Unidade III

7 GERENCIAMENTO DE TRANSAÇÃO

O gerenciamento de transações é um assunto que merece atenção especial. Os primeiros sistemas de informação utilizavam arquivos sequenciais indexados ou arquivos de acesso direto para guardar e atualizar seus dados.

Lembrete

Esses mecanismos eram limitados e não tinham recurso de gerenciamento, como encontramos atualmente nos sistemas de bancos de dados.

Com o início dos sistemas multiusuário, surgiu o problema da concorrência pelos dados e era imprescindível administrar todos os acessos, consultas e atualizações que ocorriam simultaneamente. Todos esses acessos geravam várias necessidades, como garantir o isolamento de cada transação, isto é, não deixar que uma transação interferisse na outra, o que poderia ocasionar atualizações incorretas, resultando na inconsistência dos dados. Além disso, era preciso garantir que cada transação fosse atômica, ou seja, todas as transações deveriam ser concluídas integralmente ou não seriam efetivadas, causando inconsistência nos dados.

A solução inicial para esse problema foi a gestão de transações através da aplicação, ou seja, o próprio sistema de informação era responsável por gerenciar cada transação finalizada. Isso era muito complicado e, enquanto analistas e projetistas lutavam para implementar verificações para garantir a integridade das transações, havia, ainda, problemas ambientais ou de hardware, além do controle do aplicativo. Ou seja, a gestão de transações executada pela aplicação era falha e os problemas foram apenas minimizados e não resolvidos.

7.1 Transações

Uma transação é uma unidade lógica de operações de banco de dados. Uma transação pode consistir em uma ou mais operações de banco de dados que podem ser lidas ou atualizadas. Podem ser inserção (**insert**), atualização (**update**) ou exclusão (**delete**). As operações de banco de dados que compõem uma transação podem estar embutidas nos programas ou podem ser executadas interativamente usando SQL em um programa de banco de dados. Em outras palavras, as transações podem ser enviadas para um SGBD por programas aplicativos ou programas de banco de dados e devem ser processadas de forma apropriada.

Observação

Uma maneira de caracterizar uma transação é especificar seu início e fim, ou seja, os limites da transação, com a utilização das instruções <BEGIN TRANSACTION> e <END TRANSACTION>. Dessa forma, todas as operações de banco de dados entre essas instruções são reconhecidas como parte da transação.

Toda transação deve respeitar as quatro propriedades que garantem o correto funcionamento do banco de dados e impedem que os dados sejam perdidos ou corrompidos no processamento. Essas propriedades são conhecidas pela sigla Acid:

- **Atomicidade**: cada transação deve ser atômica, o que significa que não pode ser dividida ou fragmentada. A ideia é que todas as operações do banco de dados que o compõem sejam executadas como se fossem uma única operação. Se qualquer operação de banco de dados falhar, toda a transação deverá ser desfeita. Depois que todas as operações do banco de dados que compõem a transação forem concluídas com êxito, a transação poderá ser confirmada.

- **Consistência**: cada transação deve deixar o banco de dados em um estado consistente após a execução. Isso significa que a transação deve atender a todas as regras e restrições definidas no banco de dados, que incluem regras de integridade referencial, regras de domínio para valores de coluna permitidos, definição de chave primária, índices exclusivos e colunas obrigatórias.

- **Isolamento**: cada transação deve ser isolada de outras transações no banco de dados. Os resultados parciais de cada transação não devem estar disponíveis para outras transações. A ideia é que nenhuma transação possa atrapalhar a execução de outra no mesmo banco de dados.

- **Durabilidade**: cada transação tem resultados permanentes no banco de dados e só pode ser desfeita na próxima transação.

7.2 Estado da transação

Uma transação nem sempre completa sua execução com sucesso. Quando isso acontece, ela é considerada abortada. Se tivermos de assegurar a propriedade de atomicidade, uma transação abortada não deverá mudar o estado do banco de dados. Dessa forma, quaisquer mudanças que a transação abortada fez no banco de dados deverá ser desfeita. No momento em que todas as mudanças causadas por uma transação abortada tiverem sido desfeitas, consideraremos a transação revertida (rolled back). O esquema de recuperação deve gerenciar as transações revertidas. Normalmente é executado mantendo-se um log. As modificações no banco de dados realizadas por uma transação são primeiro registradas no log. Registramos o identificador da transação que fará a modificação, o identificador do item de dados que será modificado, o valor antigo (antes da modificação) e o novo valor (após a modificação) do item de dados. Apenas depois é que o banco de dados pode realmente

ser modificado. O log é importante pois oferece a possibilidade de refazer uma modificação para assegurar a atomicidade e a durabilidade, assim como a possibilidade de desfazer uma modificação para garantir a atomicidade no caso de alguma falha durante a execução da transação.

Saiba mais

Para aprofundar o conhecimento em recuperação de dados, recomendamos a leitura do subcapítulo de recuperação de dados do livro *Banco de dados* e do capítulo 19 – Sistema de recuperação, do livro *Sistema de banco de dados*.

ALVES, W. P. *Banco de dados*. São Paulo: Saraiva, 2014.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Sistema de banco de dados*. Rio de Janeiro: Elsevier, 2020.

Uma transação cuja execução foi concluída com sucesso é considerada confirmada (committed). A transação que fez as atualizações altera o banco de dados para um novo estado persistente, que deve ser preservado mesmo se o sistema falhar.

Assim que uma transação é confirmada, não podemos mais reverter seus efeitos no banco de dados tentando abortá-la. Para desfazer os efeitos de uma transação concluída, é necessário executar uma transação de compensação. Por exemplo, se uma transação somasse 500 no salário de um funcionário, a transação de compensação subtrairia 500 do salário. Nem sempre é possível criar essa transação de compensação. A responsabilidade de escrever e executar uma transação de compensação fica totalmente para o usuário, e não é tratada pelo sistema de banco de dados.

Para especificar o que significa para uma transação bem-sucedida, vamos criar um modelo simples de classificação de transação onde analisaremos alguns estados:

- **Ativo**: o estado inicial. A transação permanece nesse estado enquanto está executando.
- **Parcialmente confirmado**: depois que a instrução final foi executada.
- **Falho**: depois da descoberta de que a execução não pode mais prosseguir.
- **Abortado**: depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação.
- **Confirmado**: após o término bem-sucedido.

Baseando-nos nos possíveis estados de uma transação, apresentamos um diagrama de estados na figura a seguir. Uma transação foi confirmada somente se ela entrou no estado confirmado. Uma transação foi abortada somente se ela entrou no estado abortado. Uma transação é considerada terminada se tiver sido confirmada ou abortada.

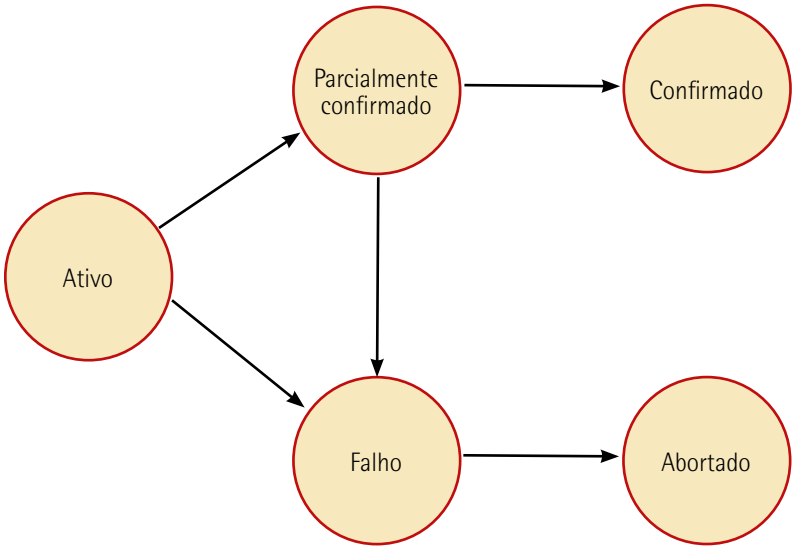

Figura 18 – Diagrama de estado de uma transação

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 455).

