FACULDADE DE ADMINISTRAÇÃO E NEGÓCIOS DE SERGIPE

NÚCLEO DE PÓS-GRADUAÇÃO E EXTENSÃO – NPGE

PÓS-GRADUAÇÃO MBA EM

ADMINISTRAÇÃO DE BANCO DE DADOS

ISRAEL ANTONIO PINTO SILVA

CONTROLE DE CONCORRÊNCIA EM SISTEMAS DE BANCO DE DADOS RELACIONAIS

ARACAJU

2016/01

ISRAEL ANTONIO PINTO SILVA

CONTROLE DE CONCORRÊNCIA EM SISTEMAS DE BANCO DE DADOS RELACIONAIS

Artigo científico apresentado como requisito para obtenção do título de Especialista em Administração de Banco de Dados.

ARACAJU

2016/01

CONTROLE DE CONCORRÊNCIA EM SISTEMAS DE BANCO DE DADOS RELACIONAIS

Israel Antônio Pinto Silva[1]

RESUMO

Nos sistemas de banco de dados, as transações precisam atender a quatro propriedades: atomicidade, consistência, isolamento e durabilidade. A propriedade de isolamento visa garantir que uma transação execute sem interferências de outras transações. Porém, como as transações são executadas de modo simultâneo, é possível que duas transações executando concorrentemente acessem o mesmo item de dados. Nesse caso, uma pode interferir no valor acessado pela outra. Por causa disso, os sistemas de banco de dados dedicam um módulo específico para controlar esse acesso concorrente, conhecido como sistema de controle de concorrência. Este trabalho qualitativo e bibliográfico tem como objetivo apresentar conceitos relacionados ao controle de concorrência de transações e dois importantes algoritmos desenvolvidos para realizar esse controle. Também será visto o que a *Structured Query Language* (SQL), que é a linguagem de programação padrão para banco de dados relacionais, apresenta sobre o assunto e como dois dos principais sistemas de banco de dados comerciais controlam a concorrência de suas transações.

Palavras-chave: Sistemas de banco de dados. Controle de concorrência. SQL.

1 INTRODUÇÃO

Nos sistemas de banco de dados, uma transação é uma coleção de operações sobre os dados que forma uma única unidade lógica de trabalho (SILBERSCHATZ; KORTH; SUDARSHAN, 2012). Em sistemas bancários, por exemplo, uma transação comum é a transferência de dinheiro entre duas contas. Para o usuário, esse procedimento aparenta ser uma única operação, mas internamente no banco de dados há no mínimo duas operações: o débito na conta de origem e o crédito na conta de destino.

[1] Bacharel em Ciência da Computação pela Universidade Federal de Sergipe (2013). Desde 2014 ocupa o cargo de Analista de Controle Externo II no Tribunal de Contas do Estado de Sergipe. Atualmente é aluno do curso de pós-graduação MBA em Administração de Banco de Dados na Faculdade de Administração e Negócios de Sergipe – FANESE. Email: israelantonio90@gmail.com.

Além disso, a transação deve ser uma única unidade lógica, ou seja, todas as suas operações devem ser executadas ou, em caso de falha, nenhuma será executada. Dessa forma, não pode haver execução parcial da transação, como debitar a conta de origem, mas não creditar a conta de destino, no caso da transferência bancária.

Essa propriedade da transação ser executada como única unidade lógica é conhecida como atomicidade. Além dela, existem outras três propriedades que caracterizam uma transação no banco de dados: consistência, isolamento e durabilidade. Essas quatro propriedades foram apresentadas por Haerder e Reuter (1983) com o acrônimo de ACID, que é derivado da letra inicial de cada propriedade. Atualmente, essa sigla é conhecida mundialmente na área de Banco de Dados para definir uma transação.

Segundo a propriedade consistência, o sistema de banco de dados deve manter a consistência dos dados após executar a transação. Isso significa que o banco de dados deve refletir de maneira correta os dados do mundo real. Na transferência bancária, uma forma de consistência estaria relacionada à soma dos saldos das duas contas, ou seja, o valor total do saldo da conta de origem somado ao saldo da conta de destino deve ser o mesmo antes e depois da transação. Se isso não acontecer, então a transação poderia ter criado ou destruído dinheiro, e o banco de dados deixaria de refletir corretamente o mundo real.

A propriedade de isolamento visa garantir que uma transação execute sem interferências de outras transações. Se não existir esse controle, uma transação poderá ter acesso aos dados modificados por outra transação que ainda não terminou, o que pode levar o banco de dados a um estado inconsistente.

Já a propriedade de durabilidade garante que todas as alterações realizadas pela transação sejam armazenadas no banco de dados, desde que a transação tenha completado com sucesso.

Se todas as transações em um sistema de banco de dados atendem às propriedades ACID, então a consistência dos dados é garantida. No entanto, atender a todas elas requer muitos recursos do sistema de banco de dados e até mesmo dos desenvolvedores da aplicação.

