Elmasri • Navathe

Sistemas de banco de dados

6ª edição

ALWAYS LEARNING

PEARSON

capítulo 21

Introdução aos conceitos e teoria de processamento de transações

O conceito de transação oferece um mecanismo para descrever unidades lógicas de processamento de banco de dados. Os **sistemas de processamento de transação** são sistemas com grandes bancos de dados e centenas de usuários simultâneos que executam transações de banco de dados. Alguns exemplos desses sistemas incluem reservas aéreas, sistemas bancários, processamento de cartão de crédito, compras on-line, mercados de ações, caixas de supermercado e muitas outras aplicações. Esses sistemas exigem alta disponibilidade e tempo de resposta rápido para centenas de usuários simultâneos. Neste capítulo, apresentamos os conceitos necessários em sistemas de processamento de transação. Definimos o conceito de uma transação, que é usado para representar uma unidade lógica de processamento de banco de dados que deve ser concluída por inteiro para garantir a exatidão. Uma transação normalmente é implementada por um programa de computador, que inclui comandos de banco de dados como recuperações, inserções, exclusões e atualizações. Apresentamos algumas das técnicas básicas para programação de banco de dados nos capítulos 13 e 14.

Neste capítulo, focalizamos os conceitos básicos e a teoria necessários para garantir a execução correta das transações. Discutimos o problema de controle de concorrência, que ocorre quando várias transações submetidas por diversos usuários interferem umas com as outras de uma maneira que produz resultados incorretos. Também discutimos os problemas que podem ocorrer quando as transações falham, e como o sistema de banco de dados pode se recuperar de diversos tipos de falhas.

Este capítulo é organizado da seguinte forma: a Seção 21.1 discute informalmente por que o controle de concorrência e recuperação são necessários em um sistema de banco de dados. A Seção 21.2 define o termo *transação* e discute conceitos adicionais relacionados ao processamento de transação nos sistemas de banco de dados. A Seção 21.3 apresenta as propriedades importantes de atomicidade, a preservação de consistência, o isolamento e a durabilidade ou permanência — chamadas propriedades ACID —, que são consideradas desejáveis nos sistemas de processamento de transação. A Seção 21.4 apresenta o conceito de schedules (ou históricos) de execução de transações e caracteriza sua *recuperabilidade*. A Seção 21.5 discute a noção de *serialização* da execução concorrente da transação, que pode ser usada para definir as sequências de execução corretas (ou schedules) de transações simultâneas. Na Seção 21.6, apresentamos alguns dos comandos que dão suporte ao conceito de transação em SQL. No final do capítulo há um resumo.

Os dois capítulos seguintes fornecem mais detalhes sobre os métodos e técnicas reais usadas para dar suporte ao processamento de transação. O Capítulo 22 oferece uma visão geral dos protocolos básicos de controle de concorrência e o Capítulo 23 introduz as técnicas de recuperação.

21.1 Introdução ao processamento de transações

Nesta seção, discutimos os conceitos de execução concorrente de transações e recuperação de transações com falhas. A Seção 21.1.1 compara sistemas de banco de dados de monousuário e multiusuários, e demonstra como a execução simultânea de transações pode ocorrer nos sistemas multiusuários. A Seção 21.1.2 define o conceito de transação e apresenta um modelo simples de execução de transação base-

ado em operações de leitura e gravação de banco de dados. Esse modelo é usado como base para definir e formalizar os conceitos de controle de concorrência e recuperação. A Seção 21.1.3 utiliza exemplos informais para mostrar por que as técnicas de controle de concorrência são necessárias em sistemas multiusuários. Por fim, a Seção 21.1.4 discute por que são necessárias técnicas para lidar com a recuperação do sistema e falhas de transação, discutindo as diferentes maneiras como as transações podem falhar quando são executadas.

21.1.1 Sistemas de monousuário *versus* multiusuário

Um critério para classificar um sistema de banco de dados é de acordo com o número de usuários que podem usar o sistema **simultaneamente**. Um SGBD é **monousuário** se no máximo um usuário de cada vez pode utilizar o sistema, e é **multiusuário** se muitos usuários puderem fazê-lo — e, portanto, acessar o banco de dados — simultaneamente. Os SGBDs monousuário são principalmente restritos a sistemas de computador pessoal; a maioria dos outros SGBDs é multiusuário. Por exemplo, um sistema de reservas aéreas é acessado por centenas de agentes de viagens e funcionários de reserva de maneira simultânea. Os sistemas de banco de dados usados em bancos, agências de seguros, mercado de ações, supermercados e muitas outras aplicações são de multiusuários. Nesses sistemas, centenas ou milhares de usuários normalmente estão operando no banco de dados ao submeter transações ao sistema ao mesmo tempo.

Multiusuários podem acessar os bancos de dados — e usar sistemas de computação — simultaneamente devido ao conceito da **multiprogramação**, que permite que o sistema operacional do computador execute vários programas — ou **processos** — ao mesmo tempo. Uma única unidade central de processamento (CPU) pode executar apenas, e no máximo, um processo de cada vez. Porém, **sistemas operacionais de multiprogramação** executam alguns comandos de um processo, depois suspendem esse processo e executam alguns comandos do processo seguinte, e assim por diante. Um processo é retomado no ponto em que foi suspenso sempre que chega sua vez de usar a CPU novamente. Assim, a execução simultânea dos processo é, na realidade, **intercalada**, conforme ilustrado na Figura 21.1, que mostra dois processos, A e B, executando simultaneamente em um padrão intercalado. A intercalação mantém a CPU ocupada quando um processo exige uma operação de entrada ou saída (E/S), como a leitura de um bloco do disco. A CPU é trocada para executar outro processo, em vez de permanecer ociosa durante o tempo da E/S. A intercalação também impede que um processo longo atrase os demais processos.

Se o sistema de computação tiver múltiplos processadores de hardware (CPUs), o **processamento paralelo** de vários processos é possível, conforme ilustrado pelos processos C e D da Figura 21.1. A maior parte da teoria referente ao controle de concorrência nos bancos de dados é desenvolvida em relação à **concorrência intercalada**, de modo que, para o restante deste capítulo, assumiremos esse modelo. Em um SGBD multiusuário, os itens de dados armazenados são os recursos principais que podem ser acessados simultaneamente por usuários ou programas de aplicação interativos, que estão sempre recuperando informações e modificando o banco de dados.

21.1.2 Transações, itens de banco de dados, operações de leitura e gravação e buffers do SGBD

Uma **transação** é um programa em execução que forma uma unidade lógica de processamento de banco de dados. Ela inclui uma ou mais operações de acesso ao banco de dados — estas podem incluir operações de inserção, exclusão, modificação ou recuperação. As operações de banco de dados que

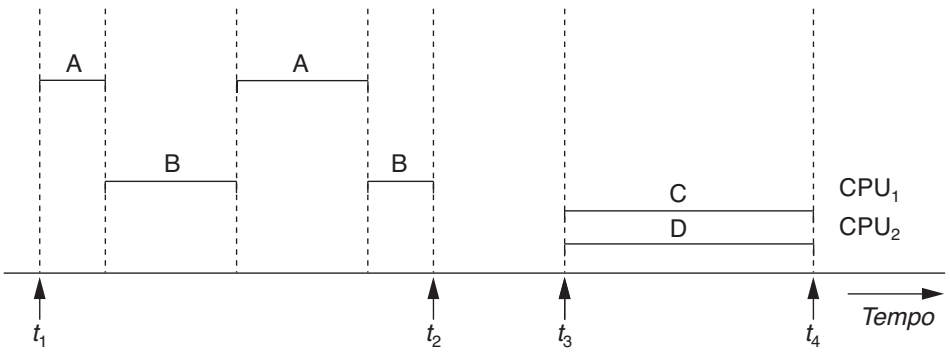


Figura 21.1
Processamento intercalado *versus* processamento paralelo de transações simultâneas.

formam uma transação podem ser embutidas em um programa de aplicação ou podem ser especificadas interativamente por meio de uma linguagem de consulta de alto nível, como a SQL. Um modo de especificar os limites de transação é determinando pelas instruções explícitas **begin transaction** e **end transaction** em um programa de aplicação; nesse caso, todas as operações de acesso ao banco de dados entre os dois são consideradas formando uma transação. Um único programa de aplicação pode conter mais de uma transação se tiver vários limites de transação. Se as operações de banco de dados em uma transação não atualizarem o banco de dados, mas apenas recuperarem dados, a transação é chamada de **transação somente de leitura**; caso contrário, ela é conhecida como **transação de leitura-gravação**.

O *modelo de banco de dados* utilizado para apresentar conceitos de processamento de transação é muito simples em comparação com modelos de dados que discutimos anteriormente no livro, como o modelo relacional ou o modelo de objeto. Um **banco de dados** é basicamente representado como uma coleção de *itens de dados nomeados*. O tamanho de um item de dados é chamado de sua **granularidade**. Um **item de dados** pode ser um *registro de banco de dados*, mas também pode ser uma unidade maior, como um *bloco de disco* inteiro, ou mesmo uma unidade menor, como um *valor de campo* (*atributo*) individual de algum registro no banco de dados. Os conceitos de processamento de transação que discutimos são independentes da granularidade (tamanho) do item de dados e se aplicam a itens de dados em geral. Cada item de dados tem um *nome único*, mas esse nome em geral não é usado pelo programador; em vez disso, ele é apenas um meio para *identificar exclusivamente cada item de dados*. Por exemplo, se a granularidade do item de dados for um bloco de disco, então o endereço do bloco de disco pode ser utilizado como o nome do item de dados. Ao usar esse modelo de banco de dados simplificado, as operações básicas de acesso ao banco de dados que uma transação pode incluir são as seguintes:

- **read_item(*X*).** Lê um item do banco de dados chamado *X* para uma variável do programa. Para simplificar nossa notação, consideramos que *a variável de programa também é chamada X*.
- **write_item(*X*).** Grava o valor da variável de programa *X* no item de banco de dados chamado *X*.

Conforme discutimos no Capítulo 17, a unidade básica de transferência de dados do disco para a memória principal é um bloco. A execução de um comando read_item(*X*) inclui as seguintes etapas:

1. Ache o endereço do bloco de disco que contém o item *X*.
2. Copie esse bloco de disco para um buffer na memória principal (se esse bloco de disco ainda não estiver em algum buffer da memória principal).
3. Copie o item *X* do buffer para a variável de programa chamada *X*.

A execução de um comando write_item(*X*) inclui as seguintes etapas:

1. Ache o endereço do bloco de disco que contém o item *X*.
2. Copie esse bloco de disco para um buffer na memória principal (se esse bloco de disco ainda não estiver em algum buffer da memória principal).
3. Copie o item *X* da variável de programa chamada *X* para o local correto no buffer.
4. Armazene o bloco atualizado do buffer de volta no disco (imediatamente ou em algum momento posterior).

É a etapa 4 que de fato atualiza o banco de dados no disco. Em alguns casos, o buffer não é imediatamente armazenado no disco, caso mudanças adicionais tenham de ser feitas no buffer. Em geral, a decisão sobre quando armazenar um bloco de disco modificado cujo conteúdo está em um buffer da memória principal é tratado pelo gerenciador de recuperação do SGBD em cooperação com o sistema operacional subjacente. O SGBD manterá na **cache do banco de dados** uma série de **buffers de dados** na memória principal. Cada buffer costuma manter o conteúdo de um bloco de disco do banco de dados, que contém alguns dos itens de banco de dados que estão sendo processados. Quando esses buffers estão todos ocupados, e blocos de disco de banco de dados adicionais devem ser copiados para a memória, alguma política de substituição de buffer é utilizada para escolher quais buffers atuais devem ser substituídos. Se um buffer escolhido tiver de ser modificado, ele precisa ser gravado de volta no disco antes de ser reutilizado.[1]

Uma transação inclui operações read_item e write_item para acessar e atualizar o banco de dados. A Figura 21.2 mostra exemplos de duas transações muito simples. O **conjunto de leitura** de uma tran-

[1] Não discutiremos as políticas de substituição de buffer aqui, pois elas normalmente são abordadas em livros-texto sobre sistemas operacionais.

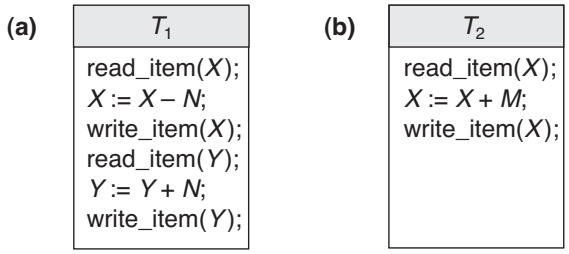


Figura 21.2
Duas transações de exemplo. (a) Transação T_1. (b) Transação T_2.

sação é o conjunto de todos os itens que a transação lê, e o **conjunto de gravação** é o conjunto de todos os itens que a transação grava. Por exemplo, o conjunto de leitura de T_1 da Figura 21.2 é {*X*, *Y*} e seu conjunto de gravação também é {*X*, *Y*}.