Iniciamos a transação no estado ativo. No momento em que ela termina sua última instrução, entra no estado parcialmente confirmado. Isso significa que a transação completou sua execução, mas ainda pode ser abortada, pois a saída real ainda pode estar temporariamente residindo na memória principal e, por isso, uma falha de hardware pode impedir seu término bem-sucedido.

O SGBD grava dados suficientes em disco de modo que, se ocorrer uma falha, as atualizações feitas pela transação possam ser refeitas quando o sistema reiniciar após a falha. Apenas quando o último dado for gravado, a transação entrará no estado confirmado.

Uma transação entra no estado de falha após o sistema determinar que a transação não pode mais continuar sua execução normal (por exemplo, devido a erros de hardware ou de lógica). Tal transação deve ser cancelada. Depois disso, ele entra em um estado abortado. Neste ponto, o sistema tem duas escolhas possíveis:

- **Reiniciar a transação**: deve-se, todavia, observar o motivo do aborto e considerar se o erro foi de hardware ou se foi produzido por problemas na lógica interna da transação. Caso a transação seja reiniciada, ela é considerada uma nova transação.

- **Matar a transação**: normalmente isso pode acontecer se o problema foi causado por algum erro lógico interno, que poderá ser corrigido somente com a reescrita do programa de aplicação; ou porque a entrada foi incorreta, ou porque os dados desejados não foram encontrados no banco de dados.

7.3 Execuções simultâneas

Os sistemas de processamento de transações geralmente permitem que várias transações sejam processadas ao mesmo tempo. No entanto, essa permissão causa alguns problemas de consistência de dados. Assegurar a consistência com a execução de transações simultâneas requer trabalho adicional. É muito mais simples exigir que as transações sejam executadas sequencialmente, iniciando cada uma somente após a conclusão da anterior. No entanto, existem duas boas razões para permitir a concorrência:

- **Melhor vazão (throughput) e utilização de recursos**: a transação possui várias etapas. Algumas incluem funções de E/S; outras estão relacionados ao desempenho do processador. O processador e os discos de um sistema de computador podem trabalhar em paralelo. A operação de E/S pode ser executada em paralelo com o processamento da CPU. Esse paralelismo do processador e do sistema de E/S pode ser explorado para executar diversas transações em paralelo. Caso uma operação de leitura ou gravação estiver em execução em um disco em nome de uma transação, outra transação pode estar sendo executada no processador, enquanto outro disco executa uma operação de leitura ou gravação em nome de uma terceira transação. Isso aumenta a vazão (throughput), dessa forma, aumenta o número de transações realizadas em um determinado período. As taxas de uso de CPU e do disco também aumentam proporcionalmente. Em outras palavras, a CPU e o disco passam menos tempo ociosos ou sem fazer trabalho útil.

- **Tempo de espera reduzido**: um sistema pode ter várias transações, algumas curtas e outras longas. Quando as transações são executadas sequencialmente, uma transação curta pode aguardar a conclusão da transação longa anterior, o que pode causar atrasos imprevisíveis na execução da transação. Se as transações estiverem sendo executadas em diferentes partes do banco de dados, é melhor executá-las simultaneamente, compartilhando ciclos de CPU e uso de disco entre elas. A execução simultânea reduz atrasos imprevisíveis na execução de transações. Além disso, reduz o tempo médio de resposta (o tempo médio que uma transação leva para ser concluída após o envio).

A motivação para usar execução concorrente em um banco de dados é essencialmente a mesma que a motivação para usar multiprogramação em um sistema operacional. Se várias transações forem executadas simultaneamente, a propriedade de isolamento pode ser violada, resultando na perda de consistência do banco de dados, independentemente da precisão de cada transação individual.

O sistema de banco de dados deve controlar a interação entre transações concorrentes, para que elas não destruam a consistência do banco de dados. Ele faz isso por meio de vários mecanismos chamados sistemas de controle concorrência.

Considere o cadastro de uma conta bancária e um conjunto de transações que acessam e atualizam essas contas. Considere que T1 e T2 sejam duas transações que movimentam os saldos dessas contas. A transação T1 transfere 500 da conta C1 para a conta C2. Ela é definida como:

Tabela 20 – Transação T1

T1:
read(C1);
C1 := C1 – 500;
write(C1);
read(C2);
C2 := C2 + 500;
write(C2).

A transação T2 transfere 20% do saldo da conta C1 para a conta C2. Ela é definida como:

Tabela 21 – Transação T2

T2:
read(C1);
temp := C1 * 0.2;
C1 := C1 – temp;
write(C1);
read(C2);
C2 := C2 + temp;
write(C2).

Vamos supor que os valores atuais das contas C1 e C2 sejam 1.000 e 2.000, respectivamente. Suponha, também, que as duas transações sejam executadas uma de cada vez na ordem T1 e em seguida T2. A sequência de etapas de instrução deve estar em ordem cronológica de cima para baixo, com as instruções de T1 aparecendo na coluna esquerda e as instruções de T2 aparecendo na coluna direita. Os valores finais das contas C1 e C2 após a execução são 400 e 2.600, respectivamente. A quantia nas contas C1 e C2 – ou seja, a soma C1 + C2 – é preservada depois da execução das duas transações. Na tabela a seguir apresentamos o **Schedule 1** serial que apresenta a sequência de execução T1 e T2:

Tabela 22 – Schedule 1: T1 seguido de T2

T1:	T2:
read(C1);	
C1 := C1 – 500;	
write(C1);	
read(C2);	
C2 := C2 + 500;	
write(C2).	
commit	
	read(C1);

T1:	T2:
	temp := C1 * 0.2;
	C1 := C1 – temp;
	write(C1);
	read(C2);
	C2 := C2 + temp;
	write(C2).
	commit

De modo semelhante, se as transações forem executadas uma de cada vez na ordem T2 seguida por T1, então a sequência de execução é a da tabela a seguir. Conforme esperamos, a soma C1 + C2 é preservada, e os valores finais das contas C1 e C2 agora são 300 e 2.700, respectivamente.

Tabela 23 – Schedule 2: T2 seguido de T1

T1:	T2:
	read(C1);
	temp := C1 * 0.2;
	C1 := C1 – temp;
	write(C1);
	read(C2);
	C2 := C2 + temp;
	write(C2).
	commit
read(C1);	
C1 := C1 – 500;	
write(C1);	
read(C2);	
C2 := C2 + 500;	
write(C2).	
commit	

Os ciclos de execução descritos são chamados de schedules. Eles representam a ordem cronológica de execução de comandos no sistema. Obviamente, os schedules para uma série de transações devem não apenas conter todas as instruções para essas transações, mas também preservar a ordem em que as instruções aparecem para cada evento individual. Por exemplo, na transação T1, o comando write(C1) deve aparecer antes do comando read(C2) em qualquer schedule válido.

Os schedules 1 e 2 são sequenciais; cada schedule consiste em uma sequência de instruções com diversas transações, cujas instruções aparecem juntas nesse schedule. A fórmula da análise combinatória para um conjunto de n transações, será n! (n fatorial) diferentes schedules seriais válidos

Saiba mais

Para saber mais sobre métodos de análise de algoritmos, recomendamos o capítulo 2 – Combinatória, do livro *Métodos para análise de algoritmos*.

DOBRUSHKIN, V. A. *Métodos para análise de algoritmos*. Rio de Janeiro: LTC, 2012.

Se o sistema de banco de dados executa diversas transações ao mesmo tempo, o schedule correspondente não precisa mais ser serializado. Se duas transações estiverem sendo executadas ao mesmo tempo, o sistema operacional pode executar uma por um tempo, então trocar de contexto, executar outra por um tempo, então retornar brevemente à primeira transação e assim por diante. Com várias transações, o tempo de CPU é compartilhado entre todas as transações.

Diversas sequências de execução são possíveis, pois as diversas instruções das duas transações podem ser intercaladas. Geralmente, não é possível prever a exata quantia de instruções de uma transação que serão executadas antes que a CPU passe para outra transação.

