiramente análogo. Vale a pena, ainda com relação a transações canceladas, examinar a Figura 9.8 onde é ilustrada a situação que se configuraria quando o gerente de paginação envia *P* a memória secundária após o registro de tipo *CANCELAMENTO*, ou *CANC_LOCAL*, ter sido lançado na ata. Ao desfazermos os efeitos das ações referentes a *T*, os ponteiros de página regridem na ata, produzindo o efeito indicado na figura. Durante a fase de recuperação, caminhamos da direita para a esquerda pela cadeia de ponteiros de página relativos a *P*. Em conseqüência, passaremos do ponto indicado por *A* diretamente para aquele indicado por *B*. Isto acarretará com que todos os registros relativos a *T* sejam ignorados como era de se esperar, uma vez que a operação de remover efeitos teve chance de completar e passar a memória secundária.

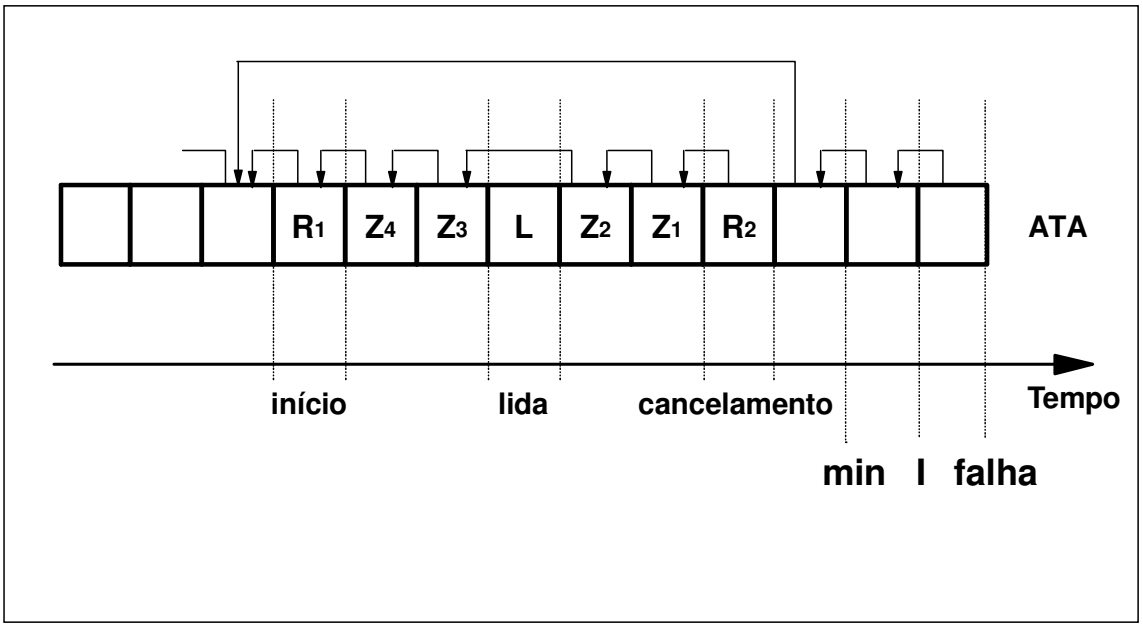

Figura 9.9 - Transações que cancelaram antes do balizador.

Observe que devemos analisar todos os registros da ata referentes a transações de *TA*. Assim procedendo, poderemos ser forçados a ler a ata até um ponto bem anterior aquele indicado por *I*. Este fato é evidente na Figura 9.5. Entretanto, os problemas desta fase não findam aqui. Para visualizar que tipo de problemas podem surgir, considere a lista *TA* obtida ao final da fase preparatória. Suponha que *MIN* indique a posição da ata onde se situa o registro mais antigo dentre todos aqueles registros referentes a transações presentes em *TA*. Lembre que o fim da segunda fase é indicado quando em *TA* não mais podemos encontrar transações marcadas "não_examinadas". Assim, *MIN* indica o último registro da ata que será analisado durante esta fase. Assuma que *P* é uma página do BD que foi lida para a memória principal em um momento *L*, anterior aquele indicado por *MIN*. Assuma também que *P* não foi subsequentemente reescrita em memória secundária. Seja *T* uma transação que cancelou antes de *I* e após *L*. Em conseqüência, se *T* invocou, antes de *L*, ações elementares que modificaram valores de objetos em *P* e estas ações terão que ser, portanto, invertidas. Mas *T*, tendo cancelado antes de *I*, não aparece em *TA*. Mais ainda, a segunda fase terminaria ao atingirmos o ponto *MIN*. Antes, portanto, de sequer atingirmos o ponto de cancelamento de *T* e, por razão mais forte, antes de invertermos os efeitos de qualquer de suas ações elementares.

Considere a Figura 9.9, onde *I*, *MIN* e o instante da falha estão claramente indicados. A marca *L* indica a posição do registro tipo *LIDA* mais antigo relativamente a todas as páginas que estavam na área de trabalho durante o instante da falha. Também estão marcadas as posições dos registros de uma transação *T* que iniciou e foi cancelada antes de *MIN*. Deveríamos parar de regredir na ata ao alcançarmos o ponto indicado por *MIN* e, portanto, as ações relativas aos registros Z_4 e Z_3 não teriam seus efeitos removidos. No entanto, *P* foi lida para a memória principal no instante indicado por *L* e nunca mais foi reescrita em memória secundária. O trabalho de desfazer os efeitos das ações relativas aqueles registros foi perdido quando o conteúdo de *P* ficou inutilizado pela falha. Sanar esta situação, porém, é fácil. Basta que continuemos a regredir na ata, mesmo após a lista *TA* só conter transações marcadas "examinadas", até que tenhamos alcançado o ponto

L. Assim procedendo, o registro R_2 indicando o cancelamento de *T* seria atingido antes de chegarmos a *L*. Neste instante, *T* seria adicionada a *TA* garantindo, desta forma, que iremos ler a ata até, pelo menos, o ponto indicado por R_1, ou seja, a posição na ata do primeiro registro referente a *T*.

Resumindo, a condição de parada para esta fase dita que devemos interromper o processo apenas no caso de:

1. *TA* não mais conter transações marcadas "não_examinadas" e
2. já termos passado o ponto onde a página mais antiga, dentre todas que estavam na área de trabalho, foi lida para a memória principal pela última vez antes da falha.

Observe que este processo pode nos levar a regredir na ata além do ponto *L*. Entretanto o processo não será cascateado indefinidamente. Uma vez que tenhamos passado pelo ponto *L*, não mais adicionaremos transações a *TA*. Apenas remarcaremos transações de "não_examinadas" para "examinadas". Desta forma, em algum ponto, chegaremos a situação onde *TA* contém apenas transações marcadas "examinadas", terminando a segunda fase.

Durante a terceira fase, devemos refazer efeitos de todas as transações que devem confirmar isto é, que estão na lista *TC*, já compilada na fase preparatória. A primeira pergunta que surge diz respeito ao ponto da ata onde devemos iniciar esta fase. Note que só são perdidos os conteúdos das páginas que estavam na área de trabalho quando do instante da falha. Assim, nada mais justo que iniciar a operação de refazer efeitos a partir do momento em que foi lida a página mais antiga que permanecia na área de trabalho no instante da falha. Mais ainda, podemos ignorar todos os registros de *EXECUÇÃO* que indiquem páginas que não estavam na área de trabalho neste instante. E as páginas que estavam na área de trabalho são conhecidas pois foram determinadas na fase preparatória. Todo efeito sobre páginas que residiam na memória secundária no instante da falha está automaticamente garantido.

Usando os mesmos personagens que das vezes anteriores, uma situação típica é mostrada na Figura 9.10. A única diferença está no fato de termos substituído a marca *I* pela marca *L*. Lembre que *L* indica onde se situa o registro tipo *LIDA* mais antigo relativamente a todas as páginas que estavam na área de trabalho no instante da falha. Na figura, a última vez que o gerente de paginação remeteu a página *P* para a memória secundária foi em algum ponto entre Z_4 e Z_3. Assim, os efeitos de Z_3 a Z_1 foram perdidos e devem ser recuperados. Já os efeitos de Z_7 a Z_4 foram conservados e não precisam ser refeitos. Lembre, outrossim, que nesta fase estamos lendo a ata da esquerda para a direita, em sentido inverso, portanto, que quando da fase anterior. Caminhando a partir de *L* em direção ao final da ata, ignoramos os registros Z_6 a Z_4. O primeiro registro que encontraremos e que não pode ser ignorado é Z_3. Esta condição é detectada quando o ponteiro de ata em *P* e o ponteiro de página do registro em exame apontam para o mesmo local da ata, no caso Z_4. O efeito da ação elementar correspondente é reinstalado em *P* e, em seguida, o ponteiro de ata de *P* é deslocado para este mesmo registro ora em consideração, Z_3 no caso. Isto é indicado na figura pela seta número 2. Prosseguindo, leríamos os registros entre Z_3 e Z_4 tomando as medidas cabíveis em cada caso. Quando chegássemos a Z_2, a situação anterior se repetiria. Os efeitos da ação indicada em Z_2 seriam refeitos e o ponteiro de página de *P* passaria a indicar Z_2. Em seguida, viria Z_1 e, finalmente, atingiríamos R_2 quando registros desta transação deixariam de aparecer na ata. Neste ponto, as ações elementares de *T* teriam sido reinstaladas no BD.

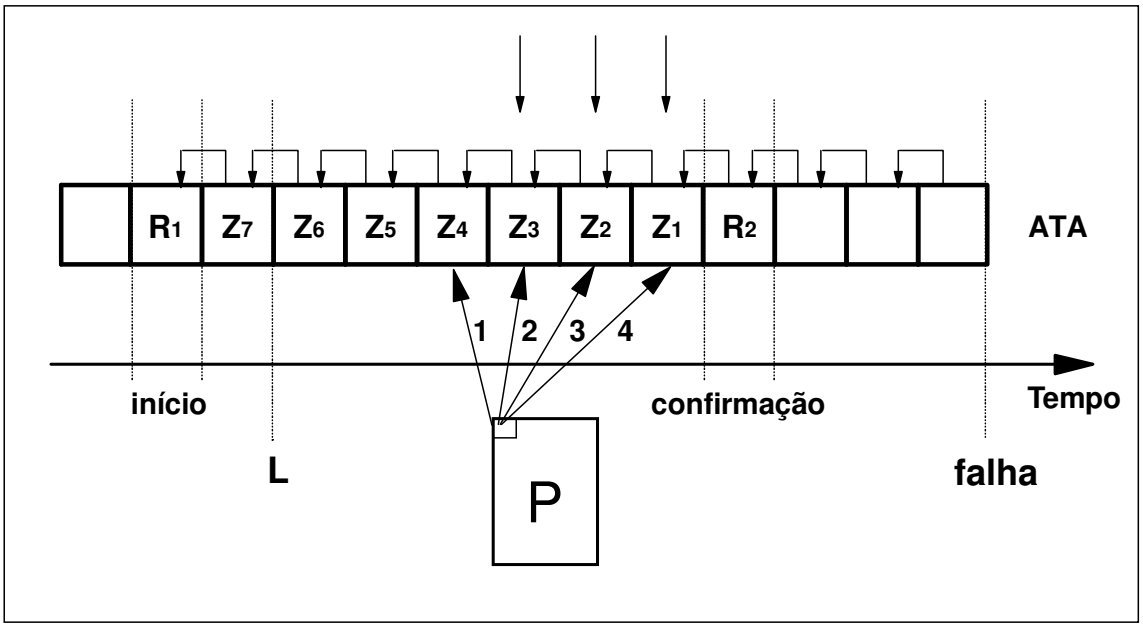

Figura 9.10 - Refazendo Ações de uma Transação Que Deve Confirmar

A fase final reinstala as transações de *TR* junto ao executor de transações local e não demanda maiores comentários.

Antes de detalhar a descrição do algoritmo, valem aqui algumas observações que permitirão agilizar ainda mais sua execução. Considere a fase onde são reinstalados os efeitos de transações que devem confirmar. Seja *T* uma destas transações. Suponha que estamos examinando um registro de *T* tipo *EXECUÇÃO* e cuja posição é indicada por *Z*. A página indicada por *Z*, seja *P*, deve estar na lista de páginas compiladas na primeira fase, do contrário o registro indicado por *Z* seria ignorado. De acordo com o que foi exposto anteriormente, precisamos comparar *Z* com o ponteiro de ata de *P*. Para este fim, se *P* não estiver neste momento na área de trabalho, deve ser obtida a partir da memória secundária. Podemos, em certos casos, economizar esta eventual operação de leitura. Seja *PG* a lista de páginas que estão na área de trabalho da memória principal no instante da falha. Assuma que, junto com a indicação de que *P* está em *PG*, disponhamos também de uma indicação, *X*, do local na ata onde se encontra o registro tipo *LIDA* referente a *P*. Comparamos *X* e *Z*. Se concluirmos que *Z* indica uma posição anterior àquela indicada por *X*, então o registro referente a *Z* pode ser sumariamente ignorado. Está claro que, quando ocorreu a falha, o efeito relativo a ação indicada por *Z* estava a salvo em memória secundária. Mais ainda, determinamos este fato sem apelar para o conteúdo de *P*, evitando uma possível leitura de página para a memória principal.