No caso do isolamento, há um componente específico do banco de dados conhecido como sistema de controle de concorrência, que faz uso de um algoritmo próprio para determinar como as transações serão executadas concorrentemente. Nas próximas seções, serão vistos dois importantes algoritmos de controle de concorrência, o que o padrão SQL apresenta sobre assunto e como alguns dos principais sistemas de banco de dados comerciais controlam a concorrência de suas transações. Antes disso, é necessário entender como as transações

interagem com o sistema do banco de dados e como é possível garantir que elas atendam às propriedades ACID.

2 MODELO DE TRANSAÇÃO

Em SQL (ISO, 2011), que é a linguagem de programação padrão para bancos de dados relacionais, há duas instruções relacionadas ao processamento de transação: *commit* e *rollback* (ou ***abort***). A primeira indica que a transação terminou com sucesso e, portanto, as alterações realizadas devem ser armazenadas no banco de dados. Já a segunda é utilizada para indicar que a transação foi encerrada sem sucesso e todas as alterações realizadas devem ser desfeitas.

Alguns sistemas de banco de dados também utilizam instruções para delimitar o início e fim da transação, como as instruções *begin transaction* e *end transaction* utilizadas pelo SQL Server. Todas as operações entre estas duas instruções fazem parte de uma mesma transação.

Através da linguagem SQL é possível realizar operações bastante complexas nos dados. Por isso, não seria didático apresentar os conceitos de processamento de transação utilizando essa linguagem. Será utilizado um modelo de transação simples, que é abordado por vários autores em suas obras, dentre eles Silberschatz, Korth e Sudarshan (2012) e Navathe e Elmasri (2011). Nesse modelo, cada item de dados contém um único valor do tipo inteiro e é identificado por um nome (por exemplo, o item de dados A contém o valor 20). Além disso, são permitidas apenas duas operações sobre os dados:

- *read*(X): lê o item de dados chamado X e armazena seu valor numa variável de mesmo nome. Essa variável pertence a um buffer da memória principal e é acessada apenas pela transação que executou a operação;
- *write*(X): grava o valor da variável X no item de dados de mesmo nome. Essa gravação não é realizada no disco no momento em que a operação *write* é executada. Na verdade, a gravação é feita no buffer e o sistema de banco de dados escolherá o momento oportuno para realizar a gravação no disco. Para simplificar, será considerado que o item de dados do disco foi modificado quando o comando *write* for executado.

De acordo com as operações descritas, uma transação T_1 que transfere R$ 10,00 da conta A para a conta B poderia ser representada como:

Tabela 01

Transação de transferência entre duas contas

T_1
read(A);
A := A – 10;
write(A);
read(B);
B := B + 10;
write(B);

Fonte: Autoria Própria.

Data: 05/2016.

Essa transação é composta de seis passos, que são descritos a seguir:

1. Leia o item de dados A do banco de dados e armazene seu valor na variável de memória de mesmo nome;
2. Subtraia o valor 10 da variável A presente na memória;
3. Grave o valor da variável A no item de dados de mesmo nome;
4. Leia o item de dados B do banco de dados e armazene seu valor na variável de memória de mesmo nome;
5. Adicione o valor 10 à variável B presente na memória;
6. Grave o valor da variável B no item de dados de mesmo nome.

Para que essa transação apresente a propriedade de consistência, é necessário que o banco de dados reflita corretamente os dados do mundo real após a execução da transação. Uma forma de consistência para a transação de transferência bancária seria garantir que a soma do valor das duas contas permaneceria o mesmo após a execução da transação.

Se na transação T_1 o valor inicial de A fosse R$ 100,00 e de B fosse R$ 300,00, que resulta numa soma total de R$ 400,00, então o valor de A após a transação seria R$ 90,00 e o de B seria R$ 310,00. Dessa forma, a soma total permaneceria a mesma. Caso isso não fosse garantido, então a transação poderia criar (se a soma final fosse maior que a inicial) ou destruir dinheiro (se a soma final fosse menor). Vale ressaltar que garantir essa propriedade para uma transação individual é responsabilidade do programador da aplicação, já que esse tipo de restrição é difícil de ser verificado automaticamente pelo banco de dados.

Para garantir a atomicidade, o sistema de banco de dados deve executar a transação como uma única unidade lógica, ou seja, todas as suas operações devem ser executadas ou, em

caso de falha, nenhuma será executada. Se ocorresse uma falha após a execução do passo 3 da transação T_1, por exemplo, então o banco de dados deveria reverter todas as operações que realizaram alterações no banco de dados até esse ponto (nesse caso, apenas a operação *write*(A)). Se isso não fosse feito, então se teria destruído R$ 10,00 e o banco de dados estaria num estado inconsistente.

Segundo a propriedade durabilidade, todas as alterações realizadas pela transação devem ser armazenadas no banco de dados, desde que a transação tenha completado com sucesso. Dessa forma, o sistema de banco de dados deve garantir que os dados serão gravados se a transação for finalizada com sucesso, mesmo que aconteça alguma falha entre o término da transação e a gravação dos dados. Garantir a atomicidade e a durabilidade é de responsabilidade do sistema de banco de dados, que possui um módulo específico para essa tarefa, conhecido como sistema de recuperação.