Os mecanismos de controle de concorrência e recuperação tratam principalmente dos comandos de banco de dados em uma transação. As transações submetidas pelos diversos usuários podem ser executadas simultaneamente, acessar e atualizar os mesmos itens de banco de dados. Se essa execução simultânea for *descontrolada*, ela pode ocasionar problemas, como um banco de dados inconsistente. Na próxima seção, apresentamos de maneira informal alguns dos problemas que podem ocorrer.

21.1.3 Por que o controle de concorrência é necessário

Vários problemas podem acontecer quando transações simultâneas são executadas de uma maneira descontrolada. Ilustramos alguns desses problemas ao nos referirmos a um banco de dados de reservas aéreas muito simplificado, em que um registro é armazenado para cada voo. Cada registro inclui o *número de assentos reservados* nesse voo como um *item de dados nomeado (identificável exclusivamente)*, entre outras informações. A Figura 21.2(a) mostra uma transação T_1 que *transfere N* reservas de um voo cujo número de assentos reservados é armazenado no item de banco de dados chamado *X* para outro voo cujo número de assentos reservados é armazenado no item de banco de dados chamado *Y*. A Figura 21.2(b) mostra uma transação mais simples T_2 que apenas *reserva M* assentos no primeiro voo (*X*) referenciados na transação T_1.[2] Para simplificar nosso exemplo, não mostramos partes adicionais das transações, como a verificação de um voo ter assentos suficientes disponíveis antes de reservar assentos adicionais.

Quando um programa de acesso ao banco de dados é escrito, ele tem o número do voo, a data do voo e o número de assentos a serem reservados como parâmetros; logo, o mesmo programa pode ser utilizado para executar *muitas transações diferentes*, cada uma com um número diferente de voo, data e número de assentos a serem reservados. Para fins de controle de concorrência, uma transação é uma *execução em particular* de um programa em uma data, voo e número de assentos específicos. Na Figura 21.2(a) e (b), as transações T_1 e T_2 são *execuções específicas* dos programas que se referem aos voos específicos cujos números de assentos são armazenados nos itens de dados *X* e *Y* no banco de dados. Em seguida, discutimos os tipos de problemas que podemos encontrar com essas duas transações simples se elas forem executadas simultaneamente.

O problema da atualização perdida. Esse problema ocorre quando duas transações que acessam os mesmos itens do banco de dados têm suas operações intercaladas de modo que isso torna o valor de alguns itens do banco de dados incorreto. Suponha que as transações T_1 e T_2 sejam submetidas aproximadamente ao mesmo tempo, e suponha que suas operações sejam intercaladas, como mostra a Figura 21.3(a); então, o valor final do item *X* é incorreto porque T_2 lê o valor de *X* antes que T_1 o mude no banco de dados, e, portanto, o valor atualizado resultante de T_1 é perdido. Por exemplo, se *X* = 80 no início (originalmente, havia 80 reservas no voo), *N* = 5 (T_1 transfere cinco reservas de assento do voo correspondente a *X* para o voo correspondente a *Y*), e *M* = 4 (T_2 reserva quatro assentos em *X*), o resultado final deveria ser *X* = 79. No entanto, na intercalação de operações mostrada na Figura 21.3(a), ele é *X* = 84, pois a atualização em T_1 que removeu os cinco assentos de *X* foi perdida.

O problema da atualização temporária (ou leitura suja). Esse problema ocorre quando uma transação atualiza um item do banco de dados e depois a transação falha por algum motivo (ver Seção 21.1.4). Nesse meio-tempo, o item atualizado é acessado (lido) por outra transação, antes de ser alterado de volta para seu valor original. A Figura 21.3(b) mostra um exemplo em que T_1 atualiza o item *X* e então falha antes de terminar, de modo que o sistema deve mudar *X* de volta para seu valor original. Contudo, antes que ele possa fazer isso, a transação T_2 lê o valor temporário de *X*, que não será gravado permanentemente no

[2] Um exemplo semelhante, mais utilizado, considera um banco de dados bancário, com uma transação realizando uma transferência de fundos da conta *X* para a conta *Y* e outra transação realizando um depósito na conta *X*.

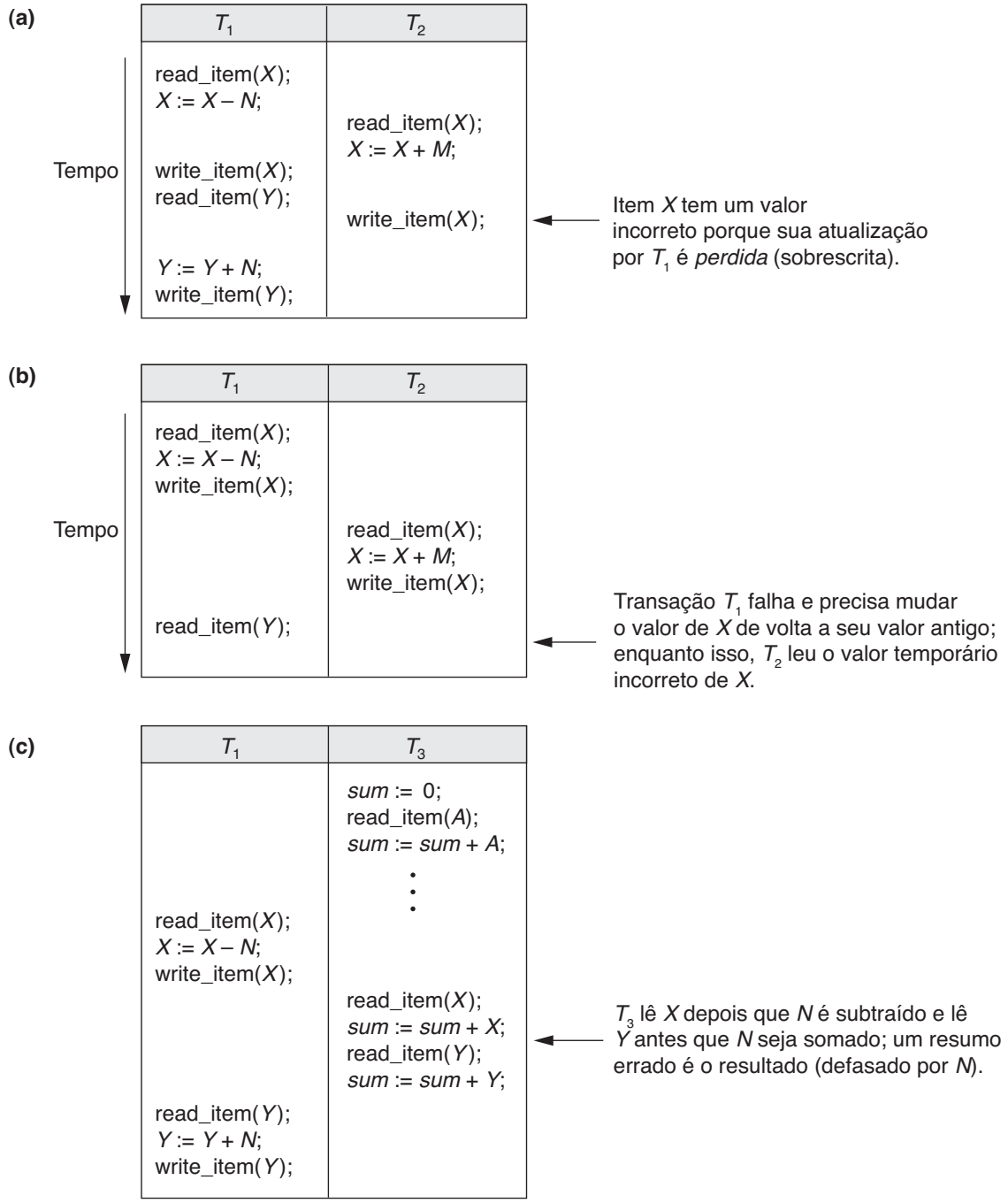


Figura 21.3
Alguns problemas que ocorrem quando a execução simultânea não é controlada. (a) O problema da atualização perdida. (b) O problema da atualização temporária. (c) O problema do resumo incorreto.

banco de dados devido à falha de T_1. O valor do item X que foi lido por T_2 é chamado de dado sujo, pois foi criado por uma transação que não foi concluída nem confirmada; portanto, esse problema também é conhecido como problema de leitura suja.

O problema do resumo incorreto. Se uma transação está calculando uma função de resumo de agregação em uma série de itens de banco de dados, enquanto outras transações estão atualizando alguns desses itens, a função de agregação pode calcular alguns valores antes que eles sejam atualizados e outros, depois que eles forem atualizados. Por exemplo, suponha que uma transação T_3 esteja calculando o número total de reservas em todos os voos; enquanto isso, a transação T_1 está sendo executada. Se a intercalação de operações mostrada na Figura 21.3(c) acontecer, o resultado de T_3 estará defasado por uma quantidade N, pois T_3 lê o valor de X após N assentos terem sido

subtraídos dele, mas lê o valor de *Y* antes que esses *N* assentos tenham sido acrescentados.

O problema da leitura não repetitiva. Outro problema que pode acontecer é chamado de leitura não repetitiva, em que uma transação *T* lê o mesmo item duas vezes e o item é alterado por outra transação *T'* entre as duas leituras. Logo, *T* recebe valores diferentes para suas duas leituras do mesmo item. Isso pode acontecer, por exemplo, se durante uma transação de reserva aérea, um cliente consultar a disponibilidade de assento em vários voos. Quando o cliente decide sobre um voo em particular, a transação então lê o número de assentos nesse voo pela segunda vez antes de completar a reserva, e pode acabar lendo um valor diferente para o item.

21.1.4 Por que a recuperação é necessária

Sempre que uma transação é submetida a um SGBD para execução, o sistema é responsável por garantir que todas as operações na transação sejam concluídas com sucesso e seu efeito seja registrado permanentemente no banco de dados, ou que a transação não tenha qualquer efeito no banco de dados ou quaisquer outras transações. No primeiro caso, a transação é considerada **confirmada** (committed), ao passo que, no segundo caso, a transação é **abortada**. O SGBD não deve permitir que algumas operações de uma transação *T* sejam aplicadas ao banco de dados enquanto outras operações de *T* não o são, pois a *transação inteira* é uma unidade lógica de processamento de banco de dados. Se a transação **falhar** depois de executar algumas de suas operações, mas antes de executar todas elas, as operações já executadas precisam ser desfeitas e não têm efeito duradouro.

Tipos de falhas. As falhas geralmente são classificadas como falhas de transação, sistema e mídia. Existem vários motivos possíveis para uma transação falhar no meio da execução:

1. **Uma falha do computador (falha do sistema).** Um erro de hardware, software ou rede no sistema de computação durante a execução da transação. Falhas do hardware normalmente são falhas de mídia — por exemplo, uma falha na memória principal.
2. **Um erro de transação ou do sistema.** Alguma operação na transação pode fazer que esta falhe, como um estouro de inteiro ou divisão por zero. A falha da transação também pode ocorrer devido a valores de parâmetro errôneos ou a um erro lógico de programação.[3] Além disso, o usuário pode interromper a transação durante sua execução.
3. **Erros locais ou condições de exceção detectadas pela transação.** Durante a execução da transação, podem ocorrer certas condições que necessitam de cancelamento da transação. Por exemplo, os dados da transação podem não ser encontrados. Uma condição de exceção,[4] como um saldo de conta insuficiente em um banco de dados bancário, pode fazer que uma transação, como um saque, seja cancelada. Essa exceção poderia ser programada na própria transação, e nesse caso não seria considerado uma falha da transação.
4. **Imposição de controle de concorrência.** O método de controle de concorrência (ver Capítulo 22) pode decidir abortar uma transação porque ela viola a serialização (ver Seção 21.5), ou pode abortar uma ou mais transações para resolver um estado de deadlock entre várias transações (ver Seção 22.1.3). As transações abortadas devido a violações de serialização ou deadlock em geral são reiniciadas automaticamente em outro momento.
5. **Falha de disco.** Alguns blocos de disco podem perder seus dados devido a um defeito de leitura, gravação ou por causa de uma falha da cabeça de leitura/gravação. Isso pode acontecer durante uma operação de leitura ou gravação da transação.
6. **Problemas físicos e catástrofes.** Isso se refere a uma lista sem fim de problemas que incluem falha de energia ou de ar-condicionado, incêndio, roubo, sabotagem, regravação de discos ou fitas por engano e montagem da fita errada pelo operador.

As falhas dos tipos 1, 2, 3 e 4 são mais comuns do que aquelas dos tipos 5 ou 6. Sempre que ocorre uma falha dos tipos de 1 a 4, o sistema precisa manter informações suficientes para recuperar-se rapidamente da falha. A falha de disco ou outras falhas catastróficas de tipo 5 ou 6 não acontecem com frequência; se ocorrerem, a recuperação é uma tarefa importante. Discutiremos sobre a recuperação de falhas no Capítulo 23.