Vamos supor que as duas transações sejam executadas simultaneamente. Um schedule possível aparece na tabela a seguir. Depois que essa execução acontece, chegamos ao mesmo estado daquele em que as transações são executadas em série na ordem T1 seguida por T2. A soma C1 + C2 é realmente preservada.

Tabela 24 – Schedule 3: um schedule simultâneo equivalente ao schedule 1

T1:	T2:
read(C1);	
C1 := C1 – 500;	
write(C1);	
	read(C1);
	temp := C1 * 0.2;
	C1 := C1 – temp;
	write(C1);
read(C2);	
C2 := C2 + 500;	
write(C2).	
commit	
	read(C2);
	C2 := C2 + temp;
	write(C2).
	commit

Nem todas as execuções simultâneas resultam em um estado correto. Para ilustrar, considere o schedule da tabela a seguir. Depois da execução desse schedule, chegamos a um estado em que os valores finais das contas C1 e C2 são 500 e 2.200, respectivamente. Esse estado final é um estado inconsistente, pois perdemos 300 no processo da execução simultânea. De fato, a soma C1 + C2 não é preservada pela execução das duas transações.

Tabela 25 – Schedule 4: um schedule simultâneo resultando em um estado inconsistente

T1:	T2:
read(C1);	
C1 := C1 – 500;	
	read(C1);
	temp := C1 * 0.2;
	C1 := C1 – temp;
	write(C1);
	read(C2);
write(C1);	
read(C2);	
C2 := C2 + 500;	
write(C2).	
commit	
	C2 := C2 + temp;
	write(C2).
	commit

Se o gerenciamento da execução simultânea for de inteira responsabilidade do sistema operacional, muitos schedules – incluindo os que deixam o banco de dados em um estado inconsistente, como aquele que descrevemos na tabela anterior – são possíveis. A tarefa do sistema de banco de dados é assegurar que qualquer schedule executado mantenha o banco de dados em um estado consistente. O componente de controle de concorrência do sistema de banco de dados executa essa tarefa.

Podemos assegurar a consistência do banco de dados sob execução simultânea, cuidando para que qualquer schedule executado tenha o mesmo resultado do schedule que poderia ter acontecido sem nenhuma execução simultânea. Ou seja, ele precisa ser equivalente a um schedule serial. Esses schedules são chamados schedules serializáveis.

7.4 Facilidade de recuperação

Considere o schedule parcial 5 na tabela 26, em que T4 é uma transação que realiza apenas uma instrução: read(C1). Denominamos isso de schedule parcial, pois não incluímos uma operação commit ou abort para T3. Observe que T4 é confirmada imediatamente após a execução da instrução read(C1). Assim, T4 é confirmada enquanto T3 ainda está no estado ativo. Agora, suponha que T3 falhe antes de

ser confirmada. Como T4 leu o valor do item de dados C1 escrito por T3, dizemos que T4 é dependente de T3. Por causa disso, temos de abortar T4 para garantir a atomicidade. Porém, T4 já foi confirmada e não pode ser abortada. Assim, temos uma situação em que é impossível se recuperar corretamente da falha de T3.

O schedule 5 é um exemplo de schedule não recuperável. Um schedule recuperável é aquele em que, para cada par de transações Ti e Tj tal que Tj leia um item de dados previamente escrito por Ti, a operação commit de Ti apareça antes da operação commit de Tj. Para que o exemplo do schedule 5 seja recuperável, T4 teria de adiar a confirmação até que T3 fosse confirmada.

Tabela 26 – Schedule 5: um schedule não recuperável

T3:	T4:
read(C1);	
write(C1);	
	read(C1);
	commit
read(C2);	

7.5 Implementação do isolamento

Existem várias formas de controle de concorrência que podem ser utilizadas para assegurar que, mesmo quando diversas transações são executadas ao mesmo tempo, somente os schedules aceitáveis sejam gerados, independentemente de como o sistema operacional compartilhe o tempo dos recursos entre as transações.

Considere esse padrão como um exemplo trivial de um esquema de controle de concorrência: uma transação bloqueia todo o banco de dados antes da execução e libera o bloqueio após a execução. Mesmo que uma transação retenha o bloqueio, nenhuma outra transação tem permissão para adquirir o bloqueio e, portanto, todas devem esperar que o bloqueio seja liberado. Como resultado do sistema de bloqueio, apenas uma transação pode ser executada por vez. Portanto, apenas schedules seriais são criados. Eles são organizados trivialmente em conjuntos e é fácil verificar que também não são schedules em cascata.

Esse esquema de controle de simultaneidade leva a um desempenho ruim porque força as transações a esperarem até que as transações anteriores sejam concluídas antes de poderem começar. Em outras palavras, oferece menor concorrência.

Os métodos de controle de simultaneidade visam garantir um alto nível de concorrência e garantir que todos os schedules criados não sejam serializáveis por conflito ou visão e em cascata.

8 CONTROLE DE CONCORRÊNCIA

Uma das principais características de uma transação é o isolamento. No entanto, se várias transações forem executadas simultaneamente no banco de dados, o atributo de isolamento não poderá mais ser mantido. Para garantir isso, o sistema deve monitorar as interações entre transações concorrentes. Esse controle é obtido por meio de diversos mecanismos chamados sistemas de controle simultâneos.

Existe uma variedade de esquemas de controle de concorrência. Nenhum é especificamente o melhor. Cada um tem suas vantagens. Na prática, os dois esquemas mais usados são o bloqueio de duas fases e o isolamento baseado em instantâneo (também simplesmente chamado de isolamento de instantâneo, ou snapshot isolation).

8.1 Protocolos baseados em bloqueio

Uma maneira de garantir o isolamento é exigir que os itens de dados sejam usados de maneira mutuamente exclusiva, ou seja, se uma transação usa um objeto de dados, nenhuma outra transação pode modificá-lo. A maneira mais comum de implementar esse requisito é permitir que uma transação acesse um objeto de dados somente se estiver atualmente bloqueado nesse objeto.

8.1.1 Bloqueios (locks)

Veremos dois modos como um item de dados pode ser bloqueado:

- **Compartilhado**: se uma transação Ti tiver obtido um bloqueio no modo compartilhado (indicado por S do inglês, shared) sobre o item Q, então Ti pode ler, mas não pode escrever Q.

- **Exclusivo**: se uma transação Ti tiver obtido um bloqueio no modo exclusivo (indicado por X do inglês, exclusive) sobre o item Q, então Ti pode ler e escrever Q.

Cada transação deve solicitar um bloqueio em Q de maneira adequada, dependendo do tipo de operação que está sendo executada em Q. A transação faz uma solicitação ao gerente para controle de concorrência. Uma transação só pode continuar depois que o gerenciador de controle de simultaneidade bloquear a transação. Usando esses dois modos de bloqueio, várias transações podem ler o objeto de dados, mas o acesso de gravação é limitado a apenas uma transação por vez.

Em geral, dado um conjunto de modos de bloqueio, podemos definir uma função de compatibilidade para eles da seguinte forma: deixe C1 e C2 representar qualquer forma de inibição. Suponha que o evento Ti solicite um bloqueio de modo C1 para o destino Q, no qual o evento Tj (Ti ≠ Tj) detenha no momento um bloqueio de modo C2. Se o evento Ti pode aceitar imediatamente o bloqueio Q, apesar da existência do bloqueio no estado C2, então o estado C1 corresponde ao estado C2. Essa função pode ser convenientemente representada como uma matriz. A proporção de correspondência dos dois estados de inibição processados não é mostrada na tabela de partição a seguir. O elemento da matriz comp (C1, C2) é verdadeiro se, e somente se, o modo C1 for compatível com o modo C2.

Tabela 27 - Matriz comp de compatibilidade de bloqueio

	S	X
S	True	False
X	False	False

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 471).

Observe que o modo compartilhado é compatível com o modo compartilhado, mas não com o modo exclusivo. Vários espaços compartilhados podem ser bloqueados simultaneamente (com transações diferentes) em um determinado objeto de dados. O próximo bloqueio de espaço exclusivo deve aguardar até que os bloqueios de espaço compartilhado atualmente mantidos sejam liberados.