Devido a sua importância para o controle de concorrência, a propriedade de isolamento será vista em detalhes na Seção 3, que também aborda os principais problemas que podem acontecer quando transações são executadas concorrentemente.

3 ISOLAMENTO DE TRANSAÇÕES

A propriedade de isolamento visa garantir que uma transação execute sem interferências de outras transações. Se não existir esse controle, uma transação poderá ter acesso aos dados modificados por outra transação que ainda não terminou, o que pode levar o banco de dados a um estado inconsistente. Para exemplificar isso, considere a transação T_2 descrita na Tabela 02. Ela realiza um saque no valor de R$ 50,00 da conta A.

Tabela 02

Transação que realiza um saque da conta A

T_2
read(A);
A := A – 50;
write(A);

Fonte: Autoria Própria.

Data: 05/2016.

Em sistemas de banco de dados que executam transações de forma concorrente, é possível que as operações das transações sejam intercaladas de uma maneira que resulte num estado inconsistente. Se as transações T1 e T2 forem executadas concorrentemente, suas operações poderiam ser intercaladas da seguinte forma:

Tabela 03

Possível execução concorrente entre as transações T_1 e T_2

T_1	T_2
read(A);	
A := A – 10;	
	read(A);
	A := A – 50;
	write(A);
write(A);	
read(B);	
B := B + 10;	
write(B);	

Fonte: Autoria Própria.

Data: 05/2016.

Ao executar as operações na ordem apresentada na Tabela 03, a alteração realizada pela transação T_2 é perdida. Isso ocorre porque T_1 lê o valor de A antes de T_2, mas escreve esse valor após T_2 ter escrito. Dessa forma, o valor escrito por T_1 não levou em consideração a atualização feita por T_2. Esse problema é conhecido como atualização perdida. O padrão SQL (ISO, 2011) descreve ainda três outros problemas que podem acontecer se as transações forem executadas concorrentemente: a leitura suja, a leitura não repetitiva e o fenômeno fantasma.

A leitura suja ocorre quando uma transação lê um valor que foi modificado por outra transação que ainda não terminou. Na execução concorrente da Tabela 04, T_2 lê o valor de A atualizado por T_3, que ainda não terminou. Então, T_2 altera o valor de A e, em seguida, T_3 é encerrada com um comando *abort*. Assim, todas as atualizações feitas por T_3 seriam desfeitas, mas T_2 já considerou o valor modificado em suas operações.

Tabela 04

Problema da leitura suja

T_2	T_3
	read(A);
	A := A + 100;
	write(A);
read(A);	
A := A – 50;	
write(A);	
	abort;

Fonte: Autoria Própria.

Data: 05/2016.

Na leitura não repetitiva, uma transação lê o valor de um item duas vezes e, entre essas duas leituras, outra transação modifica o valor desse mesmo item. Dessa forma, a primeira transação lê valores diferentes para o mesmo item de dados.

Já o fenômeno fantasma acontece quando uma transação lê, em dois momentos distintos, um conjunto de itens que atendem determinadas condições e, entre essas leituras, outra transação insere um item de dados que também atende a essas condições. Portanto, esse novo item (considerado o fantasma) aparecerá na segunda leitura, mas não estará presente na primeira.

Todos esses problemas podem ocorrer em sistemas que permitem a execução concorrente, por causa da possível intercalação entre as operações das transações. Uma solução seria proibir a execução simultânea, ou seja, no máximo uma transação estaria em execução em determinado momento.

No entanto, isso degradaria bastante o desempenho do sistema, por dois motivos: se uma transação em execução aguardasse operações de disco, o processador ficaria ocioso, já que ele não poderia atender nenhuma outra transação; e a execução de uma transação de longa duração prejudicaria todas as demais transações, porque teriam que esperar o término dela. Por isso, a execução concorrente é necessária, mas deve existir um controle que impeça a ocorrência desses problemas. Isso será visto na Seção 4, através do conceito de *schedules*.

4 SCHEDULES

Um *schedule* é uma sequência de execução das operações de um conjunto de transações. Na Tabela 03, há um *schedule* envolvendo as transações T_1 e T_2. Como visto, esse schedule deixa o banco de dados num estado inconsistente, porque o valor atualizado por T_2 é perdido. Na Tabela 05, há outro *schedule* envolvendo as transações T_1 e T_2, mas sem intercalar operações de diferentes transações. Esse tipo de *schedule* é conhecido como serial (porque as transações estão em série) e representa uma execução não concorrente de transações.

Tabela 05

Schedule S_1 envolvendo as transações T_1 e T_2

T_1	T_2
read(A);	
A := A – 10;	
write(A);	
read(B);	
B := B + 10;	
write(B);	
	read(A);
	A := A – 50;
	write(A);

Fonte: Autoria Própria.

Data: 05/2016.

Conforme visto na Seção 3, não é aceitável eliminar a concorrência, porque isso piora o desempenho do sistema. Dessa forma, *schedules* seriais não são utilizados na prática. Porém, a partir deles se descobriu uma solução para os problemas de execução concorrente. A solução seria utilizar *schedules* de execução simultânea que fossem equivalentes a *schedules* seriais. Esse novo tipo de *schedule* ficou conhecido como serializável (GRAY; REUTER, 1993).