O conceito de transação é fundamental para muitas técnicas de controle de concorrência e recu-

[3] Em geral, uma transação deve ser testada completamente para garantir que não tenha quaisquer bugs (erros lógicos de programação).

[4] Condições de exceção, se programadas corretamente, não constituem falhas de transação.

peração de falhas.

21.2 Conceitos de transação e sistema

Nesta seção, discutimos conceitos adicionais relevantes ao processamento de transação. A Seção 21.2.1 descreve os diversos estados em que uma transação pode estar, e discute outras operações necessárias no processamento de transação. A Seção 21.2.2 discute o log do sistema, que mantém informações sobre transações e itens de dados que serão necessários para recuperação. A Seção 21.2.3 descreve o conceito de pontos de confirmação das transações, e por que eles são importantes no processamento da transação.

21.2.1 Estados de transação e operações adicionais

Uma transação é uma unidade atômica de trabalho, que deve ser concluída totalmente ou não ser feita de forma alguma. Para fins de recuperação, o sistema precisa registrar quando cada transação começa, termina e confirma ou aborta (ver Seção 21.2.3). Portanto, o gerenciador de recuperação do SGBD precisa acompanhar as seguintes operações:

- BEGIN_TRANSACTION. Esta marca o início da execução da transação.
- READ ou WRITE. Estas especificam operações de leitura ou gravação nos itens do banco de dados que são executados como parte de uma transação.
- END_TRANSACTION. Esta especifica que operações de transação READ e WRITE terminaram e marca o final da execução da transação. Porém, nesse ponto pode ser necessário verificar se as mudanças introduzidas pela transação podem ser permanentemente aplicadas ao banco de dados (confirmadas) ou se a transação precisa ser abortada, pois viola a serialização (ver Seção 21.5) ou por algum outro motivo.
- COMMIT_TRANSACTION. Esta sinaliza um *final bem-sucedido* da transação, de modo que quaisquer mudanças (atualizações) executadas pela transação podem ser seguramente **confirmadas** (committed) ao banco de dados e não serão desfeitas.
- ROLLBACK (ou ABORT). Esta operação sinaliza que a transação *foi encerrada sem sucesso*, de modo que quaisquer mudanças ou efeitos que a transação possa ter aplicado ao banco de dados precisam ser **desfeitos**.

A Figura 21.4 mostra um diagrama de transição de estado que ilustra como uma transação percorre seus estados de execução. Uma transação entra em um **estado ativo** imediatamente após iniciar a execução, onde pode executar suas operações READ e WRITE. Quando a transação termina, ela passa para o **estado parcialmente confirmado**. Nesse ponto, alguns protocolos de recuperação precisam garantir que uma falha no sistema não resultará em uma incapacidade de registrar as mudanças da transação permanentemente (em geral, gravando mudanças no log do sistema, discutido na próxima seção).[5] Quando essa verificação é bem-sucedida, diz-se que a transação alcançou seu ponto de confirmação e ela entra no **estado confirmado**. Os pontos de confirmação serão discutidos com mais detalhes na Seção 21.2.3. Quando uma transação é confirmada, ela concluiu sua execução com sucesso e todas as suas mudanças

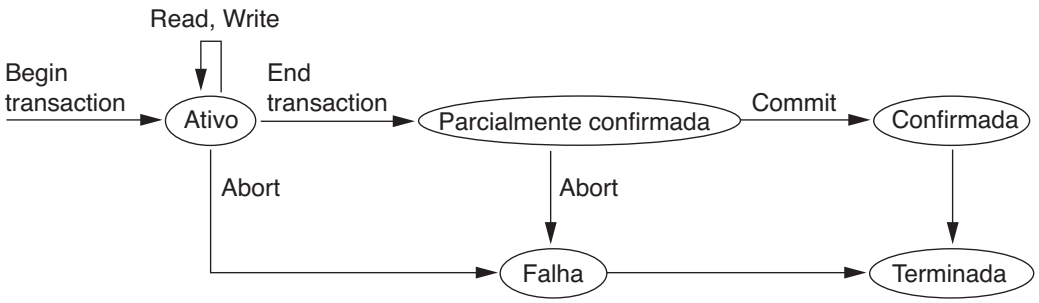


Figura 21.4
Diagrama de transição de estado ilustrando os estados para execução da transação.

[5] O controle de concorrência otimista (ver Seção 22.4) também exige que certas verificações sejam feitas nesse ponto para garantir que a transação não interfira em outras transações em execução.

[6] O log às vezes é chamado de *diário do SGBD*.

precisam ser gravadas permanentemente no banco de dados, mesmo que haja uma falha no sistema.

Entretanto, uma transação pode ir para o **estado de falha** se uma das verificações falhar ou se a transação for abortada durante seu estado ativo. A transação pode então ter de ser cancelada para desfazer o efeito de suas operações WRITE no banco de dados. O **estado terminado** corresponde à transação que sai do sistema. A informação da transação que é mantida nas tabelas do sistema enquanto a transação estava rodando é removida quando esta termina. As transações com falha ou abortadas podem ser *reiniciadas* depois — seja de maneira automática ou depois de serem submetidas outra vez pelo usuário — como transações totalmente novas.

21.2.2 O log do sistema

Para poder recuperar-se de falhas que afetam transações, o sistema mantém um **log**[6] para registrar todas as operações de transação que afetam os valores dos itens de banco de dados, bem como outras informações de transação que podem ser necessárias para permitir a recuperação de falhas. O log é um arquivo sequencial, apenas para inserção, que é mantido no disco, de modo que não é afetado por qualquer tipo de falha, exceto por falha de disco ou catastrófica. Normalmente, um (ou mais) buffers de memória mantêm a última parte do arquivo de log, de modo que as entradas do log são primeiro acrescentadas ao buffer da memória principal. Quando o **buffer de log** é preenchido, ou quando ocorrem certas condições, o buffer de log é *anexado ao final do arquivo de log no disco*. Além disso, o arquivo de log do disco é periodicamente copiado para arquivamento (fita), para proteger contra falhas catastróficas. A seguir estão os tipos de entradas — chamados **registros de log** — que são gravados para o arquivo de log e a ação correspondente para cada registro de log. Nessas entradas, *T* refere-se a uma **id de transação** exclusiva que é gerada automaticamente pelo sistema para cada transação e que é usada para identificar cada transação.

1. [**start_transaction, *T***]. Indica que a transação *T* iniciou sua execução.
2. [**write_item, *T*, *X*, *valor_antigo*, *valor_novo***]. Indica que a transação *T* mudou o valor do item do banco de dados *X* de *valor_antigo* para *valor_novo*.
3. [**read_item, *T*, *X***]. Indica que a transação *T* leu o valor do item de banco de dados *X*.
4. [**commit, *T***]. Indica que a transação *T* foi concluída com sucesso, e afirma que seu efeito pode ser confirmado (registrado permanentemente) no banco de dados.
5. [**abort, *T***]. Indica que a transação *T* foi abortada.

Protocolos para recuperação que evitam propagação de rollbacks (ver Seção 21.4.2) — que incluem quase todos os protocolos práticos — *não exigem que* operações READ sejam gravadas no log do sistema. Contudo, se o log também for usado para outras finalidades — como auditoria (mantendo registro de todas as operações do banco de dados) —, então essas entradas podem ser incluídas. Além disso, alguns protocolos de recuperação que exigem entradas WRITE mais simples só incluem um *valor_novo* ou *valor_antigo* em vez de incluir ambos (ver Seção 21.4.2).

Observe que não estamos assumindo que todas as mudanças permanentes no banco de dados ocorrem nas transações, de modo que a noção de recuperação de uma falha de transação equivale a desfazer ou refazer operações de transação individualmente com base no log. Se o sistema falhar, podemos recuperar para um estado coerente do banco de dados ao examinar o log e usar uma das técnicas descritas no Capítulo 23. Como o log contém um registro de cada operação WRITE que muda o valor de algum item do banco de dados, é possível **desfazer** o efeito dessas operações WRITE de uma transação *T* rastreando o log de volta e retornando todos os itens alterados por uma operação WRITE de *T* a seus *valores_antigos*. Também pode ser necessário **refazer** uma operação se uma transação tiver suas atualizações registradas no log, mas houver uma falha antes que o sistema possa estar certo de que todos esses *novos_valores* tenham sido gravados no banco de dados real em disco com base nos buffers da memória principal.[7]

21.2.3 Ponto de confirmação de uma transação

Uma transação *T* alcança seu **ponto de confirmação** quando todas as suas operações que acessam o banco de dados tiverem sido executadas com sucesso *e* o efeito de todas as operações de transação no banco de dados tiverem sido registradas no log. Além do ponto de confirmação, a transação é considerada **confirmada**, e seu efeito deve ser *registrado permanentemente* no banco de dados. A transação então grava um registro de confirmação [commit, *T*] no log. Se houver uma falha no sistema, podemos pesquisar de volta no log para todas as transações *T* que gravaram um registro [start_transaction, *T*] no log, mas

[7] Desfazer e refazer são operações discutidas de maneira mais completa no Capítulo 23.

ainda não gravaram seu registro [commit, *T*]. Essas transações podem ter de ser *descartadas (rollback)* para *desfazer seu efeito* sobre o banco de dados durante o processo de recuperação. As transações que gravaram seu registro de confirmação no log também devem ter gravado todas as suas operações WRITE no log, de modo que seu efeito no banco de dados possa ser *refeito* com base nos registros de log.

Observe que o arquivo de log precisa ser mantido no disco. Conforme discutimos no Capítulo 17, a atualização de um arquivo do disco envolve copiar o bloco apropriado do arquivo para um buffer na memória principal, atualizar o buffer na memória principal e copiar o buffer para o disco. É comum manter um ou mais blocos do arquivo de log nos buffers da memória principal, chamado **buffer de log**, até que eles sejam preenchidos com entradas de log e, depois, gravá-los de volta ao disco apenas uma vez, ao invés de gravar em disco toda vez que uma entrada de log é acrescentada. Isso economiza o overhead de várias gravações de disco do mesmo buffer do arquivo de log. No momento de uma falha do sistema, apenas as entradas de log que foram *gravadas de volta para o disco* são consideradas no processo de recuperação, pois o conteúdo da memória principal pode ser perdido. Logo, *antes* que uma transação alcance seu ponto de confirmação, qualquer parte do log que ainda não tenha sido gravada no disco deve agora sê-lo. Esse processo é chamado de **gravação forçada** do buffer de log antes da confirmação de uma transação.

21.3 Propriedades desejáveis das transações

As transações devem possuir várias propriedades, normalmente chamadas propriedades **ACID;** elas devem ser impostas pelos métodos de controle de concorrência e recuperação do SGBD. A seguir estão as propriedades ACID:

- **Atomicidade.** Uma transação é uma unidade de processamento atômica; ela deve ser realizada em sua totalidade ou não ser realizada de forma alguma.
- **Preservação da consistência.** Uma transação deve preservar a consistência, significando que, se ela for completamente executada do início ao fim sem interferência de outras transações, deve levar o banco de dados de um estado consistente para outro.
- **Isolamento.** Uma transação deve parecer como se fosse executada isoladamente de outras transações, embora muitas delas estejam sendo executadas de maneira simultânea. Ou seja, a execução de uma transação não deve ser interferida por quaisquer outras transações que acontecem simultaneamente.
- **Durabilidade ou permanência.** As mudanças aplicadas ao banco de dados pela transação confirmada precisam persistir no banco de dados. Essas mudanças não devem ser perdidas por causa de alguma falha.

A *propriedade de atomicidade* exige que executemos uma transação até o fim. É responsabilidade do *subsistema de recuperação de transação* de um SGBD garantir a atomicidade. Se uma transação não for completada por algum motivo, como uma falha no sistema no meio da execução da transação, a técnica de recuperação precisa desfazer quaisquer efeitos da transação no banco de dados. Por sua vez, as operações de gravação de uma transação confirmada devem ser, por fim, gravadas no disco.

A preservação da *consistência* geralmente é considerada uma responsabilidade dos programadores que escrevem os programas de banco de dados ou do módulo de SGBD que impõe restrições de integridade. Lembre-se de que um **estado de banco de dados** é uma coleção de todos os itens de dados armazenados (valores) no banco de dados em determinado ponto no tempo. Um **estado consistente** do banco de dados satisfaz as restrições especificadas no esquema, bem como quaisquer outras restrições no banco de dados que devem ser mantidas. Um programa de banco de dados deve ser escrito de modo que garanta que, se o banco de dados estiver em um estado consistente antes de executar a transação, ele estará em um estado consistente depois de *concluir* a execução da transação, supondo que *não haja interferência em outras transações*.