A transação solicita um bloqueio compartilhado no objeto de dados Q, executando a instrução lock-S(Q). Portanto, o evento solicita um bloqueio exclusivo com o comando lock-X(Q). O evento pode desbloquear os dados Q com o comando unlock(Q).

Para acessar um objeto de dados, o evento Ti deve primeiro bloquear o objeto em questão. Se os dados já estiverem bloqueados por outra transação que não esteja em um estado compatível, o gerenciador de controle de simultaneidade não liberará o bloqueio até que todos os bloqueios incompatíveis mantidos por outras transações tenham sido liberados. Assim, Ti espera que todos os bloqueios incompatíveis de outras transações sejam liberados.

A transação Ti pode desbloquear dados anteriormente bloqueados em algum ponto. Observe que uma transação deve manter um bloqueio em um objeto de dados enquanto usar o objeto. Além disso, nem sempre desejamos que um evento abra um objeto de dados imediatamente após finalmente alcançar esse objeto de dados, porque a serialização não pode necessariamente ser garantida.

Considere novamente o sistema bancário simplificado, no qual C1 e C2 sejam duas contas acessadas pelas transações T5 e T6. A transação T5 transfere 500 da conta C2 para a conta C1 (tabela a seguir). A transação T6 apresenta a quantia total nas contas C1 e C2 – ou seja, a soma C1 + C2 (tabela 29).

Tabela 28 – Transação T5

T5:
lock-X(C2);
read(C2);
C2 := C2 – 50;
write(C2);
unlock(C2);
lock-X(C1);
read(C1);
C1 := C1 + 50;
write(C1);
unlock(C1);

Tabela 29 – Transação T6

T6:
lock-S(C1);
read(C1);
unlock(C1);
lock-S(C2);
read(C2);
unlock(C2);
display(C1+C2)

Suponha que os valores das contas C1 e C2 sejam 100 e 200, respectivamente. Se essas duas transações forem executadas em série ou na ordem T5, T6 ou na ordem T6, T5, então a transação T6 mostrará o valor 300. No entanto, se essas transações forem executadas simultaneamente, o schedule 6, na tabela a seguir, é possível. Nesse caso, a transação T6 mostra 250, que é incorreto. O motivo para esse engano é que a transação T5 desbloqueou o item de dados C2 muito cedo e, como resultado, T6 viu um estado inconsistente.

Tabela 30 – Schedule 6

T5:	T6:	Gerenciador de controle de concorrência
lock-X(C2);		
		grant-X(C2,T5)
read(C2);		
C2 := C2 – 50;		
write(C2);		
unlock(C2);		
	lock-S(C1);	
		grant-S(C1,T6)
	read(C1);	
	unlock(C1);	
	lock-S(C2);	
		grant-X(C2,T6)
	read(C2);	
	unlock(C2);	
	display(C1+C2)	
lock-X(C1);		
		grant-X(C1,T5)
read(C1);		
C1 := C1 + 50;		
write(C1);		
unlock(C1);		

O schedule apresenta as ações realizadas pelos eventos e os pontos em que o gerenciador de controle simultâneo trava. Um evento que faz uma solicitação de bloqueio não pode executar sua próxima operação até que o gerenciador de controle simultâneo tenha concedido o bloqueio. Portanto, o bloqueio deve ser concedido durante o tempo entre a solicitação e a próxima operação de transação, porque é precisamente nessa área que o bloqueio será concedido. Assim, podemos assumir com segurança que o bloqueio será concedido imediatamente antes da próxima operação da transação.

Vamos supor que o desbloqueio seja adiado para o final da transação. A transação T7 corresponde a T5 com desbloqueio adiado (tabela a seguir). A transação T8 corresponde a T6 com o desbloqueio adiado (tabela 32).

Tabela 31 – Transação T7 (transação T5 com desbloqueio adiado)

T7:
lock-X(C2);
read(C2);
C2 := C2 – 50;
write(C2);
lock-X(C1);
read(C1);
C1 := C1 + 50;
write(C1);
unlock(C2);
unlock(C1);

Tabela 32 – Transação T8 (transação T6 com desbloqueio adiado)

T6:
lock-S(C1);
read(C1);
lock-S(C2);
read(C2);
display(C1+C2)
unlock(C1);
unlock(C2);

Note que a sequência de leituras e escritas no schedule 6, que leva à exibição de um total incorreto de 250, não é mais possível com T7 e T8. Outros schedules são possíveis. T8 não imprimirá um resultado inconsistente.

Infelizmente, o bloqueio pode levar a uma situação indesejável. Considere o schedule parcial da tabela a seguir para T7 e T8. Como T7 está mantendo um bloqueio no modo exclusivo sobre C2, e T8 está solicitando um bloqueio no modo compartilhado sobre C2, T8 está esperando que T7 desbloqueie C2. De modo semelhante, como T8 está mantendo um bloqueio no modo compartilhado sobre C1, e

T7 está solicitando um bloqueio no modo exclusivo sobre C1, T7 está esperando que T8 desbloqueie C1. Assim, chegamos a um estado em que nenhuma dessas transações pode prosseguir com sua execução normal. Essa situação é chamada de impasse (deadlock). Quando ocorre impasse, o sistema precisa reverter uma das duas transações. Quando uma transação tiver sido revertida, os itens de dados que estavam bloqueados por essa transação são desbloqueados. Esses itens de dados, então, ficam disponíveis para a outra transação, que pode continuar com sua execução.

Tabela 33 – Schedule 7

T7:	T8:
lock-X(C2);	
read(C2);	
C2 := C2 – 50;	
write(C2);	
	lock-S(C1);
	read(C1);
	lock-S(C2);
lock-X(C1);	

Se não implementarmos o bloqueio ou desbloqueio de objetos de dados o mais rápido possível após a leitura ou gravação, podemos acabar com estados inconsistentes. Se não desbloquearmos um item de dados antes de solicitar o bloqueio de outro item de dados, pode ocorrer um impasse. Existem algumas maneiras de evitar ficar preso em algumas dessas situações. Geralmente, o impasse é um mal necessário se quisermos evitar estados conflitantes. Os bloqueios são definitivamente melhores do que os estados inconsistentes porque podem ser tratados por rollback de transações, enquanto os estados inconsistentes podem causar problemas do mundo real, que o sistema de banco de dados não pode manipular.

Considere que {T0, T1, ..., Tn} sejam um conjunto de transações participando de um schedule S. Dizemos que Ti precede Tj em S, escrito como Ti → Tj. Se houver um item de dados Q tal que Ti tenha mantido o modo de bloqueio A sobre Q, e Tj tenha mantido o modo de bloqueio B sobre Q mais tarde, e comp(C1,C2) = false. Se Ti → Tj, então essa precedência implica que, em qualquer schedule serial equivalente, Ti precisa aparecer antes de Tj. Os conflitos entre instruções correspondem à não compatibilidade de modos de bloqueio.

Dizemos que um schedule S é válido sob um protocolo específico de bloqueio. Se S for um schedule possível para um conjunto de transações que seguem as regras do protocolo de bloqueio, dizemos, então, que um protocolo de bloqueio garante a serializabilidade por conflito se, e somente se, todos os schedules válidos forem serializáveis por conflito.