O método mais conhecido para determinar se um *schedule* é serializável, é chamado de serializabilidade por conflito (GRAY; REUTER, 1993). Neste método, duas operações de um *schedule* são classificadas como conflitantes se atendem a três condições: pertencem a transações diferentes; acessam o mesmo item de dados; e pelo menos uma delas é uma operação *write*. Na Tabela 05, por exemplo, a operação *read*(A) de T_2 é conflitante com *write*(A) de T_1,

mas não é conflitante com *read*(A) de T_1 (porque nenhuma delas é uma operação *write*), nem com operação *write*(B) de T_1 (porque acessam itens de dados diferentes).

Quando duas operações são conflitantes, inverter a ordem de execução delas pode ocasionar resultados diferentes. Na Tabela 06, as operações *write*(A) das transações T_1 e T_2 foram invertidas em relação ao schedule da Tabela 03**Erro! Fonte de referência não encontrada.**. Nesse caso, o resultado para o item de dados A é diferente do resultado obtido anteriormente: enquanto na Tabela 03 T_1 sobrescreve o valor armazenado por T_2, na Tabela 06 T_2 sobrescreve o valor armazenado por T_1.

Tabela 06

Schedule S_2 que contém as transações T_1 e T_2

T_1	T_2
read(A);	
A := A – 10;	
	read(A);
	A := A – 50;
write(A);	
	write(A);
read(B);	
B := B + 10;	
write(B);	

Fonte: Autoria Própria.

Data: 05/2016.

No método de serializabilidade por conflito, o objetivo é encontrar *schedules* serializáveis, ou seja, que sejam equivalentes a *schedules* seriais. Para isso, a partir de um *schedule* serial, são feitas trocas de ordem entre operações não conflitantes. Como trocas entre operações não conflitantes não alteram o resultado final, então o *schedule* resultante dessas trocas será equivalente ao *schedule* serial inicial e, portanto, será serializável (ou serializável por conflito, devido ao método utilizado). Dito de outra forma, um *schedule* é serializável por conflito se a ordem de duas operações em conflito for a mesma presente no *schedule* serial.

Considere o schedule S_3 da Tabela 07, que contém as transações T_1 e T_2. Ele foi obtido a partir do schedule serial S_1 da Tabela 03, através da troca entre a operação *read*(A) de T_2 com as operações *write*(B) e *read*(B) de T_1. Como essas operações não são conflitantes, então S_3 é

equivalente a S_1 e, consequentemente, é serializável. Já o schedule S_4 presente na Tabela 08 não é serializável, porque as operações *write*(A) de T_1 e *read*(A) de T_2 (que são conflitantes) não estão na mesma ordem do schedule serial S_1.

Tabela 07

Schedule serializável S_3

T_1	T_2
read(A);	
A := A – 10;	
write(A);	
	read(A);
read(B);	
B := B + 10;	
write(B);	
	A := A – 50;
	write(A);

Fonte: Autoria Própria.

Data: 05/2016.

Tabela 08

Schedule não serializável S_4

T_1	T_2
read(A);	
A := A – 10;	
	read(A);
write(A);	
read(B);	
B := B + 10;	
write(B);	
	A := A – 50;
	write(A);

Fonte: Autoria Própria.

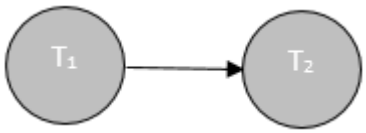
Data: 05/2016.

Navathe e Elmasri (2011) apresentam um algoritmo simples para determinar se um *schedule* é serializável por conflito. Esse algoritmo utiliza um grafo de precedência para representar o *schedule*. Esse grafo é formado por um conjunto V de vértices, cada um representando uma transação, e outro conjunto E de arestas direcionadas. As arestas têm a forma $T_i \rightarrow T_j$, onde T_i e T_j são vértices, T_i é o nó inicial e T_j é o nó final. Haverá uma aresta $T_i \rightarrow T_j$ se, e somente se, existir alguma operação na transação T_i que apareça antes de alguma outra operação na transação T_j e ambas forem conflitantes entre si. O *schedule* será serializável por conflito se não existirem ciclos no grafo de precedência obtido.

No schedule S3 da Tabela 07, *write*(A) de T_1 aparece antes de *read*(A) de T_2. Como essas operações são conflitantes e a operação de T_1 aparece antes de T_2, então existe uma aresta $T_1 \rightarrow T_2$ no grafo de precedência de S_3. A operação *read*(A) de T_2 aparece antes de *read*(B) e *write*(B) de T_1, mas não é conflitante com nenhuma delas. Por isso, não existe uma aresta de T_2 para T_1. Assim, o grafo de precedência de S_3 possui o conjunto de vértices $V = \{T_1, T_2\}$ e o conjunto de arestas $E = \{T_1 \rightarrow T_2\}$. Esse grafo é representado na Figura 01. Como ele não possui ciclos, então o *schedule* S_3 é serializável.

Figura 01

Grafo de precedência de S_3



Fonte: Autoria Própria.