A *propriedade de isolamento* é imposta pelo *subsistema de controle de concorrência* do SGBD.[8] Se cada transação não tornar suas atualizações (operações de gravação) visíveis para outras transações até que seja confirmada, uma forma de isolamento é imposta para solucionar o problema da atualização temporária e eliminar rollback em cascata (ver Capítulo 23), mas ela não elimina todos os outros problemas. Tem havido tentativas de definir o **nível de isolamento** de uma transação. Uma transação é considerada como tendo isolamento de nível 0 (zero) se não gravar sobre as leituras sujas das transações de nível mais alto. O isolamento de nível 1 (um) não tem atualizações perdidas, e o isolamento de nível 2 não tem atualizações perdidas ou leituras sujas. Finalmente, o isolamento de nível 3 (também chamado

[8] Discutiremos os protocolos de controle de concorrência no Capítulo 22.

[9] A sintaxe SQL para o nível de isolamento, discutida mais adiante na Seção 21.6, está bastante relacionada a esses níveis.

isolamento verdadeiro) tem, além das propriedades de nível 2, leituras repetitivas.[9]

E por fim, a *propriedade de durabilidade* é a responsabilidade do *subsistema de recuperação* do SGBD. Vamos apresentar o modo como os protocolos de recuperação impõem a durabilidade e a atomicidade na próxima seção, para discutirmos isso com mais detalhes no Capítulo 23.

21.4 Caracterizando schedules com base na facilidade de recuperação

Quando as transações estão executando simultaneamente em um padrão intercalado, então a ordem da execução das operações de todas as diversas transações é conhecida como um **schedule** (ou **histórico**). Nesta seção, primeiro definimos o conceito de schedules e, depois, caracterizamos os tipos de schedules que facilitam a recuperação quando ocorrem falhas. Na Seção 21.5, caracterizamos os schedules em relação à interferência das transações participantes, levando aos conceitos de serialização e schedules serializáveis.

21.4.1 Schedules (históricos) de transações

Um **schedule** (ou **histórico**) S de n transações T_1, T_2, ..., T_n é uma ordenação das operações das transações. As operações das diferentes transações podem ser intercaladas no schedule S. Contudo, para cada transação T_i que participa no schedule S, as operações de T_i em S precisam aparecer na mesma ordem em que ocorrem em T_i. A ordem das operações em S é considerada uma *ordenação total*, significando *que para duas operações quaisquer* no schedule, uma precisa ocorrer antes da outra. É possível teoricamente lidar com schedules cujas operações formam *ordens parciais* (conforme discutiremos mais adiante), mas consideraremos por enquanto a ordenação total das operações em um schedule.

Para fins de recuperação e controle de concorrência, estamos interessados principalmente nas operações read_item e write_item das transações, bem como nas operações commit e abort. Uma notação abreviada para descrever um schedule utiliza os símbolos b, r, w, e, c e a para as operações begin_transaction, read_item, write_item, end_transaction, commit e abort, respectivamente, e acrescenta como um *subscrito* a id da transação (número da transação) a cada operação no schedule. Nessa notação, o item de banco de dados X que é lido ou gravado segue as operações r e w entre parênteses. Em alguns schedules, só mostraremos as operações *read* e *write*, enquanto em outros, mostraremos todas as operações. Por exemplo, o schedule na Figura 21.3(a), que chamaremos de S_a, pode ser escrito da seguinte forma nessa notação:

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

De modo semelhante, o schedule para a Figura 21.3(b), que chamamos S_b, pode ser escrito da seguinte forma, se considerarmos que a transação T_1 foi cancelada após sua operação read_item(Y):

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Duas operações em um schedule são consideradas como entrando em **conflito** se satisfizerem a todas as três condições a seguir: (1) elas pertencem a *diferentes transações*; (2) elas acessam o *mesmo item* X; e (3) *pelo menos uma* das operações é um write_item(X). Por exemplo, no schedule S_a, as operações $r_1(X)$ e $w_2(X)$ estão em conflito, assim como as operações $r_2(X)$ e $w_1(X)$ e as operações $w_1(X)$ e $w_2(X)$. No entanto, as operações $r_1(X)$ e $r_2(X)$ não estão em conflito, pois ambas são operações de leitura; as operações $w_2(X)$ e $w_1(Y)$ não estão em conflito porque operam em itens de dados distintos X e Y; e as operações $r_1(X)$ e $w_1(X)$ não estão em conflito porque pertencem à mesma transação.

Intuitivamente, duas operações estão em conflito se a mudança de sua ordem puder resultar em algo diferente. Por exemplo, se mudarmos a ordem das duas operações $r_1(X)$; $w_2(X)$ para $w_2(X)$; $r_1(X)$, então o valor de X que é lido pela transação T_1 muda, pois na segunda ordem o valor de X é mudado por $w_2(X)$ antes que seja lido por $r_1(X)$, enquanto na primeira ordem o valor é lido antes de ser alterado. Isso é chamado de **conflito de leitura-gravação**. O outro tipo é chamado de **conflito de gravação-gravação**, e é ilustrado pelo caso em que mudamos a ordem das duas operações como $w_1(X)$; $w_2(X)$ para $w_2(X)$; $w_1(X)$. Para um conflito de gravação-gravação, o *último valor* de X será diferente porque em um caso ele é gravado por T_2 e no outro caso, por T_1. Observe que duas operações de leitura não estão em conflito, porque mudar sua ordem não faz diferença no resultado.

O restante desta seção aborda algumas definições teóricas com relação a schedules. Um schedule S de n transações T_1, T_2, ..., T_n é considerado um **schedule completo** se as seguintes condições forem mantidas:

1. As operações em S são exatamente aquelas operações em T_1, T_2, ..., T_n, incluindo uma operação de confirmação ou cancela-

mento como última operação em cada transação no schedule.

2. Para qualquer par de operações da mesma transação T_i, sua ordem de aparecimento relativa em S é a mesma que sua ordem de aparecimento em T_i.
3. Para duas operações quaisquer em conflito, uma das duas precisa ocorrer antes da outra no schedule.[10]

A condição anterior (3) permite que duas *operações não em conflito* ocorram no mesmo schedule sem definir qual ocorre primeiro, levando assim à definição de um schedule como uma **ordem parcial** das operações nas n transações.[11] Porém, uma ordem total precisa ser especificada no schedule para qualquer par de operações em conflito (condição 3) e para qualquer par de operações da mesma transação (condição 2). A condição 1 simplesmente indica que todas as operações nas transações precisam aparecer no schedule completo. Como cada transação é confirmada ou cancelada, um schedule completo *não terá quaisquer transações ativas* ao final do schedule.

Em geral, é difícil encontrar schedules completos em um sistema de processamento de transação, pois novas transações estão sendo continuamente submetidas ao sistema. Logo, é útil definir o conceito da **projeção confirmada** $C(S)$ de um schedule S, que inclui apenas as operações em S que pertencem a transações confirmadas — ou seja, transações T_i cuja operação de confirmação c_i está em S.

21.4.2 Caracterizando schedules com base na facilidade de recuperação

Para alguns schedules, é fácil recuperar-se de falhas de transação e sistema, enquanto para outros o processo de recuperação pode ser bem complicado. Em alguns casos, nem sequer é possível recuperar-se corretamente após uma falha. Portanto, é importante caracterizar os tipos de schedules para os quais *a recuperação é possível*, bem como aqueles para os quais a *recuperação é relativamente simples*. Essas caracterizações não oferecem de fato o algoritmo de recuperação; elas só tentam caracterizar de modo teórico os diferentes tipos de schedules.

Primeiro, gostaríamos de garantir que, quando uma transação T é confirmada, *nunca* deve ser necessário cancelar T. Isso garante que a propriedade de durabilidade das transações não é violada (ver Seção 21.3). Os schedules que teoricamente atendem a esse critério são chamados *schedules recuperáveis*; aqueles que não o fazem são chamados **não recuperáveis** e, portanto, não devem ser permitidos pelo SGBD. A definição de **schedules recuperáveis** é a seguinte: um schedule S é recuperável se nenhuma transação T em S for confirmada até que todas as transações T', que tiverem gravado algum item X que T lê, sejam confirmadas. Uma transação T **lê** da transação T' em um schedule S se algum item X for gravado primeiro por T' e depois lido por T. Além disso, T' não deve ser cancelado antes que T leia o item X, e não deve haver transações que gravam X depois que T' o grave e antes que T o leia (a menos que essas transações, se houver, forem abortadas antes que T leia X).

Alguns schedules recuperáveis podem exigir um processo de recuperação complexo, conforme veremos, mas se forem mantidas informações suficientes (no log), um algoritmo de recuperação poderá ser criado para qualquer schedule recuperável. Os schedules (parciais) S_a e S_b da seção anterior são ambos recuperáveis, pois satisfazem a definição acima. Considere o schedule S_a' dado a seguir, que é o mesmo que o schedule S_a, exceto que duas operações de confirmação foram acrescentadas a S_a:

$$S_a'\text{: } r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ c_2;\ w_1(Y);\ c_1;$$

S_a' é recuperável, embora sofra do problema da atualização; esse problema é tratado pela teoria da serialização (ver Seção 21.5). Porém, considere os dois schedules (parciais) S_c e S_d a seguir:

$$S_c\text{: } r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y);\ w_2(X);\ c_2;\ a_1;$$
$$S_d\text{: } r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y);\ w_2(X);\ w_1(Y);\ c_1;\ c_2;$$
$$S_e\text{: } r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y);\ w_2(X);\ w_1(Y);\ a_1;\ a_2;$$

S_c não é recuperável porque T_2 lê o item X de T_1, mas T_2 confirma antes que T_1 confirme. O problema ocorre se T_1 abortar depois da operação c_2 em S_c, então o valor de X que T_2 lê não é mais válido e T_2 precisa ser abortado *depois* de ser confirmado, levando a um schedule que *não é recuperável*. Para o schedule ser recuperável, a operação c_2 em S_c precisa ser adiada até depois de T_1 confirmar, como mostramos em S_d. Se T_1 abortar em vez de confirmar, então T_2 também deve abortar, conforme mostrado em S_e, pois o valor de X lido não é mais válido. Em S_e, abortar T_2 é aceitável porque ainda

[10] Teoricamente, não é necessário determinar uma ordem entre pares de operações *não em conflito*.

[11] Na prática, a maioria dos schedules possui uma ordem total de operações. Se o processamento paralelo for empregado, na teoria é possível ter schedules com operações não em conflito parcialmente ordenadas.

não foi confirmado, o que não é o caso para o schedule não recuperável S_c.

Em um schedule recuperável, nenhuma transação confirmada precisa ser cancelada e, portanto, a definição da transação confirmada como durável não é violada. Porém, é possível que um fenômeno conhecido como **rollback em cascata** (ou **propagação de cancelamento**) ocorra em alguns schedules recuperáveis, no qual uma transação *não confirmada* foi cancelada porque leu um item de uma transação que falhou. Isso é ilustrado no schedule S_e, onde a transação T_2 foi cancelada porque leu o item X de T_1, e T_1 então foi cancelada.

Como o rollback em cascata pode ser muito demorado — pois diversas transações podem ser canceladas (ver Capítulo 23) —, é importante caracterizar os schedules nos quais esse fenômeno certamente não ocorrerá. Um schedule é considerado **sem cascata**, ou que **evita o rollback em cascata**, se cada transação nele ler apenas itens que foram gravados por transações confirmadas. Nesse caso, todos os itens lidos não serão descartados, de modo que nenhum rollback em cascata ocorrerá. Para satisfazer esse critério, o comando $r_2(X)$ nos schedules S_d e S_e precisam ser adiados até depois que T_1 tiver sido confirmada (ou cancelada), adiando assim T_2, mas garantindo que não haja rollback em cascata se T_1 for cancelada.

Finalmente, existe um terceiro tipo de schedule, mais restritivo, chamado **schedule estrito**, em que as transações *não podem ler nem gravar* um item X até que a última transação que gravou X tenha sido confirmada (ou cancelada). Schedules estritos simplificam o processo de recuperação. Em um schedule estrito, o processo de desfazer uma operação write_item(X) de uma transação abortada serve apenas para restaurar a **imagem anterior** (*valor_antigo* ou BFIM) do item de dados X. Esse procedimento simples sempre funciona corretamente para schedules estritos, mas pode não funcionar para schedules recuperáveis ou sem cascata. Por exemplo, considere o schedule S_f:

$$S_f: w_1(X, 5); w_2(X, 8); a_1;$$

Suponha que o valor de X fosse originalmente 9, que é a imagem anterior armazenada no log do sistema junto com a operação $w_1(X, 5)$. Se T_1 for cancelada, como em S_f, o procedimento de recuperação que restaura a imagem anterior de uma operação de gravação cancelada restaurará o valor de X para 9, embora já tenha sido alterado para 8 pela transação T_2, levando, então, a resultados potencialmente incorretos. Embora o schedule S_f seja sem cascata, ele não é um schedule estrito, pois permite que T_2 grave o item X embora a transação T_1, que gravou X por último, ainda não tenha sido confirmada (ou cancelada). Um schedule estrito não tem esse problema.