Reconsidere agora a segunda fase, quando devemos desfazer os efeitos das transações de *TA*. Vamos indicar em que situações poderemos dispensar leituras supérfluas de páginas do BD. Seja *T* uma transação de *TA* que seria classificada como incompleta no instante da falha. Ainda denotaremos por *Z* a posição de um registro de *T* e por *X* a posição do registro tipo *LIDA* referente a *P*. Se *X* aparecer aquém de *Z* na ata, então *P* não foi reescrita em memória secundária após ter seu conteúdo modificado pela ação elementar descrita pelo registro *Z*. Poderemos, portanto, ignorar *Z*. Por fim, assuma que *T* é uma das transações de *TA* que cancelou, local ou

globalmente, antes da ocorrência da falha. Lembre que, quando a decisão de cancelar *T* é tomada, também é feito um esforço no sentido de desfazer os efeitos de suas ações elementares. Seja *Y* a posição da ata onde se situa o registro tipo *CANCELAMENTO*, ou *CANC_LOCAL*, referente a *T*. Se *Y* estiver aquém de *X* na ata, então *P* foi reescrita na memória secundária após desfazermos todos os efeitos de *T*. Concluímos que *Z* pode, de novo, ser ignorado.

Como uma última observação, note que, para completar as três primeiras fases, percorremos a ata apenas três vezes. Na primeira fase percorremos a ata da esquerda para a direita desde *I* até seu final. A segunda fase lê a ata da direita para a esquerda começando por seu fim e prosseguindo até um certo ponto aquém do local indicado por *L*. A terceira e última fase que usa a ata, varre-a da esquerda para a direita. Começa no ponto onde a fase anterior parou e procura o ponto *L*. De *L*, vai até o final da ata. Este sincronismo de direções é especialmente útil se a ata, ou parte dela, residem em fita magnética.

Após estas considerações, podemos passar a descrição do algoritmo. Estamos assumindo que dispomos de:

1. um ponteiro para a ata, *I* ;
2. a identidade de cada transação que estava incompleta no ponto indicado por *I* ;
3. a identidade de cada transação que havia cancelado localmente, porém não globalmente, no instante mostrado por *I*. Além disto, sabemos em que posição da ata está o registro de *CANC_LOCAL* de cada uma destas transações;
4. a identidade de todas as transações, junto com o respectivo estado de retorno, que estavam envolvidas na execução do protocolo bifásico, no momento apontado por *I* ;
5. o endereço de toda página que estava na área de trabalho da memória principal, no instante indicado por *I*. Também conhecemos a posição na ata onde está o registro tipo *LIDA* para cada uma destas páginas.

Os elementos na lista *TA* poderão receber dois tipos de marcas: "incompleta"/"local"/"global" ou "examinada"/"não_examinada". Os elementos da lista *PG* terão associado a si um ponteiro para a ata. O mesmo pode acontecer para os elementos de *TA*. No caso de falha primária os seguintes passos são executados

FASE PREPARATÓRIA:

1) a partir das informações presentes em *I*, inicie as listas *TA, TC, TR e PG* ;
2) marque toda transação em *TA* como "incompleta" ;
3) partindo do ponto indicado por *I*, examine cada registro até o fim da ata. Seja *R* o próximo registro a ser examinado. Se o tipo de *R* for
   a) *INÍCIO*:
      i) adicione esta transação a lista *TA*, marcando-a como "incompleta" ;
   b) *CONFIRMAÇÃO*:
      i) retire esta transação da lista *TR* ;
      ii) coloque esta transação na lista *TC* ;

c) *CANCELAMENTO*:

i) retire esta transação de *TR* ;

ii) se esta transação está em *TC*, retire-a desta lista ;

iii) se esta transação

(1) está em *TA*, remarque-a como "global" ;

(2) está em *TA*, adicione-a a *TA* e marque-a como "global" ;

(3) associe a mesma um ponteiro para a presente posição da ata ;

d) *CANC_LOCAL*:

i) remarque esta transação em *TA* como "local" ;

ii) associe a esta transação um ponteiro para esta posição da ata ;

e) *TÉRMINO*:

i) se o estado de retorno de *R* for *PRONTO_SIM* ou *CONFIRMANDO*, inclua esta transação em *TC* junto com o respectivo estado de retorno;

ii) em qualquer outro caso:

(1) se esta transação não está em *TR*, deve ser lá incluída, junto com o estado de retorno indicado no registro *R* ;

(2) se esta transação já está em *TR*, deve ter seu estado de retorno substituído por aquele indicado por *R* ;

f) *LIDA:*

i) inclua esta página em *PG*, associando a esta página a posição do registro *R* ;

g) *ESCRITA:*

i) retire esta página da lista *PG* ;

h) *EXECUÇÃO:*

i) ignore este registro

DESFAZENDO EFEITOS DE AÇÕES ELEMENTARES:

1) marque cada transação em *TA* como "não_examinada" ;

2) seja *L* o ponto da ata onde se situa o registro tipo *LIDA* mais antigo relativamente a todas as páginas em *PG* ;

3) partindo do fim da ata, caminhe em direção ao começo da mesma, examinando cada registro que encontrar. Prossiga até que todas as transações em *TA* estejam marcadas "examinada", e já tenha passado o ponto indicado por *L.* Seja *Z* o próximo registro a ser examinado, seja *T* a transação a que se refere o registro *Z* e seja P a página a que se refere a ação indicada pelo registro *Z*. Se o tipo de *Z* for:

a) *CONFIRMAÇÃO, TÉRMINO, LIDA* ou *ESCRITA*:

i) ignore este registro;

b) *INÍCIO*:

i) se esta transação estiver em *TA*, remarque-a como "examinada" ;

c) *CANCELAMENTO:*

i) se já tivermos passado o ponto indicado por *I* e ainda não atingimos o ponto indicado por *L*, então adicione esta transação a *TA*, marcando-a como "global" e "não_examinada" ;

ii) em *TA*, associe o ponto da ata indicado pelo registro *Z* a esta transação ;

d) *CANC_LOCAL*:

i) se já tivermos passado o ponto indicado por *I* e ainda não atingimos o ponto indicado por *L*, então adicione esta transação a *TA*, marcando-a como "local" e "não_examinada" ;

ii) em *TA*, associe o ponto da ata indicado pelo registro *Z* a esta transação

e) *EXECUÇÃO*:

i) se *T* não está em *TA*, ignore *Z* ;

ii) ignore *Z* se *T* está em *TA* e *T* não está marcada como "incompleta" e *P* não está em *PG* ;

iii) se *P* está em *PG*, seja *X* o ponteiro associado a *P* em *PG*. Ignore este registro se *T* está em *TA* e

(1) *T* está marcada "incompleta e *X* indica uma posição anterior aquela indicada por *Z* ou

(2) *T* não está marcada "incompleta" e *Y* indica uma posição anterior aquela indicada por *X*, onde *Y* é o ponteiro associado a *T* em *TA*

iv) se este registro não foi ignorado e *P* não está na área de trabalho, então leia *P* para a memória principal. Seja *W* o valor do ponteiro de ata de *P* :

(1) se *W* indicar uma posição anterior aquela assinalada por *Z*, ignore este registro;

(2) caso contrário:

(a) desfaça em *P* os efeitos da ação elementar contida em *Z*

(b) substitua em *P* o valor do ponteiro de ata por aquele indicado pelo ponteiro de página presente no registro *Z* .

REFAZENDO EFEITOS DE AÇÕES ELEMENTARES:

1) A partir de *L*, caminhe em direção ao fim da ata examinando cada registro que encontrar. Seja *Z* o próximo registro a ser examinado, seja *T* a transação a que se refere o registro *Z* e seja *P* a página a que se refere a ação indicada pelo registro *Z*. Se o tipo de *Z* for

a) *INÍCIO, CONFIRMAÇÃO, CANCELAMENTO, CANC_LOCAL, TÉRMINO, LIDA, ESCRITA:* ignore este registro ;

b) *EXECUÇÃO*:

i) se *T* não está em *TC* ou *P* não está em *PG* então ignore este registro ;

ii) caso contrário, seja *X* a posição da ata associada a P em *PG* :

(1) se *Z* indicar uma posição de ata anterior a *X*, então ignore este registro ;

(2) caso contrário, seja *W* a posição da ata indicada pelo ponteiro de ata de *P*. Prossiga como segue:

(a) se *W* indicar uma posição da ata anterior aquela assinalada pelo ponteiro de página de *Z*, então ignore este registro ;

(b) se *W* indicar uma posição da ata idêntica aquela assinalada pelo ponteiro de página de *Z*, então :

(i) reinstale em *P* os efeitos da ação elementar indicada em *Z* ;

(ii) substitua o valor do ponteiro de ata de *P* pelo valor de *Z* ;

INSTALANDO TRANSAÇÕES JUNTO AO EXECUTOR DE TRANSAÇÕES LOCAL:

1) para cada transação *T* da lista *TA* :
   a) se *T* está marcada como "global", ignore *T* ;
   b) se *T* está marcada como "incompleta" :
      i) envie uma mensagem de resposta ao no coordenador de *T* com campo de veredicto contendo *CANCELADO* ;
      ii) crie um registro tipo *CANC_LOCAL* para *T* e instale-o no final da ata ;
      iii) instale *T* na tabela do executor de transações local com a ressalva de que *T* cancelou localmente ;
2) para cada transação *T* da lista *TR* :
   a) instale *T* na tabela do executor de transações local indicando seu estado de retorno :
   b) informe ao executor de transações local para reiniciar a execução do protocolo bifásico em favor de *T*, partindo deste estado.

Os passos do algoritmo seguem fielmente a discussão anterior e, portanto, já estão devidamente comentados.

9.1.4 O Mecanismo Completo

Na subseção anterior, fizemos duas hipóteses que serão agora removidas. Em primeiro lugar, devemos relaxar a suposição irreal de que podemos inserir registros na ata *diretamente* em memória secundária, isto é, sem que a respectiva página seja trazida para a área de trabalho da memória principal. Em segundo lugar, devemos indicar como podemos obter informação suficiente para iniciar a fase preparatória. Isto pressupõe conhecimento do ponto *I*, como discutido na seção anterior. Mais ainda, devemos obter também as demais informações necessárias no que diz respeito ao estado de transações e páginas no ponto *I*. Uma vez que estas duas hipóteses sejam relaxadas, o processo se tornaria autônomo. Finalizaremos esta subseção com alguns comentários à cerca de falhas secundárias.

9.1.4.1 Páginas da Ata em Memória Principal

Vamos agora relaxar a hipótese de que podemos escrever na ata, diretamente em memória secundária. O perigo, naturalmente, está em que parte da área de trabalho será ocupada pelas páginas mais recentes da ata. Uma falha primária no sistema destruiria, portanto, informação que faz parte da ata. Entretanto, não há como evitarmos a perda do conteúdo da memória principal. Resta-nos uma única saída: evitar que sejam destruídas aquelas informações que são vitais para o processo de recuperação.

Para perceber claramente que tipo de informação é crucial ou irrelevante no momento de recuperação, vamos analisar o caso, típico, de uma transação que seria classificada como incompleta quando o sistema falhou. Será necessário desfazer os efeitos das ações elementares que já foram invocadas em benefício desta transação. Antes, isto era sempre possível pois a ata estava a salvo em memória secundária. Agora, não podemos ter como garantido que as informações da ata de que necessitaremos estarão disponíveis.

Seja *T* uma transação nestas condições e considere seguinte seqüência de ações

1) *T* é iniciada ;
2) ações elementares de *T* modificam o conteúdo de uma página de memória, *P*. Uma outra página, *Q*, recebe os registros referentes as mudanças efetivadas por *T* em *P* ;
3) o gerente de paginação decide reenviar *P* a memória secundária.
4) o sistema falha.

Como *T* é uma transação incompleta e a cópia da página *P* já foi enviada à memória secundária, as modificações instaladas por *T* em *P* devem ser removidas. Para efetivar esta mudança precisamos ler a versão da página *Q* onde estão os registros necessários para desfazermos os efeitos de *T* sobre *P*. Acontece que o gerente de paginação local não reinstalou *Q* na memória secundária antes da falha e, portanto, a versão de *Q* que lá reside está desatualizada. Em particular, os registros referentes a *T* de que necessitamos foram perdidos. Neste ponto não poderemos mais recuperar a consistência do BD.

Devemos, portanto, impedir que *páginas do BD* sejam enviadas a memória secundária *antes* de para lá enviarmos as *páginas da ata* que contêm os registros de modificações que afetam o conteúdo destas mesmas páginas do BD. É neste ponto que os mecanismos de controle de integridade usarão da prerrogativa de *forçar* a escrita de páginas em memória secundária, à revelia do algoritmo de substituição de páginas usado pelo gerente de paginação. O ponto crucial a observar é que devemos garantir que todos os registros da ata, referentes a mudanças no conteúdo de uma particular página do BD, estejam a salvo em memória secundária *antes* de enviarmos para lá esta página do BD. Note que não há mal algum em forçarmos certas páginas da ata, como *Q* na discussão acima, para a memória secundária e o sistema venha a falhar antes de serem escritas as correspondentes páginas do BD, como *P*. Neste caso, ao se recuperar, o sistema simplesmente inverteria algumas ações elementares registradas em *Q* que nem chegaram a fazer parte do BD em memória secundária, pois o conteúdo mais recente de *P* onde foram instaladas estas modificações é destruído pela falha. A rigor, não seria necessário desfazer tais efeitos que nem chagaram a ser salvos. Por outro lado, não causa qualquer inconveniência se assim o fizermos.