Data: 05/2016.

No *schedule* S_4 da Tabela 08, *read*(A) de T_2 aparece antes de *write*(A) de T_1. Como são operações conflitantes, então existe uma aresta $T_2 \rightarrow T_1$ no grafo de precedência de S_4. Além disso, *write*(A) de T_1 aparece antes de *write*(A) de T_2 e, como as duas operações são conflitantes entre si, há também a aresta $T_1 \rightarrow T_2$ no grafo correspondente. Dessa forma, o grafo de precedência de S4 possui o conjunto de vértices $V = \{T_1, T_2\}$ e o conjunto de arestas $E = \{T_1 \rightarrow T_2, T_2 \rightarrow T_1\}$. Como existe um ciclo, então S_4 não é serializável. Isso pode ser visto na Figura 02, que representa o grafo de precedência de S_4.

Figura 02

Grafo de precedência de S_4

Fonte: Autoria Própria.

Data: 05/2016.

5 ALGORITMOS

O método de serialização por conflito é utilizado apenas didaticamente para facilitar o entendimento dos conceitos de processamento de transação. Em sistemas de banco de dados reais, esse método é ineficiente porque requer que o *schedule* contendo as transações já esteja pronto para ser processado. Nesse momento, verifica-se a serializabilidade do *schedule*: se for serializável, ele prossegue para o processamento; senão, ele é cancelado. Dessa forma, o sistema gasta recursos para montar o *schedule* e para verificar sua serializabilidade. O ideal seria impedir que *schedules* não serializáveis fossem criados, o que levou ao desenvolvimento de algoritmos específicos para realizar esse controle.

Entre os mais importantes estão o bloqueio de duas fases e o baseado em *snapshot*, que serão vistos nessa seção. Esses algoritmos foram aperfeiçoados ao longo dos anos, mas serão apresentadas suas versões iniciais, que até hoje servem de base para os sistemas de controle de concorrência.

5.1 Algoritmos Baseados Em Bloqueio

Nos algoritmos baseados em bloqueio, cada transação precisa solicitar o bloqueio dos itens de dados que deseja acessar. Se o bloqueio for concedido, a transação pode continuar com suas operações; senão, a transação deve esperar a concessão do bloqueio ou ser revertida. Além disso, a transação deve liberar os bloqueios que obteve antes de finalizar. Existem diversos modos de bloqueio que podem ser feitos sobre um item de dados, mas serão considerados apenas dois: o bloqueio compartilhado e o bloqueio exclusivo.

No bloqueio compartilhado, a transação pode ler o item de dados, mas não pode modificá-lo. Já no modo exclusivo, a transação pode ler e modificar o item de dados. Quando uma transação solicita um bloqueio compartilhado sobre um item de dados A, o bloqueio é concedido se não existirem bloqueios exclusivos sobre A. Já para um bloqueio exclusivo ser concedido, nenhum bloqueio pode existir sobre o item, nem mesmo o compartilhado. Dessa forma, é possível que diferentes transações mantenham bloqueio compartilhado sobre o mesmo item de dados, mas no máximo uma transação pode manter um bloqueio exclusivo sobre um item de dados.

Para representar o funcionamento dos algoritmos de bloqueio, também será utilizado o modelo de transação apresentado na Seção 2. Porém, serão adicionadas mais três operações a esse modelo:

1. lock-S(A): solicita bloqueio compartilhado sobre o item de dados A;
2. lock-X(A): solicita bloqueio exclusivo sobre o item de dados A;
3. unlock(A): libera todos os bloqueios que a transação tem sobre o item de dados A.

Na Tabela 09, a transação T_1 é representada com a utilização das operações de bloqueio. Antes de acessar cada item de dado, a transação solicita um bloqueio através da operação *lock*. Como T_1 realiza escritas em A e B, então o bloqueio é exclusivo. Quando T_1 termina suas operações sobre os dados, ela os desbloqueia com a operação *unlock*.

Tabela 09

Transação T_1 com bloqueios

T_1
lock-X(A);
read(A);
A := A – 10;
write(A);
unlock(A);
lock-X(B);
read(B);
B := B + 10;
write(B);
unlock(B);

Fonte: Autoria Própria.

Data: 05/2016.

O sistema de controle de concorrência é responsável pelo gerenciamento dos bloqueios, mas não foi representado para simplificar o modelo. Quando a transação T_1 da Tabela 09 executa uma operação *lock*, esse sistema verifica se é possível conceder o bloqueio. Se for possível, o bloqueio é concedido; senão, a transação é revertida ou aguarda a concessão no momento futuro. Nos exemplos desta seção, considera-se que o bloqueio foi concedido, exceto quando for dito o contrário.

A utilização de bloqueios não é suficiente para garantir a serializabilidade do *schedule*. Considere a transação T_4, que lê os valores das contas A e B, e o *schedule* S_5 da Tabela 10. Comparando S_5 com o *schedule* serial que contém T_1 seguida por T_4, as operações *read*(B) de T_4 e *write*(B) de T_1 (operações conflitantes entre si) foram invertidas em S_5. Portanto, de acordo com o método de serializabilidade por conflito, S_5 não é serial.