É importante observar que qualquer schedule estrito também é sem cascata, e qualquer schedule sem cascata também é recuperável. Suponha que tenhamos i transações T_1, T_2, ..., T_i, e seu número de operações seja n_1, n_2, ..., n_i, respectivamente. Se criarmos um conjunto de todos os schedules possíveis dessas transações, podemos dividir os schedules em dois subconjuntos disjuntos: recuperáveis e não recuperáveis. Os schedules sem cascata serão um subconjunto dos schedules recuperáveis, e os schedules estritos serão um subconjunto dos schedules sem cascata. Assim, todos os schedules estritos são sem cascata, e todos os schedules sem cascata são recuperáveis.

21.5 Caracterizando schedules com base na facilidade de serialização

Na seção anterior, caracterizamos os schedules com base em suas propriedades de facilidade de recuperação. Agora, caracterizamos os tipos de schedules que são sempre considerados *corretos* quando transações concorrentes estão sendo executadas. Esses schedules são conhecidos como *schedules serializáveis*. Suponha que dois usuários — por exemplo, dois agentes de reservas aéreas — submetam às transações do SGBD T_1 e T_2 da Figura 21.2 aproximadamente ao mesmo tempo. Se nenhuma intercalação de operações for permitida, existem apenas dois resultados possíveis:

1. Executar todas as operações da transação T_1 (em sequência) seguidas por todas as operações da transação T_2 (em sequência).
2. Executar todas as operações da transação T_2 (em sequência) seguidas por todas as operações da transação T_1 (em sequência).

Esses dois schedules — chamados *schedules seriais* — são mostrados na Figura 21.5(a) e (b), respectivamente. Se a intercalação de operações for permitida, haverá muitas ordens possíveis em que o sistema pode executar as operações individuais das transações. Dois schedules possíveis aparecem na Figura 21.5(c). O conceito de **serialização de schedules** é usado para identificar quais schedules estão corretos quando as execuções da transação tiverem intercalação de suas operações nos schedules. Esta seção define a serialização e discute como ela pode ser usada na prática.

21.5.1 Schedules seriais, não seriais e serializáveis por conflito

Os schedules A e B da Figura 21.5(a) e (b) são

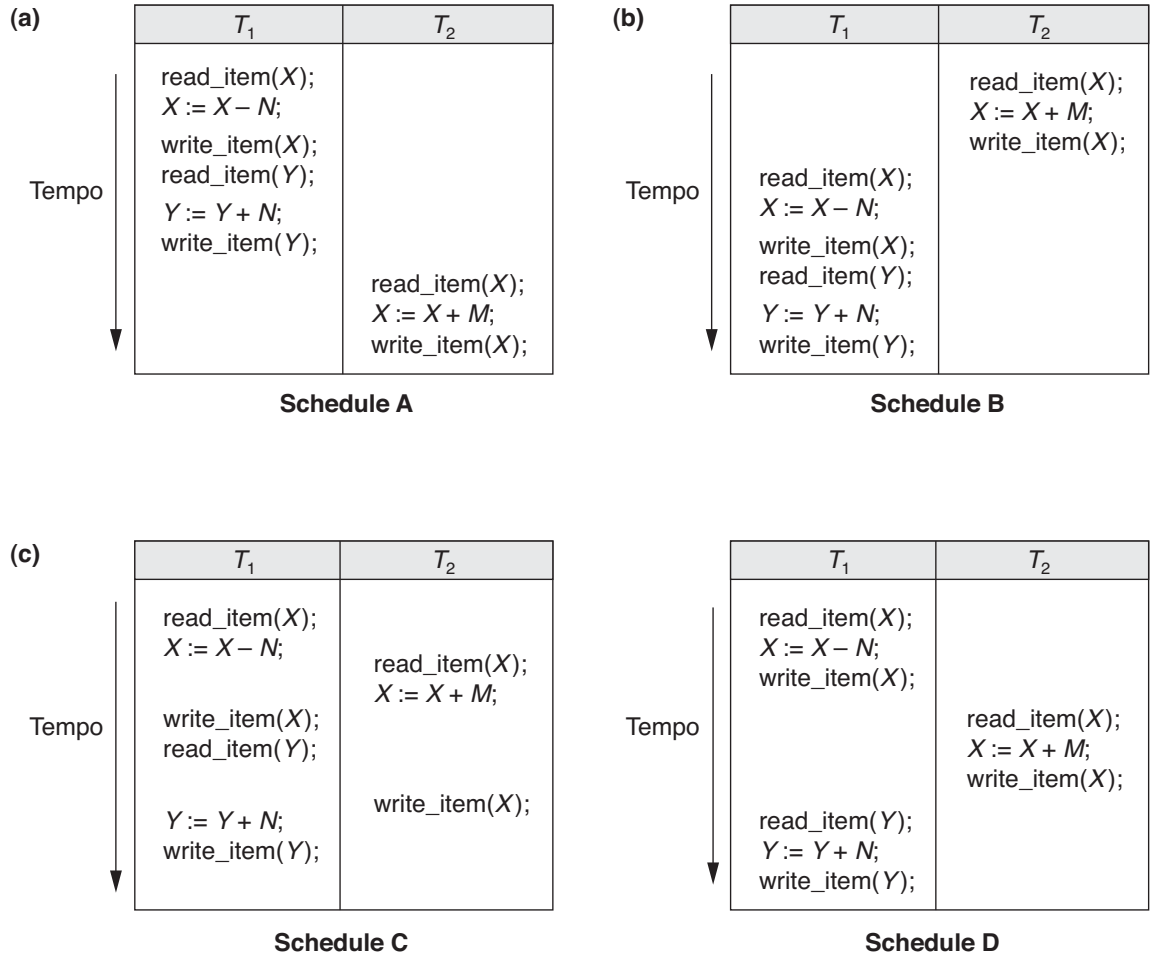


Figura 21.5

Exemplos de schedules seriais e não seriais envolvendo as transações T_1 e T_2. (a) Schedule serial A: T_1 seguida por T_2. (b) Schedule serial B: T_2 seguida por T_1. (c) Dois schedules não seriais C e D com intercalação de operações.

chamados de *seriais* porque as operações de cada transação são executadas consecutivamente, sem quaisquer operações intercaladas da outra transação. Em um schedule serial, transações inteiras são realizadas em ordem serial: T_1 e, depois, T_2 na Figura 21.5(a), e T_2 e, depois, T_1 na Figura 21.5(b). Os schedules C e D da Figura 21.5(c) são chamados de *não seriais*, pois cada sequência intercala operações das duas transações.

De mesma forma, um schedule *S* é **serial** se, para cada transação *T* participante do schedule, todas as operações de *T* forem executadas consecutivamente no schedule; caso contrário, o schedule é chamado de **não serial**. Portanto, em um schedule serial, somente uma transação de cada vez está ativa — o commit (ou abort) da transação ativa inicia a execução da próxima transação. Não ocorre nenhuma intercalação em um schedule serial. Uma suposição razoável que podemos fazer, se considerarmos que as transações são *independentes*, é que *cada schedule serial é considerado correto*. Podemos assumir isso porque cada transação é considerada correta se executada por conta própria (de acordo com a propriedade de *preservação de consistência* da Seção 21.3). Logo, não importa qual transação é executada em primeiro lugar. Desde que cada transação seja executada do início ao fim isoladamente das operações das outras transações, obtemos um resultado correto no banco de dados.

O problema com os schedules seriais é que eles limitam a concorrência ao proibir a intercalação de operações. Em um schedule serial, se uma transação espera que uma operação de E/S termine, não podemos passar o processador da CPU para outra transação, desperdiçando assim um valioso tempo de processamento da CPU. Além disso, se alguma transação *T* for muito longa, as outras transações deverão esperar até que *T* termine todas as suas operações antes de poderem começar. Portanto, os schedules seriais são *considerados inaceitáveis* na prática. Porém, se pudermos determinar quais outros schedules são

equivalentes a um schedule serial, podemos permitir que estes ocorram.

Para ilustrar nossa discussão, considere os schedules da Figura 21.5 e considere que os valores iniciais dos itens de banco de dados sejam $X = 90$ e $Y = 90$, e que $N = 3$ e $M = 2$. Depois de executar as transações T_1 e T_2, esperamos que os valores do banco de dados sejam $X = 89$ e $Y = 93$, de acordo com o significado das transações. Com certeza, a execução dos schedules seriais A ou B produz os resultados corretos. Agora, considere os schedules não seriais C e D. O schedule C (que é o mesmo da Figura 21.3(a)) produz os resultados $X = 92$ e $Y = 93$, em que o valor X está errado, enquanto o schedule D produz os resultados corretos.

O schedule C produz um resultado errado devido ao *problema da atualização perdida*, discutido na Seção 21.1.3. A transação T_2 lê o valor de X antes de ele ser alterado pela transação T_1, de modo que somente o efeito de T_2 em X é refletido no banco de dados. O efeito de T_1 em X é *perdido*, sobrescrito por T_2, levando ao resultado incorreto para o item X. No entanto, alguns schedules não seriais produzem o resultado correto esperado, como o schedule D. Gostaríamos de determinar quais dos schedules não seriais *sempre* produzem o resultado correto e quais podem gerar resultados errôneos. O conceito utilizado para caracterizar schedules dessa maneira é o da serialização de um schedule.

A definição de *schedule serializável* é a seguinte: um schedule S de n transações é **serializável** se for *equivalente a algum schedule serial* das mesmas n transações. Definiremos o conceito de *equivalência de schedules* em breve. Observe que existem $n!$ schedules seriais possíveis de n transações e muito mais schedules não seriais possíveis. Podemos formar dois grupos distintos dos schedules não seriais — aqueles que são equivalentes a um (ou mais) dos schedules seriais e, portanto, que são serializáveis, e aqueles que não são equivalentes a *qualquer* schedule serial e, portanto, não são serializáveis.

Dizer que um schedule não serial S é serializável é equivalente a dizer que ele é correto, pois é equivalente a um schedule serial, que é considerado correto. A pergunta que resta é: quando dois schedules são considerados *equivalentes*?

Existem várias maneiras de definir a equivalência de schedule. A definição mais simples, porém menos satisfatória, envolve comparar os efeitos dos schedules no banco de dados. Dois schedules são chamados **equivalentes no resultado** se produzirem o mesmo estado final do banco de dados. Contudo, dois schedules diferentes podem acidentalmente produzir o mesmo estado final. Por exemplo, na Figura 21.6, os schedules S_1 e S_2 produzirão o mesmo estado de banco de dados final se forem executados em um banco de dados com um valor inicial de $X = 100$; porém, para outros valores iniciais de X, os schedules *não* são equivalentes no resultado. Além disso, esses schedules executam transações diferentes, de modo que definitivamente não deverão ser considerados equivalentes. Assim, a equivalência isolada no resultado não pode ser usada para definir a equivalência de schedules. A técnica mais segura e mais geral para definir a equivalência de schedule é não fazer quaisquer suposições sobre os tipos de operações incluídas nas transações. Para dois schedules serem equivalentes, as operações aplicadas a cada item de dados afetado pelos schedules devem ser aplicadas a esse item nos dois schedules *na mesma ordem*. Duas definições de equivalência de schedules costumam ser usadas: *equivalência de conflito* e *equivalência de visão*. Vamos discutir a equivalência de conflito em seguida, que é a definição mais utilizada.

A definição de *equivalência de conflito* dos schedules é a seguinte: dois schedules são considerados **equivalentes em conflito** se a ordem de duas *operações em conflito* quaisquer for a mesma nos dois schedules. Lembre-se, da Seção 21.4.1, que duas operações em um schedule são consideradas em *conflito* se pertencerem a transações diferentes, acessarem o mesmo item do banco de dados, e se uma ou ambas forem operações write_item ou uma for um write_item e a outra, um read_item. Se duas operações em conflito forem aplicadas em *ordens diferentes* em dois schedules, o efeito pode ser diferente no banco de dados ou nas transações no schedule, e, portanto, os schedules não são equivalentes em conflito. Por exemplo, conforme discutimos na Seção 21.4.1, se uma operação de leitura e gravação ocorrer na ordem $r_1(X)$, $w_2(X)$ no schedule S_1, e na ordem contrária $w_2(X)$, $r_1(X)$ no schedule S_2, o valor lido por $r_1(X)$ pode ser diferente nos dois schedules. De modo semelhante, se duas operações de gravação ocorrerem na ordem $w_1(X)$, $w_2(X)$ em S_1, e na ordem contrária $w_2(X)$, $w_1(X)$ em S_2, a próxima operação $r(X)$ nos dois schedules lerá valores potencialmente diferentes; ou, então, se estas forem as últimas operações gra-

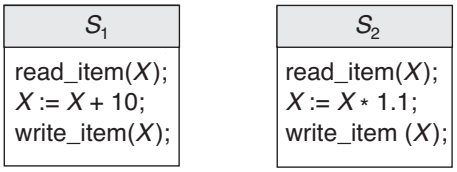


Figura 21.6
Dois schedules que são equivalentes no resultado para o valor inicial de $X = 100$, mas não são equivalentes no resultado em geral.

vando o item X nos schedules, o valor final do item X no banco de dados será diferente.