8.1.2 Concessão de bloqueios (granting of locks)

Quando uma transação solicita um bloqueio sobre um item de dados em determinado modo e nenhuma outra transação possui um bloqueio sobre o mesmo item de dados em um modo em conflito, o bloqueio pode ser concedido. Porém, deve-se ter o cuidado de evitar o cenário a seguir: suponha que uma transação T6 tenha um bloqueio no modo compartilhado sobre um item de dados, e outra transação T5 solicite um bloqueio no modo exclusivo sobre o item de dados. T5 tem de esperar que T6 libere o bloqueio no modo compartilhado. Nesse meio tempo, uma transação T7 pode solicitar um bloqueio no modo compartilhado sobre o mesmo item de dados. A solicitação de bloqueio é compatível com o bloqueio concedido a T6, assim, T7 pode receber o bloqueio no modo compartilhado. Nesse ponto, T6 pode liberar o bloqueio, mas ainda T5 precisa esperar que T7 termine. Novamente, porém, pode haver uma nova transação T8 que solicite um bloqueio no modo compartilhado sobre o mesmo item de dados, recebendo o bloqueio antes que T7 o libere. De fato, é possível que haja uma sequência de transações em que cada uma solicite um bloqueio no modo compartilhado sobre o item de dados, e cada transação libere o bloqueio pouco depois que ele foi concedido, mas T5 nunca receba o bloqueio no modo exclusivo sobre o item de dados. A transação T5 pode nunca fazer progresso e é considerada estagnada.

Podemos evitar a estagnação de transações concedendo bloqueios da seguinte maneira: no momento em uma transação Ti solicita um bloqueio sobre um item de dados Q em um modo específico M, o gerenciador de controle de concorrência concede o bloqueio desde que:

- não haja outra transação mantendo um bloqueio sobre Q em um modo que entre em conflito com M.

- não exista outra transação que esteja esperando por um bloqueio sobre Q e que fez sua solicitação de bloqueio antes de Ti.

Dessa forma, uma solicitação de bloqueio nunca será bloqueada por outra feita posteriormente.

8.1.3 Protocolo de bloqueio em duas fases

Um protocolo que garante a serializabilidade é o protocolo de bloqueio em duas fases. Esse protocolo requer que cada transação envie solicitações de bloqueio e desbloqueio em duas fases:

- **Fase de crescimento**: uma transação pode obter bloqueios, mas não pode liberar nenhum bloqueio.

- **Fase de encolhimento**: uma transação pode liberar bloqueios, mas não pode obter novos bloqueios.

Primeiramente, uma transação está na fase de crescimento. A transação adquire bloqueios caso necessite. Quando a transação libera um bloqueio, ela entra na fase de encolhimento, e não pode emitir mais solicitações de bloqueio. Por exemplo, as transações T7 e T8 são de duas fases. Por outro lado, as transações T5 e T6 não são de duas fases. Podemos observar que as instruções de desbloqueio não precisam aparecer no final da transação. Por exemplo, no caso da transação T7, poderíamos mover a instrução unlock(B) para logo depois da instrução lock-X(A) e ainda reter a propriedade de bloqueio em duas fases.

Conseguimos demostrar que o protocolo de bloqueio em duas fases assegura a serializabilidade por conflito. Considere qualquer transação. O ponto no schedule em que a transação obteve seu bloqueio final é denominado ponto de bloqueio da transação. As transações podem ser ordenadas de acordo com seus pontos de bloqueio – essa ordenação, de fato, é uma ordenação de serializabilidade para as transações.

O bloqueio em duas fases não garante a isenção de impasse (deadlock). Observe que as transações T7 e T8 são de duas fases, mas no schedule 7 (tabela anterior) elas estão em impasse.

Lembrete

Os schedules, além de serem serializáveis, não devem estar em cascata.

Com o bloqueio em duas fases, pode ocorrer rollback em cascata. Considere o schedule parcial da tabela a seguir. Cada transação observa o protocolo de bloqueio em duas fases, mas a falha de T10 após a etapa read(A) de T12 leva a um rollback em cascata de T11 e T12.

Tabela 34 – Schedule 8: parcial sob o bloqueio em duas fases

T10:	T11:	T12:
lock-X(C1);		
read(C1);		
lock-S(C2);		
read(C2);		
write(C1);		
unlock(C1);		
	lock-X(C1);	
	read(C1);	
	write(C1);	
	unlock(C1);	
		lock-S(C1);
		read(C1);

Os rollbacks em cascata podem ser evitados modificando-se o bloqueio de duas fases, conhecido como protocolo de bloqueio de duas fases estrito. Esse protocolo requer não apenas que o bloqueio seja de duas fases, mas também que todos os bloqueios de estado exclusivos na transação sejam mantidos até que a transação seja confirmada. Esse requisito garante que todos os dados gravados por uma transação não confirmada sejam bloqueados em um estado exclusivo até que a transação seja confirmada, impedindo que outras transações leiam os dados.

Outra variante do bloqueio de duas fases é o protocolo estrito de bloqueio de duas fases, que exige que todos os bloqueios sejam mantidos até que a transação seja confirmada. Podemos verificar facilmente que as transações podem ser ordenadas em sua sequência de vinculação, usando o bloqueio de duas fases estrito.

8.2 Protocolos baseados em timestamp

Os protocolos baseados em timestamp determinam a ordem entre cada par de transações em conflito em tempo de execução, começando a partir do primeiro bloqueio envolvendo modos incompatíveis que os dois membros do par solicitam. Outro método de determinar a ordem de serializabilidade consiste em selecionar uma ordenação das transações com antecedência. O método mais comum para fazer isso é usar um esquema de ordenação por timestamp.

8.2.1 Timestamps

A cada transação Ti no sistema, associamos um timestamp fixo exclusivo, indicado por TS(Ti). Esse timestamp (registro de data e hora) é atribuído pelo sistema de banco de dados antes que a transação Ti inicie sua execução. Se uma transação Ti tiver recebido o timestamp TS(Ti), e uma nova transação Tj entrar no sistema, então TS(Ti) < TS(Tj). Existem dois métodos para elaborar esse esquema:

- use o valor do clock do sistema como timestamp, ou seja, o timestamp de uma transação é igual ao valor do clock quando a transação entrar no sistema;

- use um contador lógico que é incrementado após um novo timestamp ter sido atribuído, ou seja, o timestamp de uma transação é igual ao valor do contador quando a transação entrar no sistema.

Os timestamps das transações determinam a ordem de serializabilidade. Assim, se TS(Ti) < TS(Tj), o sistema precisa garantir que o schedule produzido seja equivalente a um schedule serial em que a transação Ti aparece antes da transação Tj.

Para implementar esse esquema, associamos a cada item de dados Q dois valores de timestamp:

- W-timestamp(Q), que indica o maior timestamp de qualquer transação que executou write(Q) com sucesso;

- R-timestamp(Q) que indica o maior timestamp de qualquer transação que executou read(Q) com sucesso.

Esses timestamps são atualizados sempre que uma nova instrução read(Q) ou write(Q) é executada.

8.2.2 Protocolo de ordenação por timestamp

O protocolo de ordenação por timestamp assegura que quaisquer transações read e write em conflito sejam executadas em ordem de timestamp. Esse protocolo opera da seguinte forma:

- Suponha que a transação Ti emita read(Q):

  – se TS(Ti) < W-timestamp(Q), então Ti precisa ler um valor de Q que já foi sobrescrito. Logo, a operação read é rejeitada, e Ti é revertida;

– se TS(Ti) ≥ W-timestamp(Q), então a operação read é executada, e R-timestamp(Q) é definido como R-timestamp(Q) ou TS(Ti) o que for maior.

- Suponha que a transação Ti emita write(Q):

– se TS(Ti) < R-timestamp(Q), então o valor de Q que Ti está produzindo foi necessário anteriormente e o sistema considerou que esse valor nunca seria produzido. Logo, o sistema rejeita a operação write e reverte Ti;

– se TS(Ti) < W-timestamp(Q), então Ti está tentando escrever um valor obsoleto de Q. Logo, o sistema rejeita essa operação write e reverte Ti;

– caso contrário, o sistema executa a operação write e define W-timestamp(Q) como TS(Ti).

Se uma transação Ti for revertida (rolled back) pelo esquema de controle de concorrência como resultado da emissão de uma operação read ou write, o sistema lhe atribui um novo timestamp e a reinicia.