Tabela 10

Schedule S5 não serializável

T_1	T_4
lock-X(A);	
read(A);	
A := A – 10;	
write(A);	
unlock(A);	
	lock-S(A);
	read(A);
	unlock(A);
	lock-S(B);
	read(B);
	unlock(B);
lock-X(B);	
read(B);	
B := B + 10;	
write(B);	
unlock(B);	

Fonte: Autoria Própria.

Data: 05/2016.

O *schedule* S5 não é serializável porque ele libera o item de dados A num momento inadequado. O ideal seria manter o bloqueio de A até o final da transação T_1. A partir dessa observação, foi desenvolvido o algoritmo de bloqueio em duas fases (ESWARAN *et al*, 1976), que garante a serializabilidade dos *schedules*. Nesse algoritmo, há duas fases nas quais as transações podem solicitar bloqueios e desbloqueios:

1. Fase de crescimento: a transação pode solicitar bloqueios, mas não pode liberar bloqueios;
2. Fase de encolhimento: a transação pode liberar bloqueios, mas não pode solicitar novos bloqueios.

Quando uma transação inicia, ela está na fase de crescimento. Assim, pode solicitar bloqueios aos itens de dados que for acessar. Quando ela libera um bloqueio, é iniciada a fase

de encolhimento, na qual a transação não pode solicitar novos bloqueios, podendo apenas liberar os bloqueios que possui.

O schedule S_6 da Tabela 11, que contém as transações T_1 e T_4, está de acordo com o algoritmo de bloqueio em duas fases. Nesse schedule, a fase de crescimento de T_1 inicia com a operação *lock*-X(A) e dura até o primeiro desbloqueio *unlock*(A). Nesse momento, inicia a fase de encolhimento, na qual são proibidas solicitações de bloqueios. Isso obriga T_1 a manter todos os bloqueios até o início da fase de encolhimento e impede que T_4 obtenha um bloqueio sobre o item de dados A num momento inadequado. Dessa forma, o algoritmo de bloqueio em duas fases garante a serializabilidade dos *schedules*.

Tabela 11

Schedule S6 serializável

T_1	T_4
lock-X(A);	
read(A);	
A := A – 10;	
write(A);	
lock-X(B);	
read(B);	
B := B + 10;	
write(B);	
unlock(A);	
unlock(B);	
	lock-S(A);
	read(A);
	lock-S(B);
	read(B);
	unlock(A);
	unlock(B);

Fonte: Autoria Própria.

Data: 05/2016.

5.2 Algoritmo Baseado Em *Snapshot*

O algoritmo baseado em *snapshot* é utilizado em importantes bancos de dados comerciais, como SQL Server e Oracle. Não se sabe ao certo a autoria desse algoritmo, mas Silberschatz, Korth e Sudarshan (2012) apresentam de forma didática o funcionamento dele. Por isso, os conceitos aqui apresentados referenciam a obra desses autores.

No algoritmo baseado em *snapshot*, cada transação recebe uma cópia do banco de dados ao iniciar sua execução. Toda alteração realizada pela transação é feita apenas na sua cópia e, portanto, não afeta os dados do banco de dados. Quando a transação for confirmada, as alterações feitas em sua cópia são repassadas ao banco de dados.

Esse algoritmo favorece as transações somente leitura (que não modificam os dados), porque não precisam esperar autorização de bloqueio nem são revertidas pelo gerenciador de concorrência. Já para transações que modificam os dados, deve existir uma validação quando a transação é confirmada. Isso é necessário porque duas transações concorrentes podem modificar o mesmo item de dados. Como cada uma possui sua cópia do dado, nenhuma delas visualizará as modificações feitas pela outra. Portanto, no momento de confirmação das transações, o valor modificado por uma delas será sobrescrito pelo valor modificado pela outra. Isso resulta numa atualização perdida, como visto na Seção 3.

Para solucionar o problema da atualização perdida, há duas estratégias: primeiro confirmador vence e primeiro atualizador vence. Na primeira, quando uma transação T é confirmada, o sistema de controle de concorrência verifica se existe alguma outra transação concorrente a T que já gravou atualizações no banco de dados para algum item que T pretenda atualizar. Se existir, T é revertida; senão, T grava sua atualização. Dessa forma, se duas transações são conflitantes, aquela que passou primeiro por esse teste terá seus dados gravados no banco, enquanto a outra será revertida.

Na estratégia primeiro atualizador vence, é utilizado um mecanismo de bloqueio para as transações que atualizam dados (isso não afeta as operações de leitura). Se uma transação T_i for atualizar um item de dado, ela solicita um bloqueio para aquele item. Se o bloqueio for concedido, verifica-se se o dado foi modificado por alguma outra transação concorrente. Caso tenha sido, então T_i aborta; caso contrário, T_i atualiza o dado e pode prosseguir para a confirmação. Se já existir um bloqueio feito por uma operação T_j, então T_i espera Tj abortar ou confirmar. Se T_j confirmar, então T_i aborta; mas se T_j abortar, então T_i obtém o bloqueio.