Usando a noção de equivalência de conflito, definimos um schedule S como sendo **serializável de conflito**[12] se ele for equivalente (em conflito) a algum schedule serial S'. Nesse caso, podemos reordenar as operações *não em conflito* em S até formarmos o schedule serial equivalente S'. De acordo com essa definição, o schedule D da Figura 21.5(c) é equivalente ao schedule serial A da Figura 21.5(a). Em ambos os schedules, o read_item(X) de T_2 lê o valor de X gravado por T_1, enquanto as outras operações read_item leem os valores do banco de dados com base no estado inicial do banco de dados. Além disso, T_1 é a última transação a gravar Y, e T_2 é a última transação a gravar X nos dois schedules. Como A é um schedule serial e o schedule D é equivalente a A, D é um schedule serializável. Observe que as operações $r_1(Y)$ e $w_1(Y)$ do schedule D não estão em conflito com as operações $r_2(X)$ e $w_2(X)$, pois acessam itens de dados diferentes. Portanto, podemos mover $r_1(Y)$, $w_1(Y)$ antes de $r_2(X)$, $w_2(X)$, levando ao schedule serial equivalente T_1, T_2.

O schedule C da Figura 21.5(c) não é equivalente a qualquer um dos dois possíveis schedules seriais A e B, e, portanto, *não é serializável*. Tentar reordenar as operações do schedule C para encontrar um schedule serial equivalente gera uma falha, pois $r_2(X)$ e $w_1(X)$ estão em conflito, o que significa que não podemos mover $r_2(X)$ para baixo para obter o schedule serial equivalente T_1, T_2. De modo semelhante, como $w_1(X)$ e $w_2(X)$ estão em conflito, não podemos mover $w_1(X)$ para baixo para obter o schedule serial equivalente T_2, T_1.

Outra definição de equivalência, mais complexa — chamada *equivalência de visão*, que leva ao conceito de serialização de visão —, é discutida na Seção 21.5.4.

21.5.2 Testando a serialização por conflito de um schedule

Existe um algoritmo simples para determinar se determinado schedule é serializável de conflito ou não. A maioria dos métodos de controle de concorrência *não* testa realmente a serialização. Em vez disso, protocolos ou regras são desenvolvidas para garantir que qualquer schedule que siga essas regras será serializável. Discutimos aqui o algoritmo para testar a serialização de conflito dos schedules, para entendermos melhor esses protocolos de controle de concorrência, que serão discutidos no Capítulo 22.

O Algoritmo 21.1 pode ser usado para testar um schedule para serialização de conflito. O algoritmo examina apenas as operações read_item e write_item em um schedule para construir um **grafo de precedência** (ou **grafo de serialização**), o qual é um **grafo direcionado** $G = (N, E)$ que consiste em um conjunto de nós $N = \{T_1, T_2, ..., T_n\}$ e um conjunto de arestas direcionadas $A = \{a_1, a_2, ..., a_n\}$. Existe um nó no grafo para cada transação T_i no schedule. Cada aresta e_i no grafo tem a forma $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, onde T_j é o **nó inicial** de a_i e T_k é o **nó final** de a_i. Tal aresta do nó T_j ao nó T_k é criada pelo algoritmo se uma das operações em T_j aparecer no schedule antes de alguma *operação de conflito* em T_k.

Algoritmo 21.1. Testando a serialização de conflito de um schedule S

1. Para cada transação T_i participante no schedule S, crie um nó rotulado com T_i no grafo de precedência.
2. Para cada caso em S onde T_j executa um read_item(X) depois de T_i executar um write_item(X), crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
3. Para cada caso em S onde T_j executa um write_item(X) após T_i executar um read_item(X), crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
4. Para cada caso em S onde T_j executa um write_item(X) após T_i executar um write_item(X), crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
5. O schedule S é serializável se, e somente se, o grafo de precedência não tiver ciclos.

O grafo de precedência é construído conforme descrito no Algoritmo 21.1. Se houver um ciclo no grafo de precedência, o schedule S não é serializável (conflito); se não houve ciclo, S é serializável. Um **ciclo** em um grafo direcionado é uma **sequência de arestas** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), ..., (T_i \rightarrow T_j))$ com a propriedade de que o nó inicial de cada aresta — exceto a primeira aresta — é o mesmo que o nó final da aresta anterior, e o nó inicial da primeira aresta é o mesmo que o nó final da última aresta (a sequência começa e termina no mesmo nó).

No grafo de precedência, uma aresta de T_i para T_j significa que a transação T_i precisa vir antes da transação T_j em qualquer schedule serial que seja equivalente a S, pois duas operações em conflito aparecem no schedule nessa ordem. Se não houver ciclo no grafo de precedência, podemos criar um **schedule serial equivalente** S' que é equivalente a S, ordenando

[12] Usaremos o termo ***serializável*** para indicar serializável de conflito. Outra definição de serializável usada na prática (ver Seção 21.6) é ter leituras repetitivas, não leituras sujas, e nenhum registro fantasma (ver na Seção 22.7.1 uma discussão sobre fantasmas).

as transações que participam em *S* da seguinte forma: sempre que existir uma aresta no grafo de precedência de T_i para T_j, T_i deve aparecer antes de T_j no schedule serial equivalente *S*'.[13] Observe que as arestas ($T_i \rightarrow T_j$) em um grafo de precedência opcionalmente podem ser rotuladas pelo(s) nome(s) do item (ou itens) de dados que leva(m) à criação da aresta. A Figura 21.7 mostra esses rótulos nas arestas.

Em geral, vários schedules seriais podem ser equivalentes a *S* se o grafo de precedência para *S* não tiver ciclo. Contudo, se o grafo de precedência tiver um ciclo, é fácil mostrar que não podemos criar qualquer schedule serial equivalente, de modo que *S* não é serializável. Os grafos de precedência criados para os schedules A a D, respectivamente, na Figura 21.5, aparecem na Figura 21.7(a) a (d). O grafo para o schedule C tem um ciclo, de modo que não é serializável. O grafo para o schedule D não tem ciclo, de modo que é serializável, e o schedule serial equivalente é T_1 seguido por T_2. Os grafos para os schedules A e B não têm ciclos, como é de se esperar, pois os schedules são seriais e, portanto, serializáveis.

Outro exemplo, em que três transações participam, aparece na Figura 21.8. A Figura 21.8(a) mostra as operações read_item e write_item em cada transação. Dois schedules *E* e *F* para essas transações são exibidos na Figura 21.8(b) e (c), respectivamente, e os grafos de precedência para os schedules E e F aparecem nas partes (d) e (e). O schedule E não é serializável porque o grafo de precedência correspondente tem ciclos. O schedule F é serializável, e o schedule serial equivalente a *F* aparece na Figura 21.8(e). Embora só exista um schedule serial equivalente para *F*, em geral, pode haver mais de um schedule serial equivalente para um schedule serializável. A Figura 21.8(f) mostra um grafo de precedência representando um schedule que tem dois schedules seriais equivalentes. Para achar um schedule serial equivalente, comece com um nó que não tem quaisquer arestas chegando, e depois cuide para que a ordem dos nós para cada aresta não seja violada.

21.5.3 Como a serialização é usada para controle de concorrência

Conforme discutimos anteriormente, dizer que um schedule *S* é serializável (de conflito) — ou seja, *S* é equivalente (em conflito) a um schedule serial — é equivalente a dizer que *S* está correto. Contudo, ser *serializável* é diferente de ser *serial*. Um schedule serial representa um processamento ineficiente, pois nenhuma intercalação de operações de diferentes transações é permitida. Isso pode levar a uma baixa utilização de CPU enquanto uma transação espera pela E/S de disco, ou que outra transação termine, dessa forma, atrasando consideravelmente o processamento. Um schedule serializável oferece os benefícios da execução concorrente sem abrir mão de qualquer exatidão. Na prática, é muito difícil testar a serialização de um schedule. A intercalação de operações de transações concorrentes — que normalmente são executadas como processos pelo sistema operacional — costuma ser determinada pelo scheduler do sistema operacional, que aloca recursos para todos os processos. Fatores como carga do sistema, tempo de submissão de transação e prioridades de processos contribuem para a ordenação de operações em um schedule. Logo, é difícil determinar como as opera-

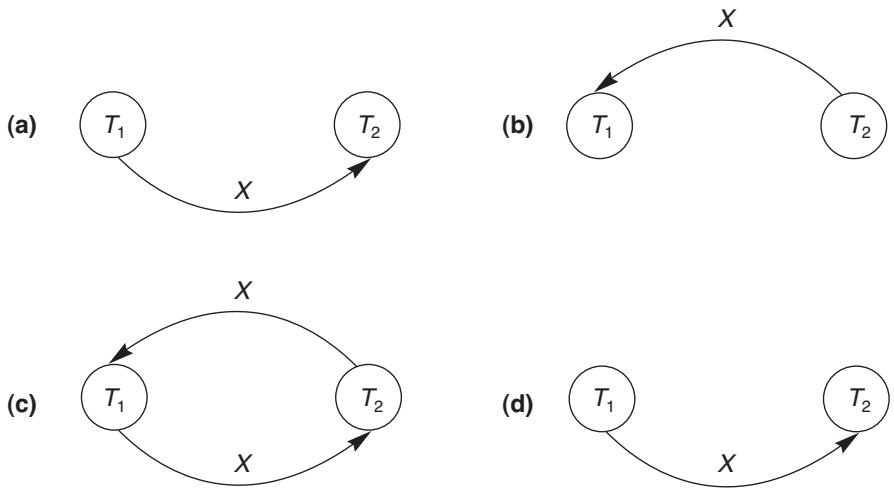
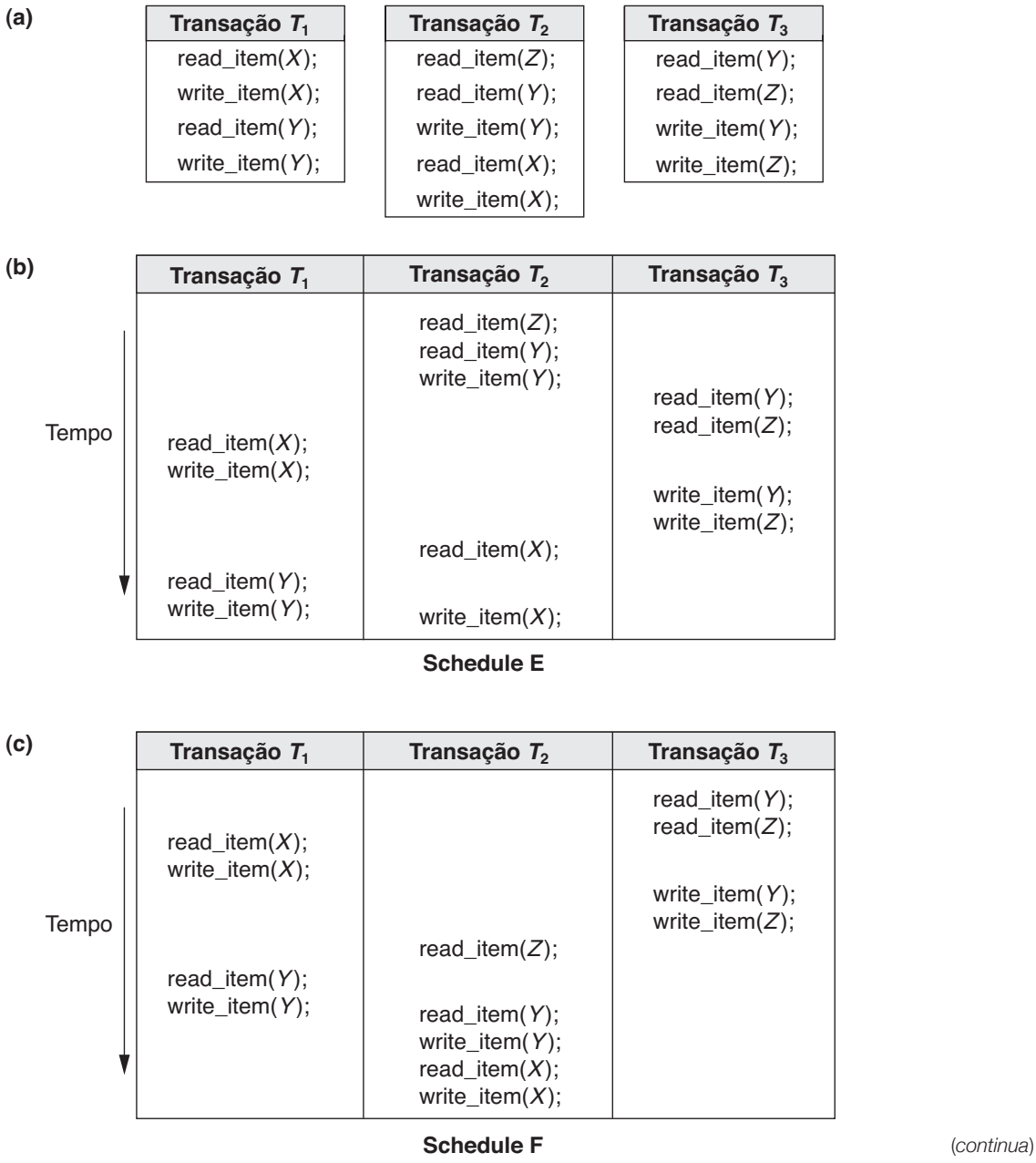


Figura 21.7
Construindo os grafos de precedência para os schedules A a D da Figura 21.5 para testar a serialização de conflito. (a) Grafo de precedência para o schedule serial A. (b) Grafo de precedência para o schedule serial B. (c) Grafo de precedência para o schedule C (não serializável). (d) Grafo de precedência para o schedule D (serializável, equivalente ao schedule A).

