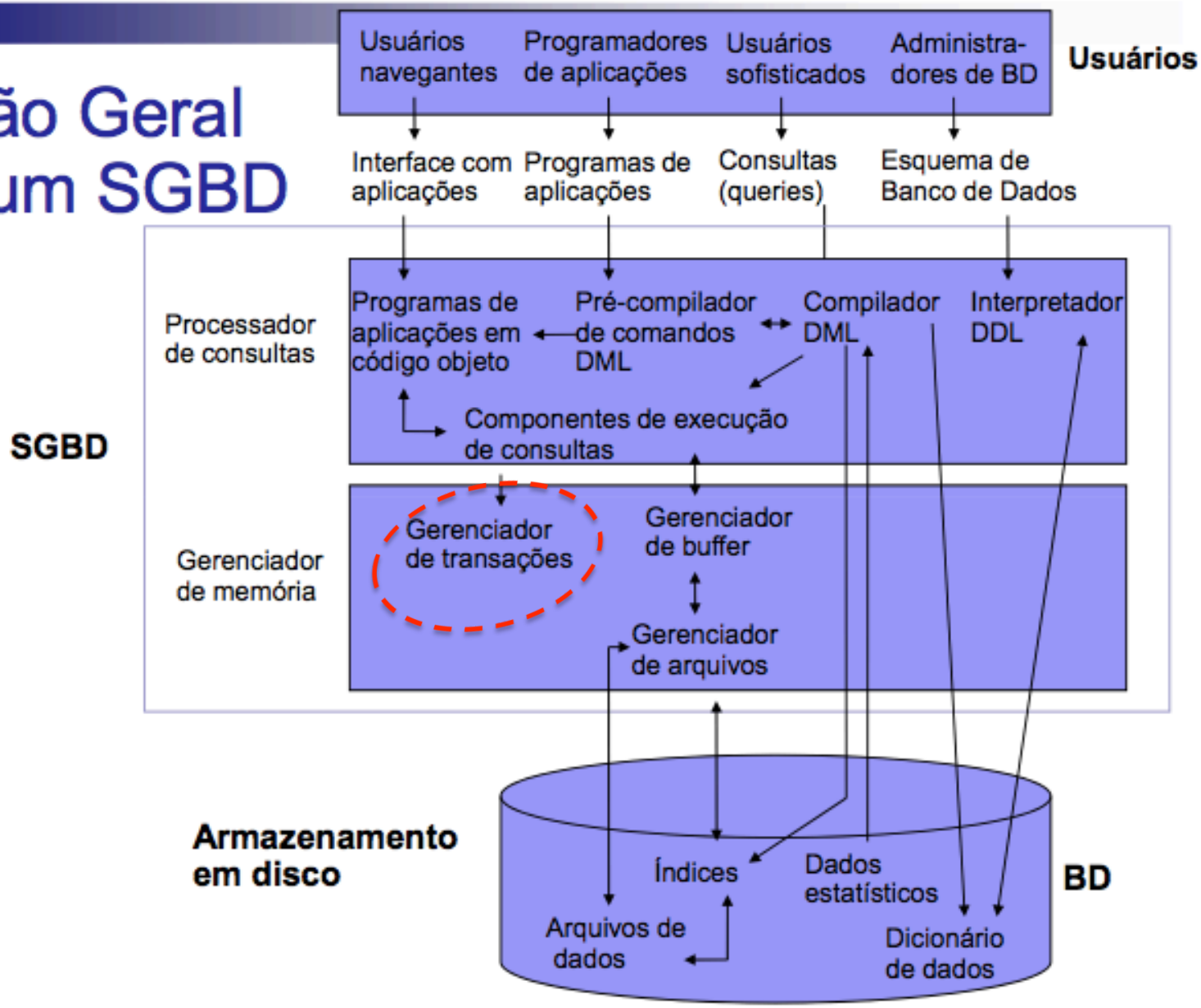
Capítulo 15: Transações

Prof. Heloise Manica P. Teixeira

Visão Geral de um SGBD



Conceito de transação

- Uma transação é uma *unidade* da execução de programa que acessa e possivelmente atualiza vários itens de dados.
- Delimitada por declarações da forma **begin transaction** e **end transaction**
- Uma transação precisa ver um banco de dados consistente.
- Durante a execução da transação, o banco de dados pode ser temporariamente inconsistente.
- Quando a transação é completada com sucesso (é confirmada), o banco de dados precisa ser consistente.