Esse algoritmo popularizou-se devido ao pequeno número de transações abortadas, já que isso ocorre apenas quando duas transações concorrentes atualizam o mesmo dado. No

entanto, ele apresenta um grave problema: a serializabilidade não é garantida. Por exemplo, suponha uma transação Ti que lê dois itens de dados A e B, e atualiza A. Enquanto isso, outra transação concorrente T_j lê A e B, e atualiza B. Pelo algoritmo baseado em *snapshot* não haveria conflito entre T_i e T_j, porque elas atualizam itens de dados diferentes. No entanto, segundo o método de serializabilidade por conflito visto na Seção 4, T_i e T_j são conflitantes. Como o algoritmo baseado em *snapshot* não impede a execução concorrente de T_i e T_j, então ele não garante a serializabilidade.

6 TRANSAÇÕES EM SQL

Conforme visto na Seção 2, o padrão SQL apresenta duas instruções para a execução de transações: *commit* e *rollback*. Alguns sistemas de banco de dados comerciais também utilizam instruções para determinar o início e o fim da transação, como as instruções *begin transaction* e *end transaction* do SQL Server. Além disso, o padrão SQL também apresenta três problemas que podem acontecer quando transações são executadas de modo concorrente: a leitura suja, a leitura não repetitiva e o fenômeno fantasma. Esses problemas foram explicados na Seção 3.

O padrão SQL também define quatro níveis de isolamento de transação (ISO, 2011). Cada transação pode ser configurada em um único desses níveis. Eles foram criados de acordo com os três problemas de execução concorrente apresentados, como pode ser visto na Tabela 12. No nível REPEATABLE READ, por exemplo, não é possível acontecer a leitura suja nem a leitura não repetitiva entre transações concorrente, mas é possível ocorrer o fenômeno fantasma. Embora especifique esses níveis e sua relação com os problemas, o padrão SQL não apresenta nenhum método prático de controle de concorrência. Dessa forma, os fabricantes dos sistemas de banco de dados podem escolher qualquer algoritmo para essa tarefa.

Tabela 12

Níveis de isolamento de transação em SQL

Nível de isolamento	Leitura Suja	Leitura não repetitiva	Fenômeno fantasma
READ UNCOMMITTED	Possível	Possível	Possível
READ COMMITTED	Impossível	Possível	Possível
REPEATABLE READ	Impossível	Impossível	Possível
SERIALIZABLE	Impossível	Impossível	Impossível

Fonte: Autoria Própria.

Data: 05/2016.

Uma breve descrição de cada nível é apresentada a seguir:

- READ UNCOMMITTED: dados não confirmados podem ser lidos. Portanto, todo tipo de problema pode acontecer;
- READ COMMITTED: apenas dados confirmados podem ser lidos. Se uma transação lê um item de dados duas vezes, esse nível permite que outra transação atualize esse mesmo item de dados entre as leituras;
- REPEATABLE READ: nesse nível, apenas dados confirmados podem ser lidos. Além disso, uma transação não pode atualizar um item de dados entre duas leituras feitas por outra transação para esse mesmo item;
- SERIALIZABLE: transações nesse nível devem ser serializáveis.

7 SISTEMAS DE BANCO DE DADOS COMERCIAIS

No SQL Server 2008 (MICROSOFT, 2008), o controle de concorrência padrão é feito através de bloqueios, conforme visto na Seção 5.1. Também existe um controle conhecido como otimista, que se baseia na ideia de que os conflitos são raros e, portanto, não é eficiente bloquear recursos. No controle otimista, as transações executam sem precisar solicitar bloqueios e, quando uma transação é confirmada, o sistema verifica se as operações feitas por essa transação conflitam com as operações realizadas por outras transações.

O SQL Server aceita todos os quatro níveis de isolamento de transação definidos pelo padrão SQL, sendo o nível READ COMMITTED o padrão desse sistema. Além deles, há um quinto nível conhecido como SNAPSHOT, que na verdade é um controle de concorrência que

utiliza o algoritmo baseado em *snapshot* visto na Seção 5.2. Dessa forma, o SQL server utiliza tanto algoritmos de bloqueio como algoritmos baseados em *snapshot* para controlar a concorrência das transações.

Para transações somente leitura, o Oracle 10g (ORACLE) utiliza um algoritmo baseado em *snapshot*, fornecendo uma cópia do banco de dados para esse tipo de transação. Na verdade, o Oracle não cria uma cópia do banco de dados, mas atribui um número de versão (conhecido como SCN – *system change number*) à transação somente leitura. Quando essa transação consulta um dado, o sistema busca o valor consistente do dado no momento em que a transação foi criada (de acordo com o SCN do dado e da transação). Esse método é conhecido como múltipla versão.

Para controlar transações que modificam dados, também é utilizado um algoritmo baseado em *snapshot*. Esse algoritmo utiliza bloqueios e a estratégia do primeiro atualizador vence, vista na Seção 5.2. Quanto aos níveis de isolamento do padrão SQL, o Oracle permite apenas os níveis READ COMMITTED e SERIALIZABLE (o nível padrão é READ COMMITTED).