[13] Esse processo de ordenação dos nós de um grafo acíclico é conhecido como *ordenação topológica*.



(*continua*)

Figura 21.8
Outro exemplo de teste de serialização. (a) As operações de leitura e gravação de três transações T_1, T_2 e T_3. (b) Schedule E. (c) Schedule F.

ções de um schedule serão intercaladas de antemão para garantir a serialização.

Se as transações forem executadas à vontade e depois o schedule resultante tiver a serialização testada, temos de cancelar o efeito do schedule se ele não for serializável. Esse é um problema sério, que torna essa técnica impraticável. Logo, a técnica usada na maioria dos sistemas práticos é determinar métodos ou protocolos que garantam a serialização, sem ter de testar os próprios schedules. A técnica usada na maioria dos SGBDs comerciais é projetar **protocolos** (conjuntos de regras) que — se seguidos por *toda* transação individual ou se impostos por um subsistema de controle de concorrência do SGBD — garantirão a serialização de *todos os schedules em que as transações participam*.

Outro problema aparece aqui: quando as transações são submetidas continuamente ao sistema, é difícil determinar quando um schedule começa e quando ele termina. A teoria da serialização pode ser adaptada para lidar com esse problema, considerando apenas a projeção confirmada de um schedule *S*.

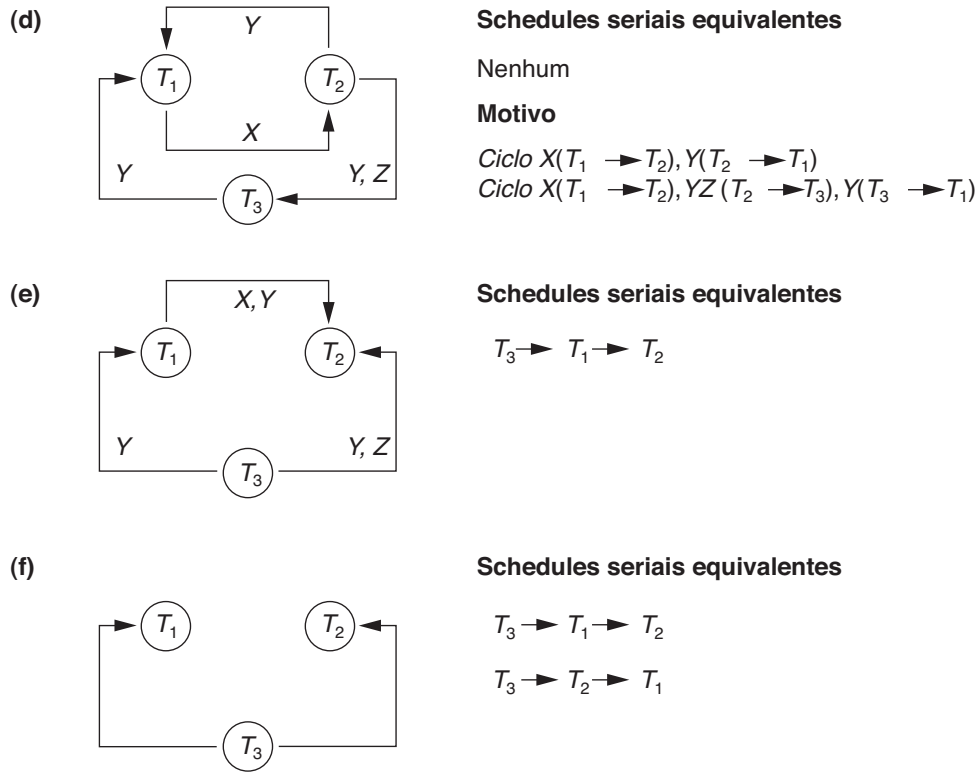


Figura 21.8 (continuação)
Outro exemplo de teste de serialização. Grafo de precedência para o schedule E. Grafo de precedência para o schedule F. Grafo de precedência com dois schedules seriais equivalentes.

Lembre-se, da Seção 21.4.1, que a *projeção confirmada* $C(S)$ de um schedule S inclui apenas as operações em S que pertencem às transações confirmadas. Teoricamente, podemos definir um schedule S para ser serializável se sua projeção confirmada $C(S)$ for equivalente a algum schedule serial, pois apenas transações confirmadas são garantidas pelo SGBD.

No Capítulo 22, discutimos uma série de protocolos de controle de concorrência diferentes, que garantem a serialização. A técnica mais comum, chamada *bloqueio em duas fases*, é baseada no bloqueio de itens de dados para impedir que transações concorrentes interfiram umas com as outras, e na imposição de uma condição adicional que garanta a serialização. Isso é usado na maioria dos SGBDs comerciais. Outros protocolos foram propostos;[14] entre eles estão a *ordenação por rótulo de tempo* (timestamp), em que cada transação recebe um rótulo de tempo único e o protocolo garante que quaisquer operações em conflito sejam executadas na ordem dos rótulos de tempo da transação; *protocolos multiversão*, que são baseados na manutenção de várias versões dos itens de dados; e *protocolos otimistas* (também chamados de *certificação* ou *validação*), que verificam as possíveis violações de serialização após as transações terminarem, mas antes que elas possam ser confirmadas.

21.5.4 Equivalência de visão e serialização de visão

Na Seção 21.5.1, definimos os conceitos de equivalência de conflito dos schedules e serialização de conflito. Outra definição menos restritiva da equivalência de schedules é chamada *equivalência de visão*. Isso leva a outra definição de serialização, chamada *serialização de visão*. Dois schedules S e S' são considerados **equivalentes de visão** se as três condições a seguir forem mantidas:

1. O mesmo conjunto de transações participa em S e S', e S e S' incluem as mesmas operações dessas transações.
2. Para qualquer operação $r_i(X)$ de T_i em S, se o valor de X lido pela operação tiver sido gravado por uma operação $w_j(X)$ de T_j (ou se for o valor original de X antes do schedule ter sido iniciado), a mesma condição deve ser mantida para o valor de X lido pela operação $r_i(X)$ de T_i em S'.
3. Se a operação $w_k(Y)$ de T_k for a última opera-

[14] Esses outros protocolos não foram incorporados em muitos sistemas comerciais; a maioria dos SGBDs relacionais utiliza alguma variação do protocolo de bloqueio em duas fases.

ção a gravar o item Y em S, então $w_k(Y)$ de T_k também deve ser a última operação a gravar o item Y em S'.

A ideia por trás da equivalência de visão é que, desde que cada operação de leitura de uma transação leia o resultado da mesma operação de gravação nos dois schedules, as operações de gravação de cada transação devem produzir os mesmos resultados. As operações de leitura, portanto, *veem a mesma visão* nos dois schedules. A condição 3 garante que a operação de gravação final em cada item de dados seja a mesma nos dois schedules, de modo que o estado do banco de dados deverá ser o mesmo ao final dos dois schedules. Um schedule S é considerado **serializável de visão** se for equivalente de visão a um schedule serial.

As definições de serialização de conflito e serialização de visão são semelhantes se uma condição conhecida como **suposição de gravação restrita** (ou **sem gravações cegas**) se mantiver em todas as transações no schedule. Essa condição afirma que qualquer operação de gravação $w_i(X)$ em T_i é precedida por um $r_i(X)$ em T_i e que o valor gravado por $w_i(X)$ em T_i depende apenas do valor de X lido por $r_i(X)$. Isso considera que o cálculo do novo valor de X é uma função $f(X)$ baseada no valor antigo de X lido do banco de dados. Uma **gravação cega** é uma operação de gravação em uma transação T em um item X que não depende do valor de X, de modo que não é precedida por uma leitura de X na transação T.

A definição de serialização de visão é menos restrita do que a da serialização de conflito sob a **suposição de gravação irrestrita**, em que o valor gravado por uma operação $w_i(X)$ em T_i pode ser independente de seu valor antigo do banco de dados. Isso é possível quando as *gravações cegas* são permitidas, e é ilustrado pelo schedule S_g a seguir, de três transações T_1: $r_1(X)$; $w_1(X)$; T_2: $w_2(X)$; e T_3: $w_3(X)$:

$$S_g\text{: } r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

Em S_g, as operações $w_2(X)$ e $w_3(X)$ são gravações cegas, pois T_2 e T_3 não leem o valor de X. O schedule S_g é serializável de visão, pois é equivalente de visão ao schedule serial T_1, T_2, T_3. Porém, S_g não é serializável de conflito, visto que não é equivalente de conflito para qualquer schedule serial. Já foi mostrado que qualquer schedule serializável de conflito também é serializável de visão, mas não o contrário, conforme ilustrado pelo exemplo anterior. Existe um algoritmo para testar se um schedule S é serializável de visão ou não. Contudo, o problema de testar a serialização de visão tem sido mostrado como muito difícil, significando que descobrir um algoritmo de tempo polinomial eficiente para esse problema é altamente improvável.

21.5.5 Outros tipos de equivalência de schedules

A serialização de schedules às vezes é considerada muito restritiva como uma condição para garantir a exatidão das execuções concorrentes. Algumas aplicações podem produzir schedules que são corretas ao satisfazer condições menos rigorosas do que a serialização de conflito ou a serialização de visão. Um exemplo é o tipo de transações conhecido como **transações de débito-crédito** — por exemplo, aquelas que aplicam depósitos e saques a um item de dados cujo valor é o saldo atual de uma conta bancária. A semântica das operações de débito-crédito é que elas atualizam o valor de um item de dados X ao subtrair ou somar ao valor do item de dados. Como as operações de adição e subtração são comutativas — ou seja, elas podem ser aplicadas em qualquer ordem —, é possível produzir schedules corretos que não sejam serializáveis. Por exemplo, considere as transações a seguir, cada qual podendo ser usada para transferir um valor monetário entre duas contas bancárias:

$$T_1\text{: } r_1(X); X := X - 10; w_1(X); r_1(Y); Y := Y + 10; w_1(Y);$$

$$T_2\text{: } r_2(Y); Y := Y - 20; w_2(Y); r_2(X); X := X + 20; w_2(X);$$

Considere o schedule não serializável S_h a seguir para as duas transações:

$$S_h\text{: } r_1(X); w_1(X); r_2(Y); w_2(Y); r_1(Y); w_1(Y); r_2(X); w_2(X);$$

Com o conhecimento adicional, ou **semântica**, de que as operações entre cada $r_i(I)$ e $w_i(I)$ são comutativas, sabemos que a ordem de execução das sequências que consistem em (ler, atualizar, gravar) não é importante, desde que cada sequência (ler, atualizar, gravar) por uma transação T_i em um item I em particular não seja interrompida por operações em conflito. Logo, o schedule S_h é considerado correto embora não seja serializável. Os pesquisadores têm trabalhado na extensão da teoria do controle de concorrência para lidar com casos em que a serialização é considerada muito restritiva como uma condição para a exatidão dos schedules. Além disso, em certos domínios de aplicações, como o projeto auxiliado por computador (CAD) de sistemas complexos de aeronaves, as transações de projeto duram um longo período. Em tais aplicações, esquemas de controle de

concorrência mais relaxados têm sido propostos para manter a consistência do banco de dados.

21.6 Suporte para transação em SQL

Nesta seção, oferecemos uma rápida introdução ao suporte para transação em SQL. Existem muito mais detalhes, e os padrões mais novos têm mais comandos para processamento de transação. A definição básica de uma transação SQL é semelhante ao conceito já definido de uma transação. Ou seja, ela é uma unidade lógica de trabalho e tem garantias de ser atômica (ou indivisível). Uma única instrução SQL sempre é considerada atômica — ou ela completa a execução sem um erro, ou falha e deixa o banco de dados inalterado.