Considere agora o caso de uma transação que cancelou antes do instante em que ocorreu a falha. O problema é o mesmo do caso anterior, ou seja, como garantir que informações essenciais não se percam com as páginas da ata que estão na área de trabalho no momento da falha. Por um raciocínio análogo, será suficiente que tenhamos o cuidado de forçar para memória secundária todos os registros anteriores desta transação, *antes* de lançarmos na ata o registro de tipo *CANCELAMENTO*, ou *CANC_LOCAL*, da correspondente transação, Assim procedendo, garantimos que estas informações poderão ser recuperadas na eventualidade de falhas. Se o sistema falha quando estamos forçando os registros da ata para a memória secundária, a transação é tratada como incompleta pelo processo de recuperação, sem problemas.

No caso de uma transação que já confirmou, devemos refazer os efeitos de suas ações elementares. Agora devemos garantir que todos os registros relativos a esta transação estão a salvo em memória secundária *antes* de lançarmos na ata o *registro de tipo CONFIRMAÇÃO* relativo a esta transação. Assim, sempre poderemos refazer seus efeitos após uma falha primária. De novo não há dano se uma falha colhe o sistema no ato de forçar estes registros para a memória secundária e antes de lançarmos o registro de tipo *CONFIRMAÇÃO*. Se esta eventualidade ocorrer, o sistema tratará esta transação como incompleta ao se recuperar. Como já vínhamos observando o protocolo de salvar na ata todos os registros referentes a mudanças em qualquer página do BD que é enviada a memória secundária, sempre poderemos desfazer os efeitos de todas as ações elementares desta transação, agora tratada como incompleta, que porventura já tenham sido instalados em memória secundária pelo gerente de paginação. É esta ação, de resguardar os registros essenciais desta transação, que cada nó remoto deve efetuar ao receber a mensagem de *PREPARE-SE* vinda do coordenador durante e execução do protocolo bifásico para confirmar intenções invocado quando do término da transação. Só no caso de conseguir levar a cabo com sucesso esta operação é que o nó remoto deve responder *PREPARADO* e lançar na ata o registro correspondente, no caso *PRONTO_SIM*. Desta forma, estará apto a confirmar ou cancelar os efeitos da transação em qualquer circunstância futura, dependendo do veredicto final do coordenador. No caso de falhar a operação de resguardar os registros na ata, relembrando, o nó remoto enviaria mensagem de *IMPOSSIBILITADO*, registrando na ata *PRONTO_NÃO*. Observe que é necessário forçar apenas as *páginas da ata* e não as *páginas do BD* que são tocadas pela transação. Estas últimas poderão sempre ser recuperadas se garantirmos a saúde da ata a cada instante.

Forçar registros da ata para a memória secundária é uma tentativa de nos aproximarmos da condição ideal onde podemos escrever na ata diretamente em memória secundária.

Em resumo, o gerente de paginação e o gerente de transações interagem de forma tal que

1) antes que o gerente de paginação remeta uma página do BD para a memória secundária, todos os registros da ata que modificaram valores de objetos contidos nesta página devem seguir para a memória secundária;

2) antes que seja lançado na ata um registro tipo *CONFIRMAÇÃO, CANCELAMENTO* ou *CANC_LOCAL* para uma dada transação, todos os registros anteriores da ata, referentes a esta mesma transação, devem ser forçados a seguirem para a memória secundária.

A seguir vamos indicar as mudanças que devem ser acomodadas quando do início, migração e término de transações, bem como nas rotinas de leitura e escrita de páginas. O algoritmo básico

para efetivar o controle de integridade, apresentado na subseção anterior, continua valendo. Apenas vamos, agora, garantir que as informações da ata de que necessita estarão realmente disponíveis conforme suposto.

INÍCIO DE TRANSAÇÃO:

1) inalterado em relação ao processo descrito anteriormente;

EXECUÇÃO DE TRANSAÇÃO:

1) inalterado em relação ao processo descrito anteriormente;

CONFIRMAÇÃO DE TRANSAÇÃO

1) todas as páginas da área de trabalho que contém registos relativos a esta transação e que ainda não foram salvas na memória secundária devem ser forçadas para a memória secundária neste ponto;
2) um registro, tipo *CONFIRMAÇÃO*, é construído para esta transação em memória principal pelo seu agente local. Os campos deste registro contêm:
   a) identificação: recebe a identificação da transação,
   b) endereço: ignore
   c) valor inicial: ignore
   d) valor final: ignore
   e) ponteiro de página: é colocado o valor do ponteiro da ata presente nesta página
   f) ponteiro da transação: é inserido o endereço do último registro que o executor de transações instalou na ata em favor desta transação;
3) este registro é lançado na ata

CANCELAMENTO DE TRANSAÇÃO

1) obtenha do executor de transações o endereço da ata do último registro instalado até agora para esta transação. Seja *Z* este endereço. Caminhando através da cadeia formada pelos ponteiros de transação, a partir de *Z* e até encontrar um registro de tipo *INÍCIO* ;
   a) se a página que contém o objeto a ser modificado ainda não esta na área de trabalho da memória principal, requisite-a ao sistema operacional local.
   b) desfaça o efeito desta ação elementar;
   c) o ponteiro da ata desta página deve receber o valor do ponteiro de página deste registro;
   d) caminhe com *Z* pelo ponteiro de transação deste registro;
2) todas as páginas da área de trabalho que contém registos relativos a esta transação e que ainda não foram salvas na memória secundária devem ser forçadas para a memória secundária neste ponto.
3) um registro tipo *CANCELAMENTO*, ou *CANC_LOCAL*, é construído para esta transação em

memória principal pelo seu agente local. Os campos deste registro contêm:

a) identificação: recebe a identificação da transação,
b) endereço: ignore
c) valor inicial: ignore
d) valor final: ignore
e) ponteiro de página: é colocado o valor do ponteiro da ata presente nesta página;
f) ponteiro da transação: é inserido o endereço do último registro que o executor de transações instalou na ata em favor desta transação;

4) este registro é lançado na ata.

LEITURA DE PÁGINA PARA A ÁREA DE TRABALHO DA MEMÓRIA PRINCIPAL

1) gerente de paginação constrói um registro de tipo *LIDA*. Seu campo de identificação de página contém o endereço da página cuja leitura foi solicitada;
2) o gerente de paginação requisita a página ao sistema operacional local;
3) se a operação de leitura e bem sucedida, o gerente de paginação lança o registro na ata;
4) o gerente de paginação força este registro para a ata em memória secundária.

ESCRITA DE PÁGINA PARA A MEMÓRIA SECUNDÁRIA

1) gerente de paginação constrói um registro de tipo *ESCRITA* em cujo campo de identificação de página coloca o endereço desta página;
2) todas as páginas da ata que contém registros referentes a mudanças efetuadas em valores de objetos do BD que residem nesta página são forçadas para a memória secundária;
3) o gerente de paginação solicita ao sistema operacional local que esta página seja escrita na memória secundária;
4) se a operação é bem sucedida, o gerente de paginação instala na ata o registro tipo *ESCRITA* que acabou de construir.

Poderíamos, se assim o desejássemos, forçar registros da ata para a memória secundária após cada movimento. Este esquema seria o que melhor aproximaria a condição ideal de escrever diretamente na ata em memória secundária. O custo seria um elevado índice de tráfego de páginas da memória secundária para a memória principal e vice-versa. Este fato poderia degradar significativamente o temo de resposta do sistema. O que se tentou atingir foi uma solução de compromisso onde este tipo de tráfego é minimizado garantindo, porém, que a cada instante disporemos sempre das informações necessárias para recuperar o sistema em caso de falha. Dentro deste espírito, não forçamos os registros de tipo *INÍCIO* e *EXECUÇÃO* para a memória secundária. Na eventualidade de uma falha colher o sistema antes que este registros apareçam na ata em memória secundária, simplesmente a transação não teria existido no que diz respeito ao mecanismo de controle de integridade. Observe, porém, que a rotina que deve ser seguida pelo gerenciador de paginação, ao substituir páginas da área de trabalho, garante que não existirão registros tipo *EXECUÇÃO* sem que o correspondente registro tipo *INÍCIO*, bem como todos os

registros tipo *EXECUÇÃO* referentes a ações elementares anteriores, estejam devidamente protegidos na ata que reside em memória secundária. O mesmo raciocínio vale quando registramos em memória secundária os efeitos de uma transação ao lá instalarmos a correspondente página do BD: a rotina a ser seguida pelo gerente de paginação garante que todos os registros da ata que modificaram valores de objetos residentes nesta página vão para a memória secundária antes de para lá enviarmos esta particular página do BD.

Com relação a registros tipo *LIDA*, devemos assumir uma solução de compromisso. Forçar o registro para a memória secundária traz a vantagem de podermos corretamente construir, durante a fase preparatória, a lista *PG* de páginas que estavam na área de trabalho no instante da falha. No entanto, este procedimento implicará em forçar páginas da ata para memória secundária sempre que o gerente de paginação resolver ler ou escrever uma página do BD para memória secundária aumentando, assim, o trânsito entre a memória secundária e a memória principal. A alternativa seria *não forçarmos* os registros de tipo *LIDA* para a memória secundária. Neste cenário, na fase 2 simplesmente supomos, conservativamente, que *todas* as páginas do BD estavam na área de trabalho durante a falha. Este proceder, embora implicando em eventual trabalho desnecessário, garantiria a correção do algoritmo, conforme discutido acima. Mais ainda, estaríamos degradando o tempo de resposta durante a recuperação que, esperamos, seja um procedimento raro, em troca de melhorias durante a operação normal do sistema.

O esquema postulado acima pressupõe que sejamos capazes de identificar que páginas da ata ora residindo na memória principal já foram, alguma vez, enviadas a memória secundária. Se não formos capazes disto, a única alternativa seria reescrever em memória secundária, sempre que preciso, todas as páginas da ata que estão na área de trabalho. Identificar estas páginas, porém, não deve apresentar maiores dificuldades.

Considere primeiro o caso quando forçamos páginas da ata para a memória secundária antes de lá lançar registros de *CANCELAMENTO, CANC_LOCAL* ou *CONFIRMAÇÃO*. Todas as páginas da ata que foram tocadas pela transação estão encadeadas através do ponteiro de transação, acessível a partir do correspondente registro em poder do executor de transações. Tudo que temos a fazer é percorrer esta cadeia enviando para a memória secundária cada página visitada. De maneira análoga, toda a seqüência de registros da ata que modificaram valores de uma determinada página do BD é encadeada através dos ponteiros de página destes registros. Desta forma, forçar para a memória secundária as páginas da ata que tocaram certa página do BD corresponde a seguir esta cadeia, processando cada página encontrada. Note que o ponto inicial da cadeia está disponível na própria página em questão, através de seu ponteiro de ata.

É importante ressaltar que páginas da ata podem ser lidas para a memória principal de maneira não ordenada, isto é, as páginas da ata que estão na área de trabalho não necessariamente formam um bloco consecutivo de páginas da ata. Lembre que, durante o cancelamento de transações, devemos consultar as páginas da ata que foram afetadas por ações elementares de cada transação cancelada. Em conseqüência, não podemos assumir que as páginas da ata na área de trabalho obedeçam qualquer ordem pré-especificada. É possível, portanto, que uma determinada cadeia de ponteiros envolvendo registros da ata contenha nós intermediários cujas páginas já residiam em memória secundária. Ao percorrermos esta cadeia, seríamos forçados a consultar estas páginas da ata. Isto acarretaria o inconveniente de *ler* páginas para a memória principal, único ponto onde podemos manipulá-las. Este aumento de tráfego entre a memória secundária e a memória principal traz todos os inconvenientes já conhecidos. Entretanto, estas dificuldades podem ser

facilmente contornadas através do seguinte estratagema. Lembre que a ata é formada por uma seqüência de registros. Uma vez usado o espaço disponível em uma página física, passa-se a seguinte, o conceito de página sendo mera conveniência imposta pelos mecanismos de ler e escrever em memória secundária. Assim, o próprio uso da ata impõe uma ordem natural nas páginas da mesma. Esta ordem é dada pela seqüência temporal em que cada página foi usada pela primeira vez. Tendo estes fatos em mente, sempre que uma página *P* da ata for forçada para a memória secundária, *todas as páginas da ata anteriores a P* também são enviadas a memória secundária, quer pertençam ou não a cadeia de ponteiros que nos levou a *P*. Estas páginas podem ser facilmente identificadas pelos seus endereços em memória secundária. Agora não mais precisamos consultar páginas da ata em memória secundária ao percorremos qualquer cadeia de ponteiros. Basta obtermos a primeira página da cadeia e acionarmos o gerente de paginação para que envie a memória secundária todas as páginas anteriores a esta. Note que, uma vez que certo espaço da ata tenha sido ocupado por um registro, nunca mais será tocado, pelo menos no cenário que estamos assumindo agora onde o espaço dedicado a ata em memória secundária é tido como ilimitado. Assim, uma vez que certa página da ata já tenha sido instalada em memória secundária, esta página não mais será alterada e, portanto, a versão em memória secundária é permanente. Este fato dá margem a que melhoremos ainda mais o tráfego de páginas entre a memória principal e a memória secundária. Basta manter em cada página um "bit" indicando se esta página já foi escrita em memória secundária *após* ter sido *totalmente* completada com registros da ata. Se tal for o caso, durante a operação de forçar páginas para a memória secundária poderíamos terminar o processo ao encontrarmos uma tal página, visto que suas antecessoras, pelo estratagema proposto, também estarão na mesma condição.