Para ilustrar esse protocolo, consideramos as transações T13 e T14. A transação T13 apresenta o conteúdo das contas C1 e C2:

Tabela 35 – Transação T13

T13:
read(C2);
read(C1);
display(C1 + C2)

A transação T14 transfere 50 da conta C2 para a conta C1 e, depois, apresenta o conteúdo de ambas:

Tabela 36 – Transação T14

T14:
read(C2);
C2 := C2 – 50;
write(C2);
read(C1);
C1 : = C1 + 50;
write(C1);
display(C1 + C2)

Na apresentação de schedules sob o protocolo de timestamp, vamos considerar que uma transação recebeu um timestamp imediatamente antes de sua primeira instrução. Assim, no schedule 9 da tabela a seguir, TS(T15) < TS(T16), o schedule é possível sob o protocolo de timestamp.

Tabela 37 – Schedule 9

T15:	T16:
read(C2);	
	read(C2);
	C2 := C2 – 50;
	write(C2);
read(C1);	
	read(C1);
display(C1 + C2)	
	C1 := C1 + 50;
	write(C1);
	display(C1 + C2)

Observemos que a execução anterior também pode ser produzida pelo protocolo de bloqueio em duas fases. Entretanto, existem schedules que são possíveis sob o protocolo de bloqueio em duas fases, mas não o são sob o protocolo de timestamp, e vice-versa. O protocolo de ordenação por timestamp assegura a serializabilidade por conflito, pois as transações conflitantes são processadas em ordem de timestamp.

O protocolo assegura ausência de impasse (deadlock), pois nenhuma transação sequer espera. Entretanto, é possível acontecer estagnação de transações longas se uma sequência de transações curtas em conflito causar o reinício repetido da transação longa. Se uma transação for descoberta sendo reiniciada repetidamente, as transações em conflito precisam ser temporariamente bloqueadas para permitir que a transação termine. O protocolo pode gerar schedules que não são recuperáveis (not recoverable). Entretanto, ele pode ser estendido para tornar os schedules recuperáveis, em uma entre várias maneiras:

- A recuperação (recoverability) fácil e a ausência de cascata podem ser garantidas realizando todas as escritas juntas no final da transação. As escritas devem ser atômicas no seguinte sentido: enquanto as escritas estão em andamento, nenhuma transação tem permissão para acessar qualquer um dos itens de dados que foram escritos.

8.3 Protocolos baseados em validação

Nos casos em que a maioria das transações são somente leitura, a taxa de conflito de transações pode ser baixa. Assim, muitas dessas transações, se executadas sem serem verificadas pelo sistema de controle concorrente, deixariam o sistema em um estado consistente. O sistema de controle de simultaneidade coloca uma carga adicional na execução do código e gera possível atraso na transação. Pode ser melhor usar um sistema alternativo, que crie menos sobrecarga. Uma das dificuldades para

reduzir os overheads é que não sabemos de antemão quais eventos estarão envolvidos no conflito. Para obter essas informações, precisamos de um diagrama para controlar o sistema.

O protocolo de validação exige que cada transação de Ti seja executada em dois ou três momentos diferentes durante sua vida útil, dependendo se é uma transação somente de leitura ou uma transação de atualização. As etapas estão na seguinte ordem:

1) **Fase de leitura**: durante essa fase, o sistema executa a transação Ti. Ele lê os valores dos diversos itens de dados e os guarda em variáveis locais a Ti. Ele realiza todas as transações write em variáveis locais temporárias, sem atualizações do banco de dados real.

2) **Fase de validação**: o teste de validação é aplicado para a transação Ti. Ele determina se Ti tem permissão para seguir para a fase de escrita sem causar uma violação de serializabilidade. Se a transação falhar no teste de validação, o sistema aborta a transação.

3) **Fase de escrita**: se a transação Ti tiver sucesso na validação, as variáveis temporárias locais que mantêm os resultados de quaisquer operações write realizadas por Ti são armazenadas no banco de dados. As transações somente leitura omitem essa fase.

Cada transação deve passar por três etapas na ordem mostrada. No entanto, fases de transações simultâneas podem ser omitidas. Para fazer um teste de validação, precisamos saber quando ocorreram as diferentes etapas das transações. Portanto, associaremos três timestamps diferentes à transação Ti.

- StartTS(Ti): o momento em que Ti iniciou sua execução.
- ValidationTS(Ti): o momento em que Ti terminou sua fase de leitura e iniciou sua fase de validação.
- FinishTS(Ti): o momento em que Ti terminou sua fase de escrita.

Determinamos a ordem de serializabilidade pela técnica de ordenação por timestamp, usando o valor do timestamp ValidationTS(Ti). Assim, o valor TS(Ti) = ValidationTS(Ti) e, se TS(Tj) < TS(Tk), qualquer schedule produzido terá de ser equivalente a um schedule serial em que a transação Tj aparece antes da transação Tk.

O teste de validação para a transação Ti exige que, para todas as transações Tk com TS(Tk) < TS(Ti), uma das duas condições a seguir seja mantida:

- FinishTS(Tk) < StartTS(Ti). Como Tk completa sua execução antes que Ti inicie, a ordem de serializabilidade é de fato mantida.
- O conjunto de itens de dados escritos por Tk não tem interseção com o conjunto de itens de dados lidos por Ti, e Tk completa sua fase de escrita antes que Ti inicie sua fase de validação (StartTS(Ti) < FinishTS(Tk) < ValidationTS(Ti)). Essa condição garante que as escritas de Tk e Ti não

sejam sobrepostas. Como as escritas de Tk não afetam a leitura de Ti, e como Ti não pode afetar a leitura de Tk, a ordem de serializabilidade é de fato mantida.

Considere novamente as transações T15 e T16. Suponha que TS(T15) < TS(T16). Então, a fase de validação é bem-sucedida no schedule 10 da tabela a seguir. Podemos observar que as escritas para as variáveis em si são realizadas somente após a fase de validação de T16. Assim, T15 lê os valores antigos de C2 e C1, e, dessa forma, esse schedule é serializável.

Tabela 38 – Schedule 10: produzido pelo uso da validação

T15:	T16:
read(C2);	
	read(C2);
	C2 := C2 – 50;
	read(C1);
	C1 := C1 + 50;
read(C1);	
<validar>	
display(C1 + C2)	
	<validar>
	write(C2);
	write(C1);

O esquema de validação automaticamente evita reversões em cascata (cascading rollbacks), já que as escritas em si precisam acontecer somente após a transação que emitiu a escrita ser confirmada. Existe a possibilidade de estagnação (starvation) de transações longas, em face de uma sequência de transações curtas que causam conflito de reinicializações repetidas da transação longa. Para evitar essa estagnação, as transações conflitantes devem ser bloqueadas por um tempo para aguardar a transação longa terminar.

As condições de validação resultam em uma transação T apenas sendo validada contra o conjunto de transações Ti que terminaram depois que T foi iniciada e que são serializadas antes de T. As transações que terminaram antes que T fosse iniciada podem ser ignoradas nos testes de validação. As transações Ti que são serializadas depois de T (ou seja, que têm ValidationTS(Ti) > ValidationTS(T)) também podem ser ignoradas; assim, quando uma transação Ti fosse validada, ela seria validada contra T se T terminasse depois que Ti fosse iniciada.

Esse esquema de validação é chamado de esquema de controle de concorrência otimista, pois as transações são executadas de forma otimista, acreditando que serão capazes de concluir a execução e validar no final. Diferente do bloqueio, a ordenação por timestamp é pessimista, pois força uma espera ou um rollback sempre que um conflito é detectado, mesmo que exista a chance de o schedule ser serializável por conflito.

É possível usar TS(Ti) = StartTS(Ti), em vez de ValidationTS(Ti), sem afetar a serializabilidade. Porém, isso pode resultar em uma transação Ti entrando na fase de validação antes de uma transação Tj que possui TS(Tj) < TS(Ti). Então, a validação de Ti teria de esperar que Tj fosse concluída, de modo que seus conjuntos de leitura e escrita fossem completamente conhecidos. O uso de ValidationTS evita esse problema.