Com a SQL, não existe uma instrução Begin_Transaction explícita. O início da transação é feito implicitamente quando instruções SQL em particular são encontradas. Porém, cada transação precisa ter uma instrução de fim explícita, que é um COMMIT ou um ROLLBACK. Cada transação tem certas características atribuídas a ela. Essas características são especificadas por uma instrução SET TRANSACTION em SQL. As características são o *modo de acesso*, o *tamanho da área de diagnóstico* e o *nível de isolamento*.

O **modo de acesso** pode ser especificado como READ ONLY ou READ WRITE. O default é READ WRITE, a menos que o nível de isolamento de READ UNCOMMITTED seja especificado (ver a seguir), caso em que READ ONLY é assumido. Um modo READ WRITE permite a execução de comandos de seleção, atualização, inserção, exclusão e criação. Um modo READ ONLY, como o nome indica, serve simplesmente para a recuperação de dados.

A opção de **tamanho da área de diagnóstico**, DIAGNOSTIC SIZE *n*, especifica um valor inteiro *n*, que indica o número de condições que podem ser mantidas de maneira simultânea na área de diagnóstico. Essas condições fornecem informações de feedback (erros ou exceções) ao usuário ou programa nas *n* instruções SQL executadas mais recentemente.

A opção de **nível de isolamento** é especificada usando a instrução ISOLATION LEVEL <isolamento>, em que o valor para <isolamento> pode ser READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ ou SERIALIZABLE.[15] O nível de isolamento default é SERIALIZABLE, embora alguns sistemas usem READ COMMITTED como default. O uso do termo SERIALIZABLE aqui é baseado em não permitir violações que causam leitura suja, leitura não repetitiva e fantasmas,[16] e, assim, não é idêntico ao modo como a serialização foi definida anteriormente na Seção 21.5. Se uma transação é executada em um nível de isolamento inferior a SERIALIZABLE, então uma ou mais das três violações a seguir pode ocorrer:

1. **Leitura suja.** Uma transação T_1 pode ler a atualização de uma transação T_2, que ainda não foi confirmada. Se T_2 falhar e for abortada, então T_1 teria lido um valor que não existe e é incorreto.
2. **Leitura não repetitiva.** Uma transação T_1 pode ler determinado valor de uma tabela. Se outra transação T_2 mais tarde atualizar esse valor e T_1 ler o valor novamente, T_1 verá um valor diferente.
3. **Fantasmas.** Uma transação T_1 pode ler um conjunto de linhas de uma tabela, talvez com base em alguma condição especificada na cláusula SQL WHERE. Agora, suponha que uma transação T_2 insira uma nova linha que também satisfaça a condição da cláusula WHERE usada em T_1, na tabela usada por T_1. Se T_1 for repetida, então T_1 verá um fantasma, uma linha que anteriormente não existia.

A Tabela 21.1 resume as possíveis violações para os diferentes níveis de isolamento. Uma entrada *Sim* indica que uma violação é possível e uma entrada *Não* indica que ela não é possível. READ UNCOMMITTED é a mais complacente, e SERIALIZABLE é a mais restritiva porque evita todos os três problemas mencionados anteriormente.

Uma transação SQL de exemplo pode se parecer com o seguinte:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO FUNCIONARIO (Pnome, Unome, Cpf, Dnr, Salario) VALUES ('Roberto', 'Silva', '99100432111', 2, 35.000);
EXEC SQL UPDATE FUNCIONARIO
    SET Salario = Salario * 1.1 WHERE Dnr = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

[15] Estes são semelhantes aos *níveis de isolamento* discutidos rapidamente ao final da Seção 21.3.

[16] Os problemas de leitura suja e leitura não repetitiva foram discutidos na Seção 21.1.3. Fantasmas serão discutidos na Seção 22.7.1.

Tabela 21.1
Violações possíveis com base nos níveis de isolamento definidos na SQL.

	Tipo de violação		
Nível de isolamento	Leitura suja	Leitura não repetitiva	Fantasma
READ UNCOMMITTED	Sim	Sim	Sim
READ COMMITTED	Não	Sim	Sim
REPEATABLE READ	Não	Não	Sim
SERIALIZABLE	Não	Não	Não

Essa transação consiste em primeiro inserir uma nova linha na tabela FUNCIONARIO e, depois, atualizar o salário de todos os funcionários que trabalham no departamento 2. Se houver um erro em qualquer uma das instruções SQL, a transação inteira é cancelada. Isso implica que qualquer salário atualizado (por essa transação) seria restaurado a seu valor anterior e que a linha recém-inserida seria removida.

Conforme vimos, a SQL oferece uma série de recursos orientados à transação. O DBA ou programadores de banco de dados podem tirar proveito dessas opções para tentar melhorar o desempenho da transação ao relaxar a serialização, se isso for aceitável para suas aplicações.

Resumo

Neste capítulo, discutimos os conceitos de SGBD para processamento de transação. Apresentamos o conceito de uma transação de banco de dados e as operações relevantes ao processamento de transação. Comparamos sistemas monousuário com sistemas de multiusuário e, depois, apresentamos exemplos de como a execução não controlada de transações simultâneas em um sistema multiusuários pode gerar resultados e valores de banco de dados incorretos. Também discutimos os diversos tipos de falhas que podem ocorrer durante a execução da transação.

Em seguida, apresentamos os estados típicos pelos quais uma transação passa durante a execução e discutimos vários conceitos que são usados nos métodos de recuperação e controle de concorrência. O log do sistema registra os acessos do banco de dados, e o sistema utiliza essa informação para se recuperar de falhas. Uma transação tem sucesso ou atinge seu ponto de confirmação, ou falha e precisa ser cancelada. Uma transação confirmada tem suas mudanças gravadas permanentemente no banco de dados. Apresentamos uma visão geral das propriedades desejáveis das transações — atomicidade, preservação de consistência, isolamento e durabilidade — que normalmente são conhecidas como propriedades ACID.

Depois, definimos um schedule (ou histórico) como uma sequência de execução das operações de várias transações com possível intercalação. Caracterizamos os schedules em relação a sua facilidade de recuperação. Os schedules recuperáveis garantem que, quando uma transação é confirmada, ela nunca precisará ser desfeita. Os schedules sem cascata acrescentam uma condição para garantir que nenhuma transação cancelada exija o cancelamento em cascata de outras transações. Schedules estritos oferecem uma condição ainda mais forte que permite um esquema de recuperação simples, consistindo em restaurar os valores antigos dos itens que foram alterados por uma transação abortada.

Definimos a equivalência dos schedules e vimos que um schedule serializável é equivalente a algum schedule serial. Definimos os conceitos de equivalência de conflito e equivalência de visão, que levam às definições de serialização de conflito e serialização de visão. Um schedule serializável é considerado correto. Apresentamos um algoritmo para testar a serialização (conflito) de um schedule. Discutimos por que o teste de serialização é impraticável em um sistema real, embora possa ser usado para definir e verificar os protocolos de controle de concorrência, e mencionamos rapidamente definições menos restritivas de equivalência de schedule. Por fim, fornecemos uma breve visão geral de como os conceitos de transação são usados na prática dentro da SQL.

Perguntas de revisão

21.1. O que significa a execução concorrente de transações de banco de dados em um sistema multiusuário? Discuta por que o controle de concorrência é necessário e dê exemplos informais.

21.2. Discuta os diferentes tipos de falhas. O que significa uma falha catastrófica?

21.3. Discuta as ações tomadas pelas operações read_item e write_item em um banco de dados.

21.4. Desenhe um diagrama de estado e discuta os estados típicos pelos quais uma transação passa durante a execução.

21.5. Para que é usado o log do sistema? Quais são os tipos característicos de registros em um log do sistema? O que são pontos de confirmação da transação e por que eles são importantes?

21.6. Discuta as propriedades de atomicidade, durabilidade, isolamento e preservação da consistência de uma transação de banco de dados.

21.7. O que é um schedule (histórico)? Defina os conceitos de schedules recuperáveis, sem cascata e estritos, e compare-os em matéria de sua facilidade de recuperação.

21.8. Discuta as diferentes medidas de equivalência de transação. Qual é a diferença entre equivalência de conflito e equivalência de visão?

21.9. O que é um schedule serial? O que é um schedule serializável? Por que um schedule serial é considerado correto? Por que um schedule serializável é considerado correto?

21.10. Qual é a diferença entre as suposições de gravação restrita e gravação irrestrita? Qual é mais realista?

21.11. Discuta como a serialização é usada para impor o controle de concorrência em um sistema de banco de dados. Por que a serialização às vezes é considerada muito restritiva como uma medida da exatidão para os schedules?

21.12. Descreva os quatro níveis de isolamento em SQL.

21.13. Defina as violações causadas por cada um dos seguintes itens: leitura suja, leitura não repetitiva e fantasmas.

Exercícios

21.14. Mude a transação T_2 da Figura 21.2(b) para

read_item(X);
$X := X + M$;
if $X > 90$ then exit
else write_item(X);

Discuta o resultado final dos diferentes schedules na Figura 21.3(a) e (b), onde $M = 2$ e $N = 2$, em relação às seguintes questões: a inclusão da condição acima muda o resultado final? O resultado obedece à regra de consistência implícita (de que a capacidade de X é 90)?

21.15. Repita o Exercício 21.14, acrescentando uma verificação em T_1 de modo que Y não exceda 90.

21.16. Inclua o commit da operação ao final de cada uma das transações T_1 e T_2 na Figura 21.2, e depois liste todos os schedules possíveis para as transações modificadas. Determine quais dos schedules são recuperáveis, quais são sem cascata e quais são estritos.

21.17. Liste todos os schedules possíveis para as transações T_1 e T_2 na Figura 21.2 e determine quais são serializáveis de conflito (corretos) e quais não são.

21.18. Quantos schedules *seriais* existem para as três transações da Figura 21.8(a)? Quais são eles? Qual é o número total de schedules possíveis?

21.19. Escreva um programa para criar todos os schedules possíveis para as três transações da Figura 21.8(a) e para determinar quais desses schedules são serializáveis de conflito e quais não são. Para cada schedule serializável de conflito, seu programa deverá imprimir o schedule e listar todos os schedules seriais equivalentes.

21.20. Por que uma instrução de fim de transação explícita é necessária em SQL, mas não uma instrução de início explícita?

21.21. Descreva situações em que cada um dos diferentes níveis de isolamento seriam úteis para o processamento de transação.

21.22. Qual dos seguintes schedules é serializável (de conflito)? Para cada schedule serializável, determine os schedules seriais equivalentes.

a. $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$

b. $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$

c. $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$

d. $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

21.23. Considere as três transações T_1, T_2 e T_3, e os schedules S_1 e S_2 a seguir. Desenhe os grafos de serialização (precedência) para S_1 e S_2 e indique se cada schedule é serializável ou não. Se um schedule for serializável, escreva o(s) schedule(s) serial(is) equivalente(s).

T_1: $r_1(X); r_1(Z); w_1(X);$
T_2: $r_2(Z); r_2(Y); w_2(Z); w_2(Y);$
T_3: $r_3(X); r_3(Y); w_3(Y);$
S_1: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y);$
S_2: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); w_2(Z); w_3(Y); w_2(Y);$

21.24. Considere os schedules S_3, S_4 e S_5 a seguir. Determine se cada schedule é estrito, sem cascata, recuperável ou não recuperável. (Determine a condição de facilidade de recuperação mais estrita que cada schedule satisfaz.)

S_3: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); w_2(Y); c_2;$
S_4: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y); c_1; c_2; c_3;$
S_5: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1; w_2(Z); w_3(Y); w_2(Y); c_3; c_2;$

Bibliografia selecionada

O conceito de serialização e as ideias relacionadas para manter a consistência em um banco de dados foram introduzidas em Gray et al. (1975). O conceito da transação de banco de dados foi discutido inicialmente em Gray (1981), que ganhou o cobiçado ACM Turing Award em 1998 por seu trabalho sobre transações de banco de dados e implementação de transações em SGBDs relacionais. Bernstein, Hadzilacos e Goodman (1988) focalizam as técnicas de controle de concorrência e recuperação em sistemas de banco de dados centralizados e distribuídos; trata-se de uma excelente referência. Papadimitriou (1986) oferece um ponto de vista mais teórico. Um grande livro de referência de mais de mil páginas, por Gray e Reuter (1993), oferece um ponto de vista mais prático dos conceitos e técnicas de processamento de transação. Elmagarmid (1992) oferece coleções de artigos de pesquisa sobre processamento de transação para aplicações avançadas. O suporte de transação em SQL é descrito em Date e Darwen (1997). A serialização de visão é definida em Yannakakis (1984). A facilidade de recuperação de schedules e a confiabilidade em bancos de dados são discutidas em Hadzilacos (1983, 1988).