Resumindo, dada uma cadeia de páginas a ser consultadas, basta examinar a primeira página, *P*. Se o "bit" indicativo de *P* for favorável, nada é feito. Se não o for, *P* e todas as suas antecessoras até a primeira página com "bit" indicativo favorável, serão enviadas a memória secundária. Confiando que, após esta discussão, o leitor poderá implementar as correções necessárias no algoritmo apresentado, caso deseje usar a alternativa de não forçar os registros tipo *LIDA* para a memória secundária, vamos evitar de faze-lo no texto.

9.1.4.2 Obtendo Balizadores

No algoritmo de controle de integridade assumíamos a existência de um ponteiro para a ata, *I*, e também de informações sobre transações incompletas, transações canceladas localmente e também transações que já haviam entrado na execução do protocolo bifásico. Também dispúnhamos de informação à cerca das páginas que estavam na área de trabalho no instante da falha.

Vamos agora indicar como obter tal ponteiro *I* e as informações correspondentes. Estas últimas serão escritas *na ata* em registros especiais, chamados *balizadores*, e *I* será tido, portanto, como um endereço sobre a ata. Vamos introduzir mais um tipo de registro que pode ser lançado na ata. Um registro tipo *BALIZADOR* conterá três campos:

TA: lista com a identidade de todas as transações que no ponto *I* seriam classificadas como:
- i) incompletas
- ii) canceladas localmente. Neste caso, em adição, sabemos a posição na ata do respectivo registro de *CANC_LOCAL*.

TR: lista com a identidade de todas as transações que já haviam iniciado o protocolo bifásico, junto com o respectivo estado de retorno;

PG: lista com o endereço de toda página do BD que estava na área de trabalho no instante indicado por *I*, junto com a posição do respectivo registro tipo *LIDA*.

Seria interessante se logo após uma falha primária pudéssemos saber imediatamente onde se encontra o último registro tipo *BALIZADOR*. Lembre que é a partir deste ponto que deveremos iniciar a parte preparatória do algoritmo de controle de integridade. Embora não seja essencial, postularemos a existência de um *registrador de emergência* cujo conteúdo indicará sempre a posição do último registrador tipo *BALIZADOR* que há na ata. Este registrador de emergência é suposto sobreviver a falhas primárias. A função de um registro tipo *BALIZADOR* é dupla:

1) prover informações iniciais à fase preparatória;
2) limitar até que ponto devemos regredir lendo registros da ata durante a fase 2 do algoritmo, quando estamos desfazendo ações elementares.

Sabemos que o ponto mais remoto sobre a ata a que devemos nos referir, ao recuperarmos o sistema local, depende de dois fatores:

1) da posição do último registro tipo *BALIZADOR* ;
2) da posição do registro tipo *LIDA* mais antigo relativamente a todos aqueles registros deste tipo que constam na lista *PG* no instante da falha.

O primeiro item acima expõe uma situação onde devemos assumir uma solução de compromisso. De um lado, quanto mais freqüentes os balizadores instalados na ata, tanto menos precisaremos regredir na ata quando da recuperação de falhas. De outro lado, tanto mais será degradado o tempo de resposta do sistema devido a carga adicional de se obter os balizadores. A solução final será ditada pela tolerância que devemos atingir no que tange ao tempo de resposta quando comparada à freqüência de ocorrência de falhas primárias e, também, de quão ágil deva ser o mecanismo de controle de integridade ao recompor o sistema após tais falhas.

Quanto ao segundo fator, é eliminado pelo próprio mecanismo de se obter os balizadores descrito em seguida:

1) LIMPANDO A ÁREA DE TRABALHO:
   a) seja *I* o endereço na ata do último registro tipo *BALIZADOR* ;
   b) para cada página *P* do BD que esteja na área de trabalho da memória principal:
      i) seja *Z* o endereço na ata onde se situa o correspondente registro tipo *LIDA* para esta página;
      ii) se *Z* for anterior a *I*, envie esta página, através do gerenciador de paginação, para a memória secundária;
      iii) insira na ata um novo registro tipo *LIDA* para esta página, modificando também o valor do ponteiro para a ata a ela associado de modo a refletir esta nova posição de seu registro tipo *LIDA*;

2) OBTENDO *TA*, *TR* e *PG* :
   a) espere o BD atingir um estado consistente;
   b) consultando o executor de transações local, construa o campo *TA* e o campo *TC* ;
   c) consultando o gerente de paginação local, construa o campo *PG* ;
3) construa um registro tipo *BALIZADOR* a partir das informações coletadas na fase anterior;
4) lance este registro na ata;
5) atualize o conteúdo do registrador de emergência de forma a conter uma indicação para esta posição na ata.

Esperar por um estado consistente não significa que devemos aguardar até que todas as transações confirmem ou cancelem. Apenas devemos ter cuidado em garantir que cada ação elementar solicitada já executou e nada mais poderá modificar o estado do BD. Ressaltamos que o envio de páginas do BD a memória secundária, através do gerenciador de paginação, deve obedecer a todo o ritual descrito anteriormente e referente a este tipo de operação. Então, a página deve ser "fixada" na área de trabalho, o registros correspondentes da ata são forçados para a memória secundária, etc. Inclusive, claro, é lançado na ata um registro de *ESCRITA* para esta página. A inserção do registro de *LIDA* para cada página enviada a memória secundária tem o efeito de transportar seu registro tipo *LIDA* anterior para a última posição da ata, minimizando, assim, a distância que devemos retornar sobre a ata ao recuperarmos o sistema.

9.1.4.3 Atas com Espaço Limitado

É evidente que supor que dispomos de espaço ilimitado para acomodar a ata, como fizemos até agora, é uma abstração. Há várias alternativas possíveis. Todas devem, entretanto, garantir que sempre haverá meios de se registrar as atividades necessárias na ata, a qualquer momento. Se necessário, transações serão eleitas para serem canceladas ou partes da ata serão copiadas para memória secundária dormente. Em geral, atas podem ser gerenciadas como uma estrutura em anel, onde novos registros são adicionados de um lado e retirados de outro, até que não haja mais espaço disponível. Um certo registro da ata pode ser destruído, ocupando-se seu espaço para outros registros, quando

1) a correspondente transação já confirmou ou cancelou, local ou globalmente, e
2) a correspondente página já foi reenviada para a memória secundária.

Em última instância, pode-se recorrer a descargas da ata em memória secundária dormente.

9.1.4.4 Falhas Secundárias

Toda a discussão desta seção está baseada no fato de que o meio onde se encontra a ata está imune a falhas. Esta máxima pode ser aproximada na medida em que a ata é duplicada em dispositivos físicos distintos. Desta forma, uma falha em um deles ainda poderia ser recuperada usando-se o segundo. O preço, claro, está no tempo extra gasto em gerenciar duas atas distintas. Em teoria, poderíamos levar a situação adiante, triplicando, quadruplicando, etc., o número de dispositivos diferentes usados. Uma alternativa seria obtermos descargas em memória secundária dormente, usualmente localizadas em fita magnética.

Há duas maneiras principais de obtermos descargas em memória dormente que se adaptam melhor ao mecanismo de atas. Em primeiro lugar, poderíamos simplesmente deter as atividades do BD, não aceitando transações por um certo período de tempo. Após todas as transações já submetidas terem terminado, cancelando ou confirmando, uma descarga de todas as páginas do BD seria lançada em memória dormente. Na eventualidade de uma falha secundária, esta cópia estaria pronta para ser usada. Se o preço de se impedir o uso do BD de tempos em tempos for muito elevado, poderemos obter descargas individuais de cada página do BD de forma dinâmica, sem deter o sistema.

Neste último caso o próprio sistema submeteria "transações" que consistiriam em copiar páginas do BD para memória dormente. Visto que os mecanismos de controle de concorrência proíbem que uma transação inspecione o conteúdo de objetos que serão modificados por outras transações, o próprio controle de concorrência poderia ser utilizado para se obter descargas de páginas em estados aceitáveis. O problema residiria no fato de que páginas diferentes refletiriam estados diferentes do BD no sentido de que as páginas mais recentes podem ter sido modificadas por transações que nem existiam quando outras páginas mais antigas foram descarregadas. A situação é remediada construindo-se também uma ata em memória dormente, contendo apenas os registros referentes a transações que confirmaram. Note que não é necessário obtermos registros de ata para as transações que cancelaram. Isto porque estamos obtendo imagens de páginas em um estado consistente em relação as transações que executam no sistema. Suponha que uma transação *T* venha a cancelar. Então, até que tenhamos lançado em ata o registro tipo *CANCELAMENTO*, ou *CANC_LOCAL*, para esta transação, os mecanismos de controle de concorrência não permitirão que tenhamos acesso, através de outra transação, ao conteúdo de páginas modificadas pela transação que cancelou. Mas, após termos lançado o referido registro na ata, já teremos invertido todos os efeitos desta transação, em todas as páginas onde alterou valores de objetos do BD. Este mecanismo é parte integrante do processo de cancelar transações. Logo, as transações canceladas, após termos lançado em ata o correspondente registro tipo *CANCELAMENTO* ou *CANC_LOCAL*, podem ser ignoradas no instante de obtermos a descarga em memória dormente. À medida em que são obtidas cópias em memória dormente para cada página do BD precisamos, claro, obter também cópias em memória dormente dos registros de todas as transações que vão confirmando. Estas últimas podem ser obtidas regularmente como parte do processo de aliviar a área disponível para uso da ata em memória secundária ativa. A única dificuldade estaria em decidirmos, à medida que examinamos registros da ata, se a correspondente transação confirmou ou cancelou, visto que o registro de tipo apropriado está mais adiante na ata. Esta informação pode ser adquirida lendo-se, cada vez que encontramos um registro tipo *INÍCIO*, os próximos registros da ata até encontrarmos o registro de término da transação em questão. No caso desta última ainda não haver terminado, o mecanismo entra em compasso de espera até que se dê o desenlace. Não é difícil imaginar que podemos, desta forma, manter uma lista de transações que estão ativas, cada uma marcada apropriadamente, segundo tenha terminado ou cancelado. Assim, a medida que prosseguimos, podemos lançar em memória secundária dormente apenas os registros correspondentes a transações que confirmaram. O processo de recuperação de uma falha que violasse a ata em memória secundária ativa consistiria, basicamente, em refazermos as ações elementares a partir da versão mais recente em memória dormente, de todas as transações que houvessem confirmado. O preenchimento dos detalhes é deixado a cargo do leitor.

9.2 IMAGENS TRANSIENTES

Nesta seção, uma outra técnica de controle de integridade local é abordada. Esta técnica permitirá recuperar o último estado consistente do BD após falhas primárias ou secundárias, ou seja, falhas que causem a perda do conteúdo da memória principal ou da memória secundária, respectivamente, bem como desfazer localmente os efeitos de uma transação incompleta durante a operação normal do sistema.

Inicialmente, uma implementação do nível físico do subsistema de armazenamento, sem mecanismos de controle de integridade, é descrita. Em seguida, a implementação é modificada para permitir a recuperação de falhas primárias e a anulação dos efeitos de uma transação. A idéia básica consiste em manter, para cada página alterada por uma transação, uma *imagem transiente* refletindo as alterações e uma *imagem certificada* com o conteúdo original da página. Quando a transação termina, as imagens transientes são efetivadas e as imagens certificadas descartadas. Isto é feito para todas as páginas através de uma ação bem simples. Naturalmente, estruturas de dados auxiliares são necessárias para atingir este efeito. A implementação sofrerá, então, uma segunda modificação, incorporando um mecanismo de *descargas incrementais dinâmicas*, que permitirá a recuperação de falhas secundárias. Nestas três implementações supõe-se que as transações são executadas seqüencialmente, evitando-se assim os problemas de controle de concorrência. Por fim, apresenta-se uma última modificação incorporando controle de concorrência.

Todos os mecanismos aqui apresentados se referem-se a controle de integridade local. Rotinas adicionais são necessárias em um ambiente distribuído para suprir outras funções como, por exemplo, avisar o nó coordenador de uma transação em caso de cancelamento local forçado por uma falha.

9.2.1 Implementação Básica

Esta seção apresenta uma implementação do nível físico do subsistema de armazenamento sem considerar mecanismos de controle de integridade ou de controle de concorrência.

9.2.1.1 Definição da Interface

Nesta primeira implementação, e em todas as que se seguem, supõe-se que a memória secundária disponível é dividida em *M segmentos* de igual tamanho e que cada segmento é, por sua vez, dividido em *N páginas* de tamanho fixo. Os segmentos serão denotados por S_1 ,..., S_M por razões mneumônicas. Já as páginas de um segmento serão simplesmente numeradas de *1* a *N*.

A interface oferecida por esta implementação aos outros subsistemas consiste das seguintes operações básicas (onde $\boldsymbol{S}$ é um conjunto de segmentos e *T* é a identificação de uma transação):

*ABRA(**S**,T)*

Para cada $S_i \in \boldsymbol{S}$, torna S_i disponível para *T*.