8 CONCLUSÃO

Nos sistemas de banco de dados, toda transação deve atender às propriedades atomicidade, consistência, isolamento e durabilidade (ACID). A atomicidade garante que todas as operações da transação serão executadas ou, em caso de falha, nenhuma delas. Segundo a consistência, o banco de dados deve estar num estado consistente quando a transação terminar. A durabilidade garante que as alterações das transações que completaram com sucesso serão armazenadas no banco de dados. Finalmente, segundo a propriedade de isolamento, nenhuma transação pode sofrer interferências de outras transações.

Como as transações executam de modo simultâneo, garantir o isolamento é uma tarefa complexa. Diversos problemas podem acontecer em execuções concorrentes, dentre eles: a atualização perdida, a leitura suja, a leitura não repetitiva e o fenômeno fantasma. Esses problemas são causados devido à intercalação das operações de diferentes transações, o que pode causar interferências entre as transações e, consequentemente, deixar o banco de dados num estado inconsistente.

Um *schedule* é uma sequência de execução das operações de um conjunto de transações. Para verificar se a execução de um *schedule* deixará o banco de dados num estado

consistente, foi apresentado o método de serializabilidade por conflito. Ele tem por objetivo determinar se um *schedule* é equivalente ao *schedule* serial que contém as transações. Um *schedule* serial é aquele onde não há intercalação entre operações de diferentes transações, ou seja, as transações estão de modo sequencial. Quando um *schedule* é equivalente a um schedule serial, diz-se que ele é serializável.

O método de serializabilidade por conflito não é eficiente porque requer que o schedule já esteja montado para determinar sua serializabilidade. Por isso, foram desenvolvidos algoritmos preventivos, ou seja, que impedem que schedules não serializáveis sejam montados. Nos algoritmos baseados em bloqueios, uma transação precisa solicitar um bloqueio antes de alterar um dado. Os bloqueios são concedidos pelo sistema de controle de concorrência e visam impedir que duas transações modifiquem um mesmo item de dado. Já nos algoritmos baseados em *snapshot*, cada transação recebe uma cópia do banco de dados e atualiza apenas os dados de sua cópia. Quando uma transação é confirmada, é feita uma validação para garantir que os dados alterados pela transação não causarão inconsistências no banco de dados.

O padrão SQL define quatro níveis de isolamento de transação, mas não apresenta nenhum algoritmo que garanta esses níveis na prática. Dessa forma, os fabricantes dos sistemas de banco de dados podem escolher qualquer algoritmo para essa tarefa. O SQL Server 2008 permite a configuração dos quatro níveis de isolamento SQL e utiliza tanto algoritmos baseados em bloqueios quanto algoritmos baseados em *snapshot*. Já o Oracle 10g permite apenas dois níveis de isolamento e utiliza algoritmos baseados em *snapshot*.

ABSTRACT

In database systems, transactions must meet four properties: atomicity, consistency, isolation and durability. The insulating property is to ensure that a transaction runs without interference from other transactions. However, as the transactions are executed simultaneously, it is possible that two transactions running concurrently access the same data item. In this case, one can interfere in the value accessed by the other. Because of this, database systems dedicate a specific module for this controlling concurrent access, known as concurrency control system. This qualitative and bibliographic work aims to present concepts related to transaction concurrency control and two important algorithms developed to perform this control. It will also be seen that the Structured Query Language (SQL), which is the standard programming language for relational database features on the subject and as two of the major commercial database systems control the competition of their transactions.

Keywords: Database systems. Concurrency control. SQL.

REFERÊNCIAS

ESWARAN, K. P.; GRAY, J. N.; LORIE, R. A.; TRAIGER, I. L. **The Notions of Consistency and Predicate Locks in a Database System**. 1976. Disponível em < http://research.microsoft.com/en-us/um/people/gray/papers/On%20the%20Notions%20of%20Consistency%20and%20Predicate%20Locks%20in%20a%20Database%20System%20CACM.pdf>. Acesso em: 03 de maio 2016.

GRAY, J. N.; REUTER, A. **Transaction Processing: Concepts and Techniques**. 1 ed. Morgan Kaufmann: 1993.

HAERDER, T.; REUTER, A. **Principles of Transaction-Oriented Database Recovery**. 1983. Disponível em <http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/HaerderReuter83.pdf>. Acesso em: 03 de maio 2016.

ISO. **ISO/IEC 9075-2:2011. Information technology -- Database languages -- SQL -- Part 2: Foundation (SQL/Foundation)**. 2011. Disponível em < http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53682>. Acesso em: 03 de maio 2016.

MICROSOFT. Isolation Levels in the Database Engine. 2008. Disponível em <https://technet.microsoft.com/en-us/library/ms189122%28v=sql.105%29.aspx>. Acesso em: 03 de maio 2016.

NAVATHE, S. B.; ELMASRI, R. **Sistema de Banco de Dados**. 6 ed. São Paulo: Pearson Addison Wesley, 2011.

ORACLE. **Data Concurrency and Consistency**. Disponível em < https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm>. Acesso em: 03 de maio 2016.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de Banco de Dados**. 6 ed. Rio de Janeiro: Elsevier, 2012.