8.4 Granularidade múltipla

Há situações em que seria útil agrupar várias unidades de dados e processá-las como uma única unidade de planejamento. Por exemplo, se a transação Ti precisar de acesso a todo o relacionamento e o protocolo de bloqueio for usado para bloquear uma tupla, Ti deve bloquear todas as tuplas no relacionamento. Obviamente, obter muitos desses bloqueios leva tempo; pior ainda, a tabela de bloqueio pode ficar muito grande e não caber mais na memória. Seria melhor se Ti pudesse fazer uma única solicitação de chave para bloquear todo o relacionamento. Por outro lado, se a transação Tj precisar usar apenas algumas tuplas, ela não precisará bloquear toda a conexão porque a concorrência é perdida.

É necessário um mecanismo para definir diferentes níveis de granularidade no sistema. Isso pode ser feito permitindo-se que os itens de dados tenham tamanhos diferentes e definindo uma hierarquia de precisão de dados em que as precisões menores são aninhadas nas maiores. Essa hierarquia pode ser representada graficamente como uma árvore.

Observação

Note que a árvore descrita aqui é muito diferente da árvore usada pelo protocolo de árvore. Um nó não folha em uma árvore com várias raízes representa informações relacionadas a seus descendentes. Em um protocolo de árvore, cada nó é uma unidade de dados independente.

Considere a árvore da figura a seguir, que tem quatro níveis de nó. O nível superior representa todo o banco de dados. Abaixo estão os nós do tipo região. O banco de dados consiste nessas regiões. Cada região, por sua vez, tem dois nós de tipo de arquivo como filhos. Cada área contém exatamente os arquivos que são seus filhos. Nenhum arquivo reside em mais de uma região. Por fim, cada arquivo possui nós de armazenamento. Como antes, um arquivo consiste exatamente nos registros que são filhos dele, e nenhum registro pode estar em mais de um arquivo.

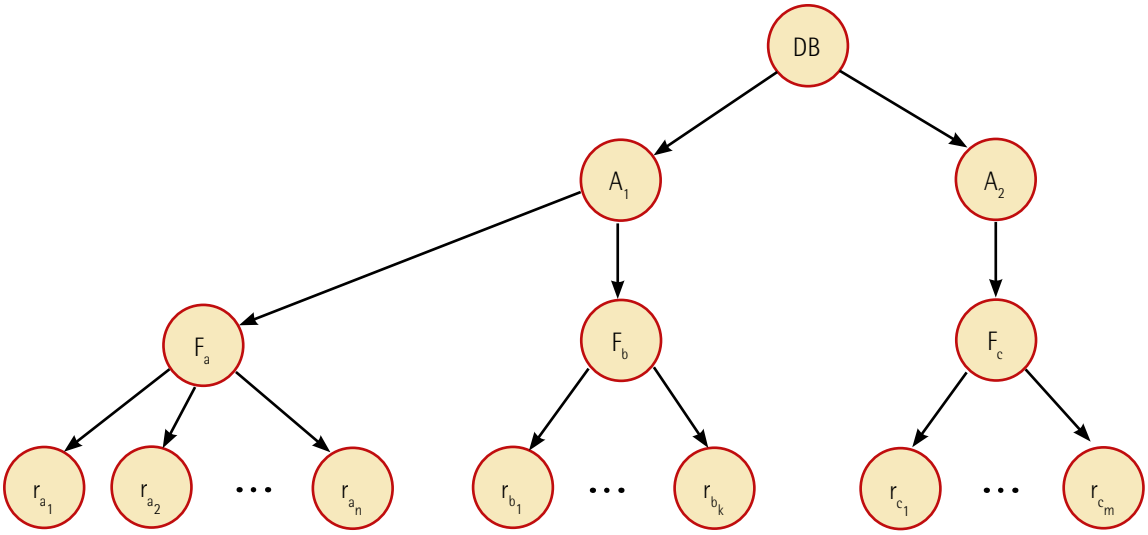

Figura 19 – Hierarquia de granularidade

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 481).

Cada nó na árvore pode ser bloqueado individualmente. Assim como no protocolo de bloqueio de duas fases, usamos modos de bloqueio compartilhados e exclusivos. Quando um evento bloqueia um nó em um estado compartilhado ou exclusivo, o evento bloqueou implicitamente todos os descendentes desse nó no mesmo estado de bloqueio. Por exemplo, se a transação Ti adquirir um bloqueio explícito no arquivo Fc em modo exclusivo na figura, ela terá um bloqueio implícito em todos os registros desse arquivo em modo exclusivo. Ele não precisa bloquear um por um os registros Fc individuais.

Suponha que o evento Tj queira bloquear (bloquear) o registro rb6 do arquivo Fb. Como Fb foi explicitamente inibido por Ti, segue-se que rb6 também é (indiretamente) inibido. No entanto, quando Tj envia uma solicitação de bloqueio para rb6, a rb6 não é bloqueada diretamente. Como o sistema determina se Tj pode bloquear rb6? Ele deve passar da raiz da árvore para o registrador rb6. Se algum nó nesse caminho estiver preso em um estado incompatível, Tj deve ser atrasado.

Agora suponha que o evento Tk queira bloquear todo o banco de dados. Para fazer isso, basta bloquear na raiz da hierarquia. Mas observe que Tk não deve bloquear o nó raiz, porque Ti atualmente mantém um bloqueio em parte da árvore (especificamente o arquivo Fb). Mas como o sistema determina se o nó raiz pode ser bloqueado? Uma possibilidade é que ele pesquise a árvore inteira. No entanto, essa solução vai contra o objetivo de eficiência de um sistema de travamento multi-pellets. Uma maneira mais eficiente de obter essas informações é introduzir uma nova classe de modos de bloqueio, chamadas modos de bloqueio intencionais. Se um nó é bloqueado em modo deliberado, um bloqueio explícito é executado em um nível inferior na árvore (ou seja, com maior precisão). Os bloqueios de destino são colocados em todos os ancestrais de um nó antes de bloqueá-lo. Portanto, uma transação não precisa pesquisar toda a árvore para determinar se pode bloquear um nó com êxito. Um evento que deseja bloquear um nó — digamos, Q — deve seguir um caminho na árvore da raiz até Q. À medida que atravessa a árvore, o evento bloqueia os vários nós no estado de intenção.

O estado compartilhado está relacionado ao estado de destino, e o outro é o estado exclusivo. Se um nó estiver bloqueado no modo Intention-Shared (IS), um bloqueio explícito será executado em um nível inferior na árvore, mas apenas com bloqueios de modo compartilhado. Da mesma forma, se um nó for bloqueado devido a uma intenção exclusiva (Intention-eXclusive – IX), o bloqueio explícito ocorre em um nível inferior e os bloqueios estão em modo exclusivo ou compartilhado. Finalmente, quando um nó é bloqueado no modo Shared and Intention-eXclusive (SES), a subárvore enraizada desse nó é explicitamente bloqueada no modo compartilhado e esse bloqueio explícito ocorre em um nível superior, um espaço exclusivo. A matriz de compatibilidade desses modos de bloco é mostrada na tabela a seguir.

O protocolo de bloqueio de granularidade múltipla usa esses modos de bloqueio para assegurar a serializabilidade. Ele requer que uma transação Ti que tente bloquear um nó Q siga essas regras:

- a transação Ti precisa observar a função de compatibilidade de bloqueio da tabela;
- a transação Ti precisa, primeiro, bloquear a raiz da árvore e pode bloqueá-la em qualquer modo;
- a transação Ti pode bloquear um nó Q no modo S ou IS somente se Ti atualmente tiver o pai de Q bloqueado no modo IX ou IS;
- a transação Ti pode bloquear um nó Q no modo X, SIX ou IX somente se Ti atualmente tiver o pai de Q bloqueado no modo IX ou SIX;
- a transação Ti só pode bloquear um nó se não tiver desbloqueado anteriormente qualquer nó (ou seja, Ti é em duas fases);
- a transação Ti pode desbloquear um nó Q somente se Ti atualmente não tiver nenhum dos filhos de Q bloqueado.

Tabela 39 – Matriz de compatibilidade

	IS	IX	S	SIX	X
IS	Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro	Falso
IX	Verdadeiro	Verdadeiro	Falso	Falso	Falso
S	Verdadeiro	Falso	Verdadeiro	Falso	Falso
SIX	Verdadeiro	Falso	Falso	Falso	Falso
X	Falso	Falso	Falso	Falso	Falso

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 482).