*FECHE(**S**,T)*

Para cada $S_i \in \boldsymbol{S}$, assegura que todas as páginas de S_i que estão na memória principal e que foram modificadas por *T* são gravadas na memória secundária, e torna S_i não disponível para *T*.

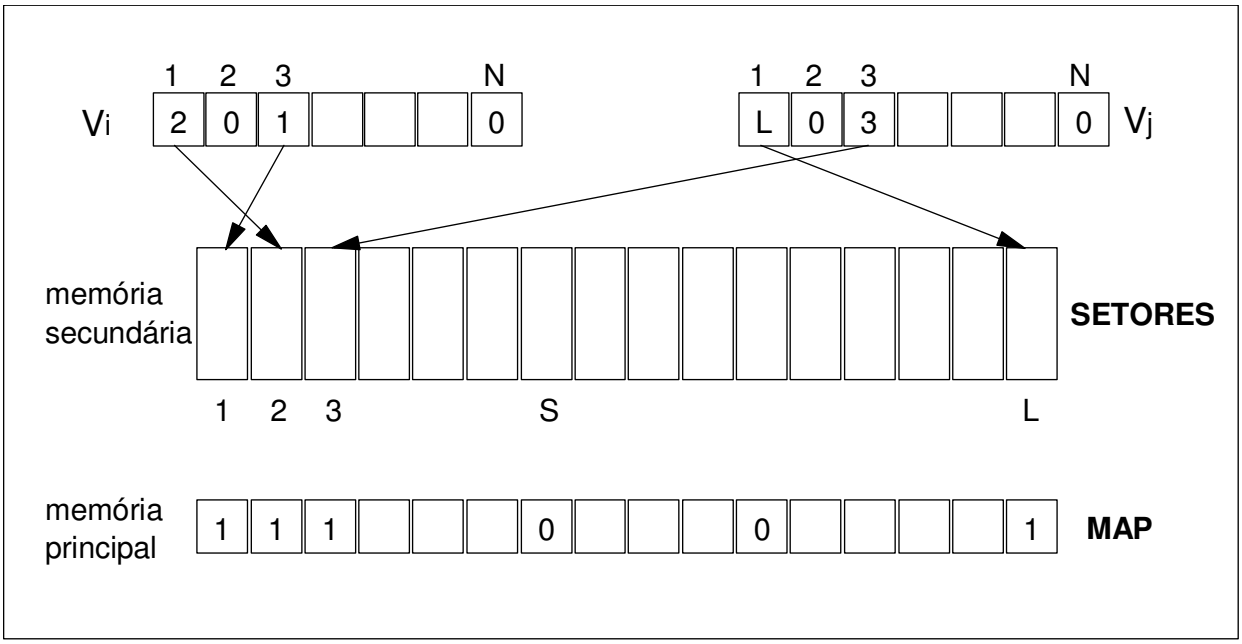

Figura 9.11 - Mapeamento entre páginas e setores

LEIA(S_i,j,F,T)

Lê a página *j* do segmento S_i para a memória principal, retornando o endereço da área da memória principal que ocupa. O conteúdo da página poderá ser ou não atualizado pela transação *T* que executou a operação, caso *F=1* ou *F=0*.

GRAVE(S_i,j,k,T)

Regrava a página *j* do segmento S_i , que estava na memória principal na área *k*, na memória secundária.

9.2.1.2 Implementação da Interface

Os segmentos e páginas são ainda um conceito lógico, podendo ser mapeados na memória secundária de várias formas. A mais simples seria alocar todas as páginas de um mesmo segmento em espaço contíguo da memória secundária. No entanto, contigüidade física é difícil de manter. Para evitar este problema e, principalmente, para facilitar a incorporação de mecanismos de controle de integridade, um segundo mapeamento será adotado.

Considere que a memória secundária está dividida em *L setores* de tamanho idêntico ao das páginas. Os setores são numerados de *1* a *L*. O mapeamento das páginas para os setores é feito através das seguintes estruturas de dados:

V_i vetor de ponteiros para o segmento S_i, de comprimento igual ao número de páginas por segmento, tal que $V_i(j)$ contém o endereço do setor que mantém a página *j* de S_i, ou $V_i(j)= 0$ se a página não está alocada. V_i está dividido em *blocos* de tamanho fixo para facilitar a sua gerência. Reside na memória secundária e é partilhado por todas as operações.

MAP vetor de "bits", de comprimento igual ao número de setores, tal que *MAP(s)=1*, se o setor *s* contém uma página de algum segmento, e *MAP(s)=0*, se o setor *s* está livre. Reside na memória principal e é partilhado por todas as operações.

A Figura 9.11 ilustra este mapeamento.

A memória principal disponível para o subsistema de armazenamento está dividida em *K áreas de trabalho*, numeradas de *1* a *K*, do mesmo tamanho que as páginas dos segmentos. A gerência das áreas de trabalho não será abordada aqui. Há também um outro conjunto de áreas de trabalho para conter os blocos dos vetores de ponteiros V_i.

As operações básicas serão então implementadas da seguinte forma:

*ABRA(**S**,T)*

Para cada $S_i \in \boldsymbol{S}$, localize onde estão armazenados os blocos de S_i.

LEIA(S_i,j,F,T)

Leia o bloco de V_i contendo $V_i(j)$, se já não estiver na memória principal. Obtenha uma área de trabalho disponível *k*. Se $V_i(j) \neq 0$, leia o setor apontado por $V_i(j)$ para a área *k*, retornando o endereço de *k*. Se $V_i(j) = 0$, a página não está alocada; neste caso apenas retorne o endereço de *k*.

GRAVE(S_i,j,k,T)

Leia o bloco de V_i contendo $V_i(j)$, se já não estiver na memória principal. Se a página *j* não estava alocada, ou seja, se $V_i(j) = 0$, pesquise em *MAP* um setor livre *s*, e faça *MAP(s)=1* e $V_i(j) = s$. Caso contrário, o setor *s* é o próprio valor de $V_i(j)$. Grave então o conteúdo da área de trabalho *k* no setor *s*. *T* é a identificação da transação que invocou a operação, e não é usada nesta implementação (mas sim nas modificações descritas nas seções seguintes).

*FECHE(**S**,T)*

Para cada $S_i \in \boldsymbol{S}$, regrave na memória secundária os blocos e páginas de S_i que foram modificados (um bloco poderá ser modificado se uma das páginas que lhe corresponde foi alocada ou desalocada) e que se encontram ainda na memória principal. Para gravar a página *j* que se encontra na área de trabalho *k*, utilize a operação *GRAVE(S_i,j,k,T)*.

As seções seguintes incorporarão mecanismos de controle de integridade a esta implementação, que é bastante vulnerável a falhas.

9.2.2 Proteção contra Falhas Primárias

Esta seção introduz modificações na implementação básica de tal forma a torná-la robusta a falhas primárias. Novamente, assume-se que não há controle de concorrência e que, portanto, as transações devem ser processadas seqüencialmente em cada nó.

O objetivo básico será fornecer um mecanismo para retornar o banco de dados ao último estado

consistente anterior a uma falha primária. Este estado consistente é definido como aquele gerado pela execução de todas as transações que terminaram até o momento da falha (na mesma ordem). As modificações introduzidas permitirão também desfazer os efeitos de uma transação parcialmente executada.

A idéia básica para resguardar o banco de dados contra falhas primárias é muito simples. Para cada página j de cada segmento S_i acessado pela transação são mantidas, durante o processamento, uma *imagem transiente* refletindo as alterações efetuadas pela transação em j, e uma *imagem certificada* com o conteúdo original de j. Quando a transação termina, as imagens transientes são efetivadas e as imagens certificadas descartadas, para todas as páginas de forma atômica, ou seja, ou todas as imagens transientes são efetivadas, ou nenhuma delas o é. Este é o ponto difícil de se atingir, requerendo uma duplexação das estruturas de dados usadas na implementação básica, conforme descrito de forma geral a seguir.

9.2.2.1 Como Atingir Atomicidade

Considere o seguinte problema genérico. Seja D uma tabela (ordenada) armazenada na memória secundária. Registros desta tabela são trazidas para a memória principal por processos, atualizados e regravados novamente na memória secundária, várias vezes. Em presença de falhas primárias, um processo pode ser interrompido deixando a tabela em um *estado inconsistente*. Este problema decorre essencialmente do fato de que a execução de um processo não pode ser considerada como atômica.

Para resolver este problema e atingir atomicidade, introduz-se uma técnica de *duplexação*, que pode ser abstraída da seguinte forma:

1) Substitua a tabela D por duas outras, $D(0)$ e $D(1)$ e por uma variável C, todas armazenadas na memória secundária. Cada processo, ao iniciar, recebe um número de senha monotonicamente crescente. A senha do último processo que terminou corretamente é sempre armazenada em C, de forma incorruptível. Cada registro de $D(0)$ e $D(1)$ é agora um par (d,r) onde d é o número de senha do processo que criou o registro r. Assume-se que cada registro também é gravado de forma incorruptível, de tal forma que a afirmação acima sobre d e r seja sempre verdade, mesmo em presença de falhas. Ou seja, a regravação de um par (d,r) é suposta atômica.
2) Antes de usar a tabela, um processo recebe uma senha d tal que $d > C$.
3) O processo lê, atualiza e regrava livremente $D(0)$ durante seu processamento normal. Para cada registro gravado, o valor de senha é mudado para d.
4) Quando o processo termina, execute as seguintes operações:
   a) Grave todos os registros atualizados que ainda estão na memória principal em $D(0)$.
   b) Após terminar corretamente a gravação, leia C, faça $C := d$, e regrave C. Estas operações sinalizam que o processo com senha d terminou de atualizar $D(0)$ corretamente.
   c) Se a gravação de C foi correta, copie todas as modificações feitas pelo processo em $D(1)$.
5) Se houver uma falha, recupere o último estado consistente de $D(0)$ e $D(1)$ da seguinte forma:
   a) Sejam $D(0)[j]=(d_0,r_0)$ e $D(1)[j]=(d_1,r_1)$ as j-ésimas entradas de $D(0)$ e $D(1)$, respectivamente.

b) se $d_0 > C$, então faça $D(0)[j]:=D(1)[j]$ pois a j-ésima entrada de $D(0)$ foi criada por um processo que não terminou corretamente.

c) se $d_0 = C$ e $d_1 < C$, então faça $D(1)[j]:=D(0)[j]$, pois a j-ésima entrada de $D(1)$ não reflete as atualizações do último processo que terminou corretamente, mas $D(0)$ reflete.

Note que a gravação de C tem que ser considerada indivisível, o que é bastante razoável. Além disto, quando um par (d,r) for transferido para memória secundária, r deve ser gravado antes de d. Assim, se $d=C$ então r certamente será o valor criado pela transação com senha d.

Esta técnica pode ser estendida para que n estruturas de dados $D_1 ,...,D_n$ sejam sempre mantidas coerentes. Para tal, basta introduzir um vetor $(C_1 ,..., C_n)$, tal que C_i faz o papel de C para a estrutura D_i. Se a regravação do vetor não for confiável, pode-se introduzir uma nova variável CC e duplexar o vetor. Desta forma, reduzimos novamente o problema a gravar apenas uma variável, CC, de forma indivisível. Na verdade, "hardware" especial pode ser projetado para armazenar de forma confiável apenas esta variável, ou o próprio vetor, evitando-se completamente o problema de gravação na memória secundária.

9.2.2.2 Estruturas de Dados da Implementação Modificada

Voltemos agora ao problema de controle de integridade, começando por uma classificação dos estágios por que passa o processamento de uma transação (supondo que o protocolo bifásico é sempre invocado ao final):

Estágio 0: do início até atingir a primeira fase do protocolo bifásico.

Estágio 1: após passar pela primeira fase do protocolo bifásico até terminar.

Lembremos que a idéia básica do mecanismo de controle de integridade é manter mais de uma versão, ou *imagem*, de uma página. Os estágios de uma transação induzem, então, uma classificação para as imagens de uma página da seguinte forma:

imagem certificada: imagem criada pela última transação que terminou corretamente;

imagem em certificação: imagem criada por uma transação que passou pela primeira fase do protocolo bifásico mas ainda não terminou;

imagem transiente: imagem criada por uma transação que está em processamento e que ainda não passou pela primeira fase do protocolo bifásico.

Em um dado instante t, chamaremos então de *estado certificado do banco* à coleção das imagens certificadas das páginas no instante t, e chamaremos de *estado transiente do banco* à coleção das imagens criadas por último (independente de tipo). Dado que as transações são executadas seqüencialmente, por suposição, o estado certificado do banco é então aquele criado por todas as transações que terminaram corretamente até t, e o estado transiente é o estado real no instante t.

Novas estruturas de dados serão introduzidas e a implementação das operações será alterada de tal forma que:

- estado transiente, naturalmente, estará sempre disponível, e
- estado certificado criado pela última transação que terminou corretamente estará sempre acessível, de forma incorruptível.

A técnica de duplexação será usada para atingir o segundo objetivo, o que exige associar senhas monotonicamente crescentes às transações. As estruturas de dados serão modificadas da seguinte forma:

V_i vetor definido de forma semelhante à implementação básica. Reside na memória secundária e é tal que

$V_i(j) = (0,0)$ se a página j nunca foi alocada; ou

$V_i(j) = (s,t)$ se s é o setor que contém a imagem da página j criada por último, e t é a senha da transação que criou esta imagem.

VC_i vetor duplexando V_i. Reside na memória secundária.

MAP definido de forma idêntica à implementação básica. Reside na memória principal.

L_i lista de setores ocupados por páginas de S_i que deverão ser liberados quando S_i for fechado. Reside na memória principal. (No caso de execução concorrente, discutido no final da seção, existirá uma encarnação de L_i para cada transação T que usar S_i).

B_i lista de blocos de V_i que foram alterados pela transação. Reside na memória principal. (No caso de execução concorrente, discutido no final da seção, existirá uma encarnação de B_i para cada transação T que usar S_i).

CONTROLE estrutura de dados, residindo na memória principal, mas com uma cópia na memória secundária, usada para indicar qual a senha da última transação que terminou corretamente e qual o estado da transação corrente. Contém as seguintes variáveis, nesta implementação:

C senha da última transação que terminou corretamente.

D *D=1* indica que a transação corrente já passou pela primeira fase do protocolo bifásico mas ainda não terminou, e
D=0 indica o contrário.

Estas estruturas serão mantidas de tal forma que a assertiva *P* abaixo será sempre verdadeira, mesmo em caso de falhas durante a execução das operações, supondo que *P* seja verdadeira no início do processamento. Assume-se apenas que a gravação de *CONTROLE* não está sujeita a falhas. Além disto, quando um par *(s,t)* for transferido para memória secundária, *s* deve ser gravado antes de *t*. Assim, se *t=C* então s certamente será o valor criado pela transação com senha *t*.

ASSERTIVA P

Definição dos Estágios da Transação Corrente

Suponha que a senha da transação corrente seja *d*.

D=0 e *d > C* sse *T* ainda não passou pela primeira fase do protocolo bifásico.

D=1 e *d > C* sse *T* passou pela primeira fase mas ainda não terminou.

d=C sse *T* terminou.

Definição dos Estados de uma Imagem

Suponha que $V_i(j) = (s,t)$ e que $VC_i(j) = (s',t')$.

se *D=0* e *t > C* então *s* contém a imagem transiente da página *j* e *s'* contém a imagem certificada de *j*.

se *D=1* e *t > C* então *s* contém a imagem em certificação da página *j* e *s'* contém a imagem certificada de *j*.

se *t = C* então *s* contém a imagem certificada da página *j*. Neste caso *s'* contém a imagem certificada de *j*, se *t=t'*, ou uma imagem indefinida, caso $t' \neq t$ (pois pode ter havido uma falha na atualização de $VC_i(j)$).

se *t < C* então *s'* contém a imagem certificada da página *j*. Neste caso *s* contém a imagem certificada de *j*, se *t=t'*, ou uma imagem indefinida, caso $t' \neq t$ (pois pode ter havido uma falha na atualização de $V_i(j)$).

Note que a assertiva *P* acima apenas resume a definição das estruturas. O fato importante é que *P* será um invariante das operações, mesmo em presença de falhas, assumindo-se apenas a gravação incorruptível de *CONTROLE*. Note ainda que o uso de vetores de ponteiros para definir os segmentos é crucial neste ponto pois permite representar o estado transiente e o estado certificado de forma duplexada sem grande desperdício de memória.

9.2.2.3 Implementação das Operações

Além das operações da implementação básica, três novas operações são introduzidas para controle de integridade (***S*** representa um conjunto de segmentos e *T* uma transação):

*SALVE(**S**,T)*

Salva atualizações que se encontram ainda na memória principal para a memória secundária.

*RESTAURE(**S**,T)*

Retorna o conteúdo dos segmentos em ***S*** aos seus valores originais, descartando as modificações feitas pela transação.

RECUPERE

Retorna o banco de dados ao estado certificado que existia no momento da falha (primária).

Suporemos que estas operações e aquelas apresentadas na seção anterior são utilizadas para processar as transações (distribuídas) da seguinte forma. Assume-se inicialmente que cada transação recebe uma senha monotonicamente crescente. Lembremos ainda que a suposição de que as transações são processadas seqüencialmente em cada nó ainda está em efeito. Localmente, um nó lê e grava páginas para uma transação através de *LEIA* / *GRAVE*. Cada segmento S_i é aberto para T ao primeiro acesso a alguma página do segmento. Ao final, T executa um *FIM_DE_TRANSACAO*, invocando o protocolo bifásico. Localmente em um nó n, o processamento se dá então da seguinte forma. Antes de n votar *PREPARADO*, as atualizações locais da transação T são salvas na memória não volátil através de *SALVE(**S**,T)*, onde ***S*** é o conjunto de segmentos usados localmente por T. Se n votar *IMPOSSIBILITADO*, as atualizações são descartadas através de uma operação *RESTAURE(**S**,T)* Quando o nó recebe a mensagem *CONFIRME* final, as atualizações de T são tornadas visíveis *de forma atômica e recuperável* através da execução de uma operação *FECHE(**S**,T)* Se a transação não foi aceita, e uma mensagem *CANCELE* for recebida, basta executar uma operação *RESTAURE(**S**,T)*

Durante o processamento da transação T, se for necessário cancelá-la basta executar *RESTAURE(**S**,T)* para descartar as suas atualizações.

Tendo em vista este modelo de processamento de transações, a implementação das operações será modificada para que a assertiva P acima seja um invariante. Convém observar neste ponto que P implica em que a seguinte assertiva também será um invariante:

ASSERTIVA Q

Para cada página j de cada segmento S_i, seja $V_i(j) = (s,t)$ e $VC_i(j) = (s',t')$. A imagem certificada de j estará em s, caso $t = C$, ou em s', em caso contrário.

Desta forma, depois de uma falha que cause a perda do conteúdo da memória principal, será sempre possível recuperar o estado certificado no momento da queda: basta inspecionar os setores como indicado no enunciado de Q. Como este estado reflete exatamente a execução de todas as transações que terminaram corretamente até o momento da falha, pois Q também é um invariante, pode-se dizer que a recuperação estará correta.

Suponha que *MAP* seja iniciado corretamente para indicar quais setores estão ocupados. (Veja a descrição de *RECUPERE* abaixo). A implementação das operações será modificada da seguinte forma:

*ABRA(**S**,T)*

Para cada $S_i \in \mathbf{S}$, localize onde estão armazenados os blocos de V_i. Inicie uma lista L_i para T como vazia.

LEIA(S_i,j,F,T)

A implementação desta operação não é alterada. Ou seja, as transações sempre lêem do estado transiente.

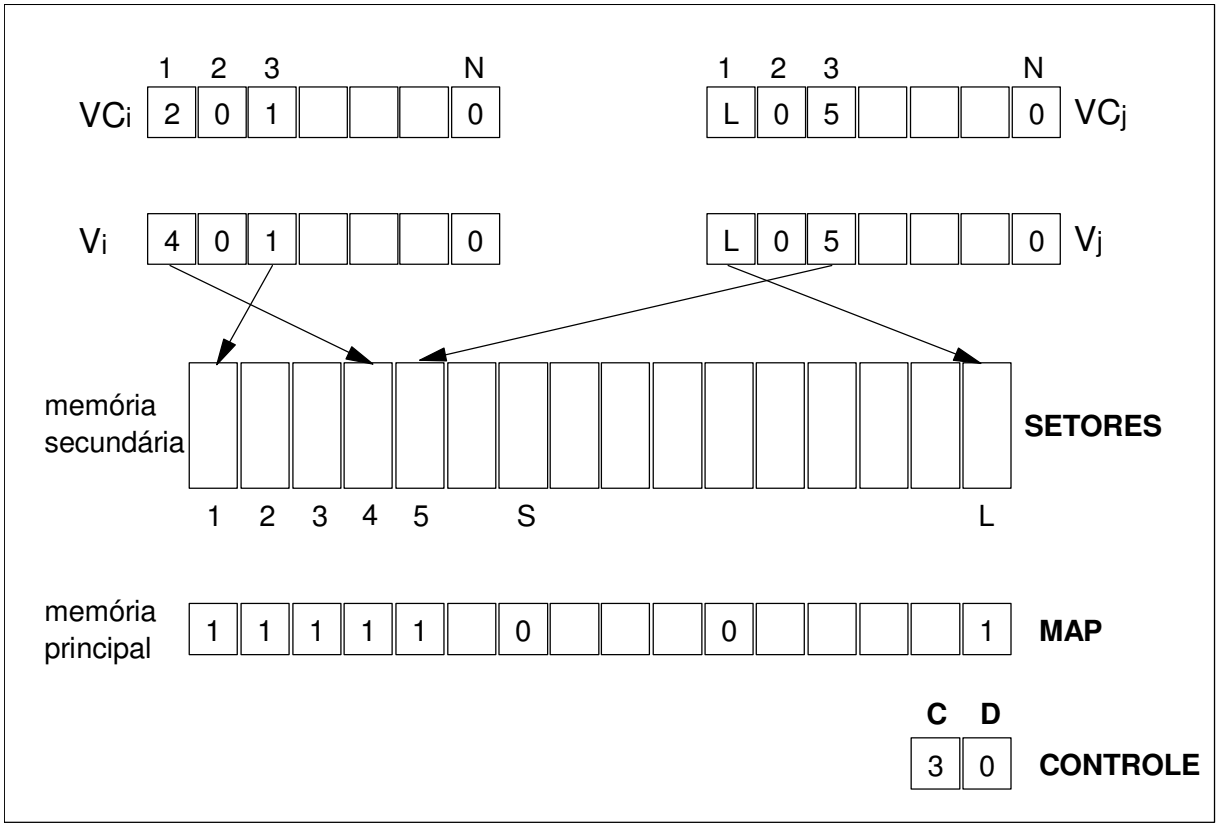

Figura 9.12 - Estruturas após *GRAVE(S_i,j,k,T)*

GRAVE(S_i,j,k,T)

> A implementação desta operação reflete a idéia básica de que uma página, ao ser modificada, não é regravada no mesmo lugar. O setor que ocupa continuará contendo a imagem certificada da página, enquanto que um novo setor é escolhido para conter a imagem transiente da página. Estas duas imagens coexistirão até que o segmento seja fechado.

Estas idéias são implementadas da seguinte forma.

Seja *d* a senha de *T*. Leia o bloco de V_i contendo a página *j* para a memória principal, se lá já não estiver. Suponha inicialmente que $V_i(j) = (s,t)$. Se a página *j* já foi modificada por *T*, temos que *t=d*. Logo, apenas copie a página *j* da área *k* para o setor *s*. Se a página *j* ainda não foi modificada ou nunca foi alocada, temos t≠d. Obtenha um novo setor *s'* não utilizado pesquisando *MAP*. Copie a página *j* da área *k* para o setor *s'*. Em seguida, indique que agora *s'* contém a imagem transiente de *j* fazendo $V_i(j) = (s',d)$.

Devemos observar neste ponto que cada bloco de V_i lido será regravado, se tiver sido modificado pela transação, quando for necessário liberar espaço para novos blocos. A identificação destes blocos modificados e regravados deverá ser mantida na lista B_i da transação que os alterou até que *T* termine.

MAP deve ser então atualizado. Os setores *s* e *s'* permanecerão alocados para todos os efeitos até que a transação termine corretamente. Logo, apenas aloque o setor *s'* fazendo *MAP(s')=1*. Acrescente o setor *s* à lista L_i da transação *T* para ser liberado quando *T* terminar.

O vetor VC_i na memória secundária será mantido intacto até que *T* termine corretamente. Assim sendo, esta operação também preserva, trivialmente, a propriedade *P*.

A Figura 9.12 ilustra o resultado da operação $GRAVE(S_i,j,k,T)$, assumindo, digamos, *j=1* e *d=3*. Inicialmente, um setor não utilizado, digamos *s=4*, é obtido pesquisando-se *MAP*. O setor é marcado como utilizado fazendo-se *MAP(4)=1*, conforme mostrado. Em seguida, o setor *4* é alocado para a imagem transiente da página *1*, fazendo-se $V_i(1) = (4,3)$. O segundo argumento de $V_i(j)$ não é mostrado explicitamente. Note que $VC_i(j)$ permanece inalterado pois aponta para a imagem certificada de *j*. O setor *2* é acrescentado à lista L_i (não mostrada) para ser desalocado quando a transação terminar.

*SALVE(**S**,T)*

> Salve para a memória secundária as páginas atualizadas, através de *GRAVE*, e os blocos modificados de V_i que se encontram ainda na memória principal. Estes blocos são também acrescentados a B_i.
>
> Apenas após a execução correta destas atualizações, leia *CONTROLE*, faça *D := 1*, e regrave *CONTROLE* efetivando a execução de *SALVE*. O nó poderá então votar *PREPARADO*, e esperar o coordenador enviar *CONFIRME* ou *CANCELE*.

*FECHE(**S**,T)*

> *FECHE* deverá instalar, de forma atômica, um novo estado certificado. Seguindo o modelo de processamento de transações apresentado no início desta subseção, supõe-se que *SALVE(**S**,T)* já foi corretamente executada e que, portanto, todas as atualizações feitas pela transação nas páginas e em V_i já se encontram salvas na memória secundária corretamente.
>
> Seja *d* a senha de *T*. Sinalize que *T* terminou corretamente. Leia *CONTROLE*, faça *C := d* e *D := 0*, regravando *CONTROLE* em seguida. O nó poderá então enviar ao coordenador a resposta da mensagem *CONFIRME*, neste ponto.
>
> Após a gravação de *CONTROLE* ter sido corretamente efetuada, os blocos de V_i atualizados, que se encontram na lista B_i, são também gravados em VC_i. Desta forma, V_i e VC_i ambos refletem as alterações da transação, se esta terminou corretamente.
>
> Finalmente *MAP* é modificado, efetivamente liberando os setores que continham imagem certificadas obsoletas, que são mantidas na lista L_i.