Observe que o protocolo de granularidade múltipla necessita que os bloqueios sejam adquiridos na ordem de cima para baixo (raiz-para-folha), enquanto os bloqueios precisam ser liberados na ordem de

baixo para cima (folha-para-raiz). O impasse é possível no protocolo de granularidade múltipla, assim como no protocolo de bloqueio em duas fases.

Esse protocolo melhora a concorrência e reduz a sobrecarga de bloqueio. Ele é particularmente útil em aplicações que incluem uma mistura de:

- transações curtas que acessam apenas alguns itens de dados;
- transações longas que produzem relatórios de um arquivo inteiro ou conjunto de arquivos.

O número de bloqueios que uma consulta SQL pode precisar adquirir geralmente pode ser estimado a partir das operações de controle relacional da consulta. Por exemplo, uma varredura relacional receberia um bloqueio de nível relacional, enquanto uma varredura de índice que recupera apenas alguns registros pode receber um erro de intenção de nível relacional e bloqueios multiníveis regulares. Se uma transação adquirir muitos bloqueios de cartão, a tabela de bloqueios pode ficar muito cheia. Para lidar com essa situação, o gerenciador de bloqueios pode executar o dimensionamento de bloqueios substituindo muitos bloqueios de nível inferior por um bloqueio de nível superior. Em nosso exemplo, um único bloqueio relacional pode substituir um grande número de vários bloqueios.

Resumo

Vimos, nesta unidade, a importância do gerenciamento de transações. Os primeiros sistemas de informação utilizavam arquivos sequenciais indexados ou arquivos de acesso direto para guardar e atualizar seus dados. Esses mecanismos de armazenamento e consulta de dado eram limitados e não possuíam recurso de gerenciamento como encontramos atualmente nos sistemas de bancos de dados.

Toda transação deve respeitar as quatro propriedades que garantem o correto funcionamento do banco de dados e impede que os dados sejam perdidos ou corrompidos no processamento. Essas propriedades são atomicidade, consistência, isolamento e durabilidade (ACID).

Uma das principais características de uma transação é o isolamento. Se várias transações forem executadas ao mesmo tempo em um banco de dados, o atributo de isolamento será comprometido. Para manter a integridade dos dados, o sistema deve monitorar as interações entre transações concorrentes. Esse controle é obtido por meio de diversos mecanismos chamados sistemas de controle simultâneos.

Exercícios

Questão 1. (CS-UFG 2017, adaptada) Uma transação é uma unidade de execução de programa que acessa e pode atualizar vários itens de dados em um sistema gerenciador de bancos de dados (SGBD). Uma transação envolve tipicamente a execução de código escrito em SQL delimitado por declarações de início e de fim de transação (*begin transaction e end transaction*). Qual das seguintes propriedades deve ser assegurada por um SGBD no processamento de transações?

A) Propriedade de atomicidade, que garante que a execução de uma transação seja feita sem outra transação em execução simultânea.

B) Propriedade de consistência, que garante que, na execução concorrente de transações, cada transação possa ser executa sem ser afetada por outras transações em execução simultânea no sistema.

C) Propriedade de durabilidade, que garante que, após uma transação ser concluída com êxito, as alterações feitas no banco de dados persistem, mesmo se houver falhas do sistema.

D) Propriedade de isolamento, que garante que, ou todas as operações da transação são refletidas corretamente no banco de dados, ou nenhuma delas o é.

E) Propriedade de variabilidade, que garante que a transação possa ser executada por diversas sequências de instruções distintas.

Resposta correta: alternativa C.

Análise das alternativas

A) Alternativa incorreta.

Justificativa: uma transação é uma unidade lógica de operações de banco de dados. Toda transação deve respeitar as quatro propriedades que garantem o correto funcionamento do banco de dados: atomicidade, consistência, isolamento e durabilidade. A propriedade de atomicidade diz que cada transação pode ser fragmentada, ou seja, que todas as operações sejam executadas como se fossem uma única operação. Desse modo, se qualquer operação de banco de dados falhar, toda a transação deverá ser desfeita.

B) Alternativa incorreta.

Justificativa: a propriedade de consistência diz que cada transação deve deixar o banco de dados em um estado consistente após sua execução. Desse modo, a transação deve atender a todas as regras e restrições exigidas pelo banco de dados.

C) Alternativa correta.

Justificativa: a propriedade de durabilidade diz que cada transação deverá ter resultados permanentes no banco de dados, e só poderá ser desfeita na próxima transação. Assim, após uma transação ser concluída com êxito, as alterações feitas no banco de dados persistem.

D) Alternativa incorreta.

Justificativa: a propriedade de isolamento diz que cada transação deve ser isolada de outras transações no banco de dados. Desse modo, os resultados parciais de cada transação não devem ser enxergados por outras transações.

E) Alternativa incorreta.

Justificativa: não existe a propriedade de variabilidade, no contexto de transações em bancos de dados.

Questão 2. (Instituto AOCP/2020, adaptada) Um SGBD que não tem um controle de concorrência efetivo pode apresentar problemas na integridade de seus dados. Suponha que uma transação T1 atualiza determinado registro de uma tabela e, nesse meio tempo, outra transação T2 utiliza esse mesmo registro para suas operações. Contudo, a transação T1 falha e é desfeita pelo SGBD. Esse problema é conhecido como:

A) Leitura fantasma.

B) Resumo incorreto.

C) Deadlock.

D) Atualização perdida.

E) Leitura suja.

Resposta correta: alternativa E.

Análise da questão

No contexto de SGDB's, uma leitura suja (*dirty read*) ocorre quando uma transação lê dados que ainda não foram confirmados. Suponha que uma transação T1 atualize uma linha e uma transação T2 leia a linha atualizada antes que a transação 1 confirme a atualização. Se a transação 1 reverter a alteração devido a alguma falha, a transação T2 terá dados de leitura incorretos. Esse processo falho de leitura, conhecido como leitura suja, corresponde à situação descrita no enunciado da questão. Essa situação evidencia um baixo nível de isolamento do sistema de bancos de dados.

REFERÊNCIAS

ALVES, W. P. *Banco de dados*. São Paulo: Saraiva, 2014.

CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, Nova York, v. 13, n. 6, p. 377-38, 1970. Disponível em: https://bit.ly/3D40K0N. Acesso em: 18 out. 2022.

DATE, C. J. *Introdução a sistemas de bancos de dados*. Rio de Janeiro: Elsevier, 2003.

Dobrushkin, V. A. *Métodos para análise de algoritmos*. Rio de Janeiro: LTC, 2012.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Pearson, 2011.

FAGIN, R. *et al.* Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, Nova York, v. 4, n. 3, p. 315-344, 1979. Disponível em: https://bit.ly/3yPSAGG. Acesso em: 18 out. 2022.

LITWIN, W. Linear hashing: a new tool for file and table addressing. *In:* INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 6., 1980, Montreal. *Proceedings* [...]. [*S.l.*]: VLDB, 1980. p. 212-223.

LITWIN, W. Virtual hashing: a dynamically changing hashing. *In:* INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 4., 1978, Berlim. *Proceedings* [...]. [*S.l.*]: VLDB, 1978. p. 517-523.

LOUDEN, K. C. *Compiladores*: princípios e práticas. São Paulo: Cengage Learning, 2004.

O'NEIL, P.; QUASS, D. Improved query performance with variant indexes. *In:* SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1997, Tucson. *Proceedings* [...]. Nova York: ACM, 1997. p. 38-49.

Ramakrishnan, R.; GEHRKE, J. *Sistemas de gerenciamento de bancos de dados*. 3. ed. Nova York: McGraw-Hill, 2008.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Sistema de banco de dados*. Rio de Janeiro: Elsevier, 2020.

TANENBAUM, A. S.; WETHERALL, D. J. *Redes de computadores*. 5. ed. Rio de Janeiro: Pearson Prentice Hall, 2011.

Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000