Assumindo-se que a regravação de *CONTROLE* é atômica e que as transações invocam *FECHE* apenas quando terminam corretamente, esta operação também preserva a assertiva *P*. Note que é necessário também lembrar que, localmente, as transações são executadas seqüencialmente. Logo, quando uma transação termina, o estado transiente mantido em V_i coincide com o novo estado certificado, o que não é verdade em presença de execução concorrente. Como o estado

certificado é atualizado segundo a técnica de duplexação, mesmo que uma falha interrompa a atualização da definição do estado certificado, *P* não é invalidado. Observe ainda que *MAP* não pode ser atualizado até que o vetor definindo o novo estado certificado tenha sido corretamente construído e *CONTROLE* atualizado pois, de outra forma, o seguinte cenário poderia ser criado:

1) Suponha que o setor *s* contém a imagem certificada da página *j* do segmento S_i e que $V_i(j) = VC_i(j) = (s,t)$. Seja *d* a senha da transação *T* corrente.

1) *T* cria uma imagem transiente para *j* no setor *s'*, fazendo $V_i(j)$ " := " (s',d), e libera a imagem certificada em *s*, alterando $MAP(s)$ para *0*.

2) O setor *s* é reusado para conter a imagem transiente da página *k* do segmento S_l.

3) Uma falha primária ocorre antes da transação terminar.

4) O sistema é recuperado para o último estado certificado. Como $V_i(j) = (s',d)$ e $d > C$, faz-se $V_i(j) := VC_i(j)$.

5) Porém, $VC_i(j) = (s,t)$ indica que o setor *s* contém a imagem certificada da página *j* do segmento S_i, o que não é verdade, pois ele contém a imagem transiente da página *k* de S_l.

Assim sendo é crucial que *MAP* só seja atualizado depois que o novo estado certificado foi corretamente criado.

*RESTAURE(**S**,T)*

Esta operação é mais ou menos o inverso de *FECHE(**S**,T)*, no sentido de que as imagens transientes das páginas deverão ser descartadas e as imagens certificadas mantidas. Novamente observe que VC_i mantém ponteiros para as imagens certificadas das páginas modificadas por *T*.

Seja *d* a senha de *T*. Para cada bloco *b* de V_i na lista B_i de *T* e para cada página *j* de S_i no intervalo correspondente a *b*, se $V_i(j) = (s,d)$, então libere *s* fazendo $MAP(s)=0$ e copie $VC_i(j)$ para $V_i(j)$.

Sinalize que a transação foi cancelada. Leia *CONTROLE*, faça *D := 0* e regrave *CONTROLE*. O nó poderá, então, enviar ao coordenador a resposta da mensagem *CANCELE* recebida neste ponto.

RECUPERE

A ação de *RECUPERE* depende do estado atingido pela transação no momento da falha. Para determiná-lo, leia inicialmente *CONTROLE* da memória secundária, obtendo os valores de *C* e *D*. Dependendo do valor de *D* temos os seguintes casos:

Caso 1: Suponha que *D=0*.

Então a transação pode ter sido interrompida antes de entrar no protocolo bifásico, ou durante *FECHE* (depois de ter atualizado *C* e *D*, mas antes de completar a atualização de VC_i. Eis porque $VC_i(j)$ pode não apontar para a imagem certificada da página *j*). Estas duas situações podem ser distinguidas pois, no primeiro caso, a

senha da transação é maior do que C, enquanto que no segundo caso é igual a C. O nó local deverá então desfazer os efeitos da transação, retornando o banco de dados local ao último estado certificado, no primeiro caso, ou terminar a atualização de VC_i, no segundo caso.

Para cada página j de cada segmento S_i, seja $V_i(j) = (s,d)$:

Caso 1.1: Suponha que $d \neq C$.

Faça $V_i(j) := VC_i(j)$

Caso 1.2: Suponha que $d = C$.

Faça $VC_i(j) := V_i(j)$

Para recuperar *MAP*, proceda da seguinte forma. Inicialmente, zere *MAP*; em seguida percorra todos os vetores V_i, para cada página j de S_i e, se $V_i(j) = (s,t)$, faça $MAP(s)=1$. Como a assertiva P é preservada por todas as operações, *RECUPERE* recria o estado certificado no momento da falha (veja a definição de estado certificado e a descrição de P acima).

Caso 2: Suponha que $D=1$.

Então a transação foi interrompida depois de *SALVE* ter sido executado e antes de *FECHE* ou *RESTAURE* terem terminado. Neste caso não há nada a fazer, pois o estado transiente criado ao final da transação já estava salvo na memória secundária. Apenas *MAP* deve ser reconstruído. Zere *MAP* inicialmente e depois percorra todos os vetores V_i e VC_i e, para cada página j de S_i, se $V_i(j) = (s,t)$ ou $VC_i(j) = (s,t')$, faça $MAP(s)=1$. O nó fica, então, esperando o coordenador reenviar *CONFIRME* ou *CANCELE* novamente, para reexecutar *RESTAURE* ou *FECHE*.

Isto conclui a descrição da implementação das operações.

Por fim, observamos que a recuperação de uma falha primária causando a perda do conteúdo da memória principal é simples: basta executar *RECUPERE*, quando o nó for reiniciado.

9.2.3 Proteção contra Falhas Secundárias

Esta seção incorpora novas modificações à implementação básica de tal forma a torná-la robusta também contra falhas secundárias, ou seja, a falhas no próprio meio que armazena o banco de dados.

A solução adotada segue a política de descarga incremental dinâmica. Para efeitos desta discussão, suporemos que a descarga é feita em fita. A descarga incremental será executada periodicamente, digamos a cada n chamadas para *FECHE* ou *SALVE*, para um dado n fixo.

Seja S o estado certificado no instante em que a descarga incremental foi iniciada pela última vez. Suponha que uma nova descarga incremental seja iniciada em um instante t' e que S' seja o estado certificado em t'. Há quatro preocupações básicas na implementação da descarga:

1) É necessário marcar que imagens certificadas foram criadas depois da última descarga (e portanto não têm cópias).
2) o momento que a descarga inicia, uma descrição de S' deverá ser congelada e cópias deverão ser feitas apenas das imagens marcadas de S'. Se S' não for congelado, as transações em andamento durante a descarga poderão tornar as cópias inconsistentes.
3) Uma vez completada a descarga, as imagens copiadas deverão ser desmarcadas. Caso contrário, as imagens marcadas de S' poderiam ser novamente copiadas pela segunda descarga.
4) Além disto, as transações em andamento não poderão liberar imagens certificadas sem cópia e que se tornaram obsoletas, que estão sendo copiadas pela descarga. Isto deverá ser feito quando a descarga terminar. De fato, suponha que um setor s contenha uma imagem certificada sem cópia de uma página j. Suponha que uma transação T inicie e crie uma imagem transiente da página j. Suponha que a descarga é executada concorrentemente com T e que, portanto, deva copiar o conteúdo de s. Quando T termina, a imagem certificada de j torna-se obsoleta e s deverá ser desalocado. Porém, isto não pode ser feito até que a descarga termine corretamente.

Estes problemas deverão ser resolvidos prevendo-se, ainda, que falhas primárias possam ocorrer. No que segue, assumiremos que as transações são processadas seqüencialmente.

Para se resolver o primeiro problema, basta guardar em uma variável E a senha da última transação que terminou antes do início da última descarga. De fato, se $V_i(j) = (s,t)$ e $E < t \leq C$, então s contém uma imagem certificada sem cópia.

O segundo problema é resolvido criando-se na memória secundária uma nova cópia de VC_i no momento em que a descarga iniciou.

O terceiro problema é resolvido adiando-se a transformação das imagens certificadas sem cópia em imagens com cópia para apenas quando a descarga terminar. Para tal, basta alterar o valor de E apenas quando a descarga terminar corretamente.

O quarto problema exige que, quando inicie, a descarga sinalize para *FECHE* (a operação que descarta imagens certificadas obsoletas), quais são as imagens certificadas sem cópia que pertencem ao estado certificado S' naquele momento. Além disto, é necessário que o processo de descarga desaloque os setores que continham imagens certificadas sem cópia de S', e que se tornaram obsoletas durante a execução da descarga, apenas quando terminar corretamente. Para tal, novas estruturas são introduzidas indicando que setores serão copiados pela descarga e quais destes setores foram liberados pelas transações durante a descarga.

As estruturas são alteradas da seguinte forma:

CONTROLE estrutura de dados, residindo na memória principal e com uma cópia na memória secundária para recuperação de falhas primárias. Contém as seguintes variáveis, nesta implementação:

- C (como anteriormente).
- D (como anteriormente).

E senha da última transação que terminou antes do início da última descarga dinâmica.

DATIVA *DATIVA=1* indica que uma descarga está em progresso, *DATIVA=0* indica o contrário.

COPIANDO vetor de comprimento igual ao número de setores, em memória principal, tal que

COPIANDO(s)=1 se *s* contém uma imagem certificada sem cópia que está sendo copiada.

COPIANDO(s)=0 em caso contrário.

LA lista dos setores contendo imagens certificadas sem cópia (no início da descarga) que deverão ser liberados ao final da descarga. Reside na memória principal.

O vetor *COPIANDO*, como *MAP*, reside na memória principal e durante o processamento usual está completamente zerado. Apenas quando a descarga inicia, os "bits" correspondentes aos setores com imagens certificadas sem cópia são levantados pela descarga.

Em presença de descargas dinâmicas, passamos a ter agora mais tipos de imagens:

- se $t < C$ e $D=0$, então a transação *T* não terminou ainda e *s* contém uma *imagem transiente* da página *j*;
- se $t < C$ e $D=1$, então a transação *T* passou pela primeira fase do protocolo bifásico mas ainda não terminou, e *s* contém uma *imagem em certificação* da página *j*;
- se $E < t \leq C$, então a transação *T* terminou depois da última descarga iniciar e *s* contém uma *imagem certificada sem cópia* da página *j*;
- se $t < C$ e $t \leq E$, então a transação *T* terminou antes da última descarga iniciar e *s* contém uma *imagem certificada com cópia* da página *j*;

O processo de descarga incremental é implementado como três operações: *INICIALIZAÇÃO*, *CÓPIA* e *TERMINAÇÃO*. Tanto *INICIALIZAÇÃO* quanto *TERMINAÇÃO* deverão ser sincronizadas com as outras operações executadas pelas transações em progresso. Porém, *CÓPIA* poderá ser executada sem nenhuma sincronização adicional, além daquela já embutida na própria definição das novas estruturas, conforme discutido acima. Isto caracteriza, na verdade, o próprio fato da descarga incremental ser dinâmica, ou seja, ser executada em conjunto com as transações. A definição destas três operações é a seguinte:

INICIALIZAÇÃO

Inicialmente, grave em fita uma marca, que será usada em caso de falhas, indicando que a descarga foi iniciada. Congele, em seguida, o estado certificado corrente copiando o vetor VC_i para um novo vetor VD_i, para cada segmento S_i. Os vetores VD_i são criados na memória principal e aí residem durante a execução da descarga. Copie *C* para *CD* na memória principal também.

Crie *COPIANDO* a partir de VD_i, para cada segmento S_i, fazendo *COPIANDO(s)=1* para cada setor *s* que contém uma imagem certificada sem cópia (ou seja, tal que $VD_i(j) = (s,t)$ e $t > E$ para alguma página *j* de S_i). Este vetor é criado na memória principal.

Finalmente, sinalize que a descarga iniciou. Leia *CONTROLE*, faça *DATIVA = 1* e regrave *CONTROLE*.

CÓPIA

Para cada entrada *j* de cada vetor VD_i, se $VD_i(j) = (s,t)$ e $t > E$, *s* contém uma imagem certificada sem cópia. A operação copia para fita a tripla *(i,j,v)*, onde *v* é o conteúdo do setor *s*.

O processo de cópia pode ser interrompido e recomeçado sem problemas, pois *E* ainda não foi alterada.

TERMINAÇÃO

Depois que o processo de cópia terminou corretamente, libere os setores contendo imagens certificadas obsoletas que estão contidos na lista *LA*. Assim, para cada *s* em *LA*, faça *MAP(s)=0*.

Ao final, adicione uma marca à fita indicando que o processo de descarga terminou. Sinalize que a descarga terminou corretamente. Leia *CONTROLE*, faça *DATIVA:=0*, *E:=CD* e regrave *CONTROLE*.

As operações *FECHE* e *RECUPERE* têm que ser modificadas em presença de descargas incrementais, como descrito abaixo. A correção destas novas implementações é obtida como anteriormente.

*FECHE(**S**,T)*

Apenas a atualização de *MAP* é agora diferente. A liberação de um setor não poderá ser efetivada se o setor contém uma imagem certificada sem cópia que está sendo copiada por uma descarga em progresso. Portanto, se *DATIVA=1*, para cada setor *s* em L_i, se *COPIANDO(s)=1*, então adie a liberação de *s* até que a descarga termine, e acrescente *s* à lista *LA*. Caso contrário, libere *s*, fazendo *MAP(s)=0*.

RECUPERE

Recupere o banco para o último estado certificado, como anteriormente. Se havia uma descarga em progresso, leia a fita, no sentido reverso, até encontrar uma marca de início de descarga. Os registros daí em diante poderão ser ignorados pois representam parte de uma descarga incompleta. Estes registros poderiam ainda ser aproveitados, mas isto tornaria a recuperação mais lenta. Optou-se por desconsiderá-los, forçando a reexecução da descarga desde o início.

Por fim, inicia-se nova descarga se havia uma em progresso no momento da falha (*DATIVA=1*). Note que o estado certificado processado pela nova descarga não é necessariamente aquele presente quando a descarga interrompida foi iniciada. Porém, todas as imagens certificadas sem cópia ainda são identificadas como anteriormente, pois

E só é alterada depois da descarga terminar corretamente.

Isto conclui a descrição do processo de descarga incremental dinâmica e das modificações das operações. Por fim, observamos que é possível recuperar da fita o estado certificado do banco de dados no momento em que a última descarga foi feita, através da seguinte operação.

RECUPERE2

Leia a fita para onde a descarga dinâmica é feita, do fim para o início. O primeiro registro do tipo *(i,j,v)* encontrado contém o conteúdo *v* da página *j* do segmento S_i do estado certificado do banco no momento do início da última descarga. Se não houver um mecanismo de atas, em caso de falhas na memória secundária, só será possível recuperar o banco para este estado, com perda das atualizações das transações que terminaram depois da última descarga.

9.2.4 Incorporação de Controle de Concorrência

Inicialmente, uma discussão genérica sobre como controle de concorrência pode ser acrescentado ao mecanismo de imagens transientes é apresentada. Em seguida, a implementação da última seção sofre nova modificação, incorporando o protocolo de bloqueio em duas fases para controlar o acesso concorrente a um banco de dados local.

9.2.4.1 Discussão Preliminar

A tarefa de introduzir controle de concorrência não é especialmente simples uma vez que a discussão do Capítulo 8 é razoavelmente abstrata quando comparada com o nível de detalhe das implementações descritas nesta seção. De fato, uma execução concorrente era abstraída pelo conceito de escalonamento, ou seja, por uma coleção (no caso distribuído) de seqüências de ações elementares. Cada ação elementar afetava um conjunto de objetos, identificados por seus nomes, e conhecidos "a priori". Esta última suposição é importante pois transforma o problema de verificar se duas operações conflitam em simplesmente testar se possuem um nome de objeto em comum no conjunto de nomes de objetos que acessam (e uma delas é uma atualização). Se os objetos acessados fossem definidos por predicados, por exemplo, a detecção de conflitos recairia no problema de determinar se dois predicados são simultaneamente satisfatíveis o que, dependendo do tipo de predicados, pode ser computacionalmente intratável (ou mesmo indecidível).

Para aproveitar os resultados sobre controle de concorrência, é necessário reinterpretar o conceito de escalonamento, identificando as ações elementares e os objetos nomeados. Esta reinterpretação recai no problema de determinar exatamente a que nível é feito o controle de concorrência. Em termos da estrutura do subsistema de armazenamento (SAR), há essencialmente três opções:

1) Controle de concorrência a nível lógico do SAR.

2) Controle de concorrência a nível físico do SAR.

3) Controle de concorrência a nível de acessos físicos a memória secundária.

A primeira opção é imediatamente descartada pois as operações oferecidas pela interface do nível lógico do SAR manipulam conjuntos de registros definidos por predicados. Desta forma, a detecção de conflitos se torna inviável, como discutido anteriormente. A única saída para se fazer controle de concorrência a este nível seria identificar os objetos com as próprias tabelas, ignorando os predicados. Porém, esta decisão forçaria decretar que duas operações conflitam quando acessam uma tabela em comum, mesmo que acessem registros diferentes da tabela. Portanto, não é razoável.

As duas últimas opções são igualmente plausíveis. Postularemos, então, que controle de concorrência será feito a nível físico do SAR com o intuito de mostrar como pode ser incorporado à discussão das seções anteriores. O resto desta subseção explora em mais detalhe esta alternativa.

Inicialmente, definiremos que um escalonamento local representará uma seqüência de operações oferecidas pela última implementação do nível físico discutida para o SAR. Ou seja, as ações elementares serão:

ABRE , LEIA , GRAVE , FECHE , SALVE , RESTAURE

e também aquelas implementando a descarga incremental:

INICIALIZACAO , COPIA , TERMINACAO

As operações *RECUPERE* e *RECUPERE2* não são consideradas pois representam exceções durante o processamento (recuperação de falhas primárias ou secundárias).

Cada ação elementar era implicitamente considerada como indivisível no capítulo de controle de concorrência, o que claramente não é verdade acerca das operações acima. Há duas formas de resolver este problema. Primeiro, podemos encarar cada operação como uma microtransação e implementar um segundo nível de controle de concorrência que force a serialização das execuções concorrentes destas microtransações. Porém, a serialização destas microtransações tem que ser compatível com aquela induzida para as transações pelo controle de concorrência do nível superior. Ou seja, T precede T' na serialização das transações se e somente se todas as operações de T precedem todas as operações de T' na serialização das operações. Esta propriedade não é fácil de se atingir, exigindo cuidados especiais.

Adotaremos aqui uma solução mais simples. Forçaremos as operações a serem executadas seqüencialmente. Cada operação, ao iniciar, imediatamente tenta ativar um semáforo, *ATIVA*, que controla o acesso a todas as estruturas da implementação. Após terminar, a operação desativa o semáforo. Desta forma, as operações são executadas seqüencialmente, mantendo a coerência interna das estruturas. Note que isto não força a execução seqüencial das transações, mas apenas garante a atomicidade das operações. A única operação não sujeita a esta regra é a operação de *COPIA* pois, de outra forma, a descarga não seria dinâmica.

Resta agora definir quem são os objetos. Consideraremos quatro opções:

1) Associar os objetos às próprias estruturas de dados (V_i, VC_i, etc.).

2) Associar os objetos aos segmentos.

3) Associar os objetos a intervalos de páginas $[j_i , k_i]$ tais que o i-ésimo bloco de V_i contém os valores de $V_i(j)$, para $j \in [j_i , k_i]$

4) Associar os objetos às páginas.

Note que as três últimas opções determinam particionamentos diferentes do conjunto de páginas.

As duas primeiras opções padecem do mesmo problema: induzem conflitos desnecessários pois representam objetos grandes demais, por assim dizer, degradando a eficiência do controle de concorrência. As duas últimas opções são igualmente plausíveis, em princípio, sendo a terceira explorada na subseção seguinte dentro do contexto do protocolo de bloqueio em duas fases.

9.2.4.2 Implementação do Protocolo de Bloqueio em Duas Fases

Nesta subseção mostraremos como o algoritmo de bloqueio em duas fases distribuído pode ser incorporado ao mecanismo de controle de integridade descrito anteriormente. Apenas o mecanismo local é mostrado, modificando-se a implementação das operações. De acordo com a alternativa escolhida na subseção anterior, os objetos a serem bloqueados correspondem aos intervalos de páginas associados aos blocos de V_i.

Em linhas gerais, a implementação do bloqueio em duas fases é a seguinte:

1) Imediatamente antes de uma operação *LEIA*(S_i ,*j,k,T)*, o intervalo cobrindo a página *j* é bloqueado para *T*.

2) Os intervalos bloqueados só serão desbloqueados quando a transação terminar e executar *FECHE*.

Mais precisamente, as operações executariam bloqueios da seguinte forma:

ABRE não executa bloqueios.

LEIA bloqueia o intervalo cobrindo a página a ser lida para a transação.

GRAVE não executa bloqueios (*LEIA* já bloqueou os intervalos necessários).

FECHE libera os intervalos mantidos bloqueados.

SALVE mantém os intervalos bloqueados.

RESTAURE libera os intervalos mantidos bloqueados (supõe-se que após esta operação a transação é reiniciada)

CÓPIA não executa bloqueios.

INICIALIZAÇÃO implicitamente bloqueia todos os intervalos pois é executada atomicamente (através do semáforo *ATIVA*) com relação às outras transações.

TERMINACAO: mesma observação.

Além do bloqueio aos dados, a implementação anterior tem que ser modificada para armazenar o estado corrente das várias transações em andamento (e não apenas de uma, como era o caso quando assumíamos execução seqüencial). As variáveis *C* e *D* deverão ser substituídas por um mecanismo de atas, bem mais simples do que o descrito na Seção 9.1, apenas com o propósito de registrar de forma recuperável o estado das várias transações em andamento. A operação *RECUPERE* deve ser modificada para desfazer os efeitos das transações que não atingiram a primeira fase do protocolo bifásico ou que estão sendo canceladas, e completar o processamento daquelas que estavam executando *FECHE* no momento da falha. As imagens criadas por transações que atingiram a primeira fase do protocolo bifásico deverão permanecer intactas. Além disto, a operação *RECUPERE* deverá reconstruir a tabela de bloqueios. Para tal, basta inspecionar qual transação criou a versão transiente de cada página. Se a transação atingiu a primeira fase do protocolo bifásico, então a página deverá ser rebloqueada para a transação.

A descarga dinâmica deverá também ser modificada para armazenar em *E* a menor senha de alguma transação que estava em progresso quando a descarga foi iniciada. Para que esta alteração funcione corretamente, as senhas deverão ser dadas baseadas na hora em que a transação iniciou. Assim, se $VC_i(j) = (s,t)$ e $t > E$, então *s* conterá uma imagem certificada sem cópia como no caso de execução seqüencial.

Pode-se provar que esta implementação de bloqueio em duas fases local está correta. Os pontos mais importantes são os seguintes. Primeiramente, pode-se mostrar que toda execução concorrente das transações é serializável com relação aos blocos de V_i e aos setores, ou seja, com relação às estruturas que definem o estado transiente. Logo, anomalias de sincronização não ocorrerão com relação aos dados. O mecanismo de controle de integridade para recuperação de falhas primárias continua operando corretamente pois a assertiva *P* (vide Seção 9.2.3.3) continua sendo um invariante das operações. No entanto, a noção de estado certificado tem que ser alterada da seguinte forma. Anteriormente, o estado certificado no instante *t* era definido como o estado do banco de dados criado pela última transação que terminou antes de *t*. Como as transações eram executadas seqüencialmente, este estado, na verdade, era aquele obtido pela execução serial das transações que terminaram corretamente até *t*, na ordem em que terminaram. Definiremos então o estado certificado no instante *t* para uma execução concorrente *E* como o estado do banco obtido pela execução serial *S/* das transações que terminaram até o instante *t* na mesma ordem em que terminaram em *E*.

NOTAS BIBLIOGRÁFICAS

Attar, Bernstein e Goodman[1981] propõem uma metodologia uniforme para tratar os problemas locais de iniciação, controle de integridade e obtenção de cópias em memória dormente. Usam conceitos de serialização, atas e bloqueio em duas fases, apresentando uma análise algo rigorosa da correção dos métodos abordados. Gray[1980] tenta formalizar os vários conceitos presentes nos mecanismos usados para manutenção de consistência e também para recuperação do sistema em face de falhas primárias. Concentra-se em sistemas centralizados, na maior parte, apresentando algumas idéias a respeito da probabilidade de bloqueios e eventos correlatos. Gray et all.[1979] é um tutorial bem escrito. Cobre o mecanismo de recuperação do Sistema R, apresentando suas partes mais importantes e avaliando os custos, erros de concepção e as qualidades do sistema. Um refinamento possível de ser incorporado ao mecanismo de atas é descrito em Hadzilacos[1982]. A idéia básica consistiria em obter-se pontos de retorno para

transações incompletas de modo que bastasse desfazer seus efeitos apenas até estes pontos de retorno. A idéia é formalizada e, em seguida, o problema de se escolher para que pontos de retorno devem as transações incompletas ser revertidas é analisado. O problema da escolha dos pontos de retorno se torna interessante quando há interdependências entre as ações de várias transações que executam concorrentemente. Uma descrição muito boa dos paradigmas de cancelar/confirmar transações aparece em Hammer e Shipman[1979]. A problemática é abordada no contexto do sistema SSD-1 e os mecanismos envolvidos são descritos em detalhe, incluindo o problema de se manter relógios globais sincronizados por todo o SGBD. Lindsay[1980] contém uma descrição, com algum detalhe, dos mecanismos de atas. Já Kohler[1981] descreve e classifica várias técnicas usadas em controle de integridade, dedicando largo espaço à questão de como se construir funções internas que se mostrem seguras a usuários operando em camadas superiores do sistema. Stonebraker[1980] contém uma descrição dos algorítmos usados para controle de integridade no sistema INGRES, bem como uma comparação destes algoritmos com outros descritos na literatura. Uma tentativa de formalizar o estudo da função de controle de integridade aparece em Skeen e Stonebraker[1983]. A proposta usa máquinas de estados finitos para modelar o ambiente de um SGBDD, incluindo a rede de comunicação de dados. Apresenta alguns resultados negativos interessantes a respeito da robustez de classes de protocolos. O método de imagens virtuais discutido na Seção 9.2 é uma adaptação para um ambiente distribuído em que transações são executadas concorrentemente do método descrito por Lorie [1977]. Os problemas relativos a controle de concorrência em dois níveis são abordados em Goodman e Lay [1983] e Lynch [1983].