Conceito de transação

- Após a confirmação da transação, as mudanças que ele faz no banco de dados persistem, mesmo se houver falhas de sistema.
- Várias transações podem ser executadas em paralelo.
- Dois problemas principais para resolver:
  - Falhas de vários tipos, como falhas de hardware e falhas de sistema
  - Execução simultânea de múltiplas transações

Propriedades ACID

- **Atomicidade. Ou todas as operações da transação são refletidas corretamente no banco de dados ou nenhuma delas é.**
- **Consistência. A execução de uma transação isolada preserva a consistência do banco de dados.**
- **Isolamento. Embora várias transações possam ser executadas simultaneamente, cada transação precisa estar desinformada das outras transações que estão sendo executadas ao mesmo tempo. Os resultados da transação intermediária precisam estar ocultos das outras transações sendo executadas simultaneamente.**
- **Durabilidade. Depois que uma transação for completada com sucesso, as mudanças que ela fez ao banco de dados persistem, mesmo que existem falhas no sistema.**

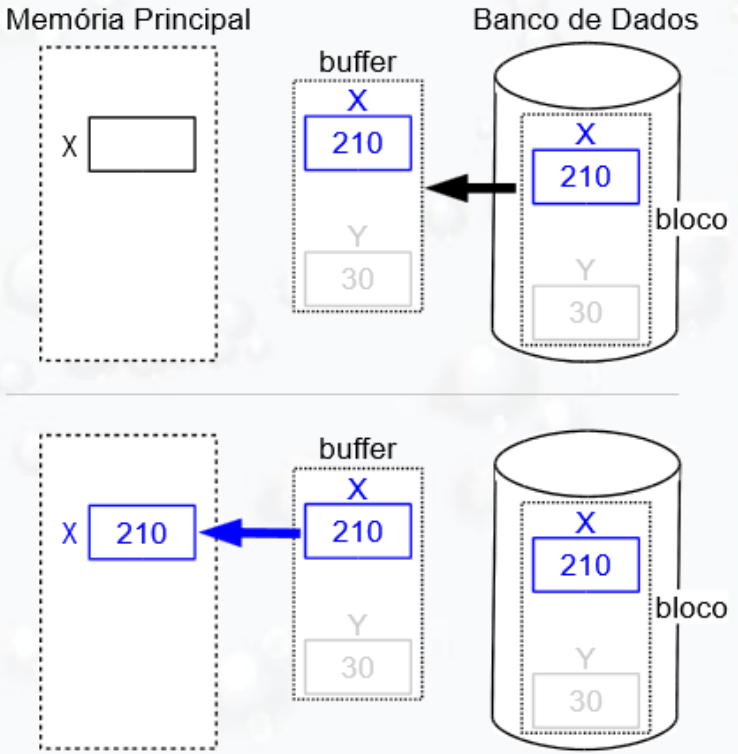
Principais operações:

- Read(X) – transfere o item de dado X **do BD** para um **buffer local** alocado à transação que executou read.
- Write(X) - transfere o item de dado X do **buffer local** que executou o write de volta **ao BD.**

- → Em um SBD real a operação write não resulta necessariamente na atualização imediata dos dados no disco; pode ser armazenada temporariamente na memória e ser executada depois.
- → Por enquanto vamos supor que uma operação write atualiza o BD imediatamente

Como acontece a operação de leitura? (Emasri, 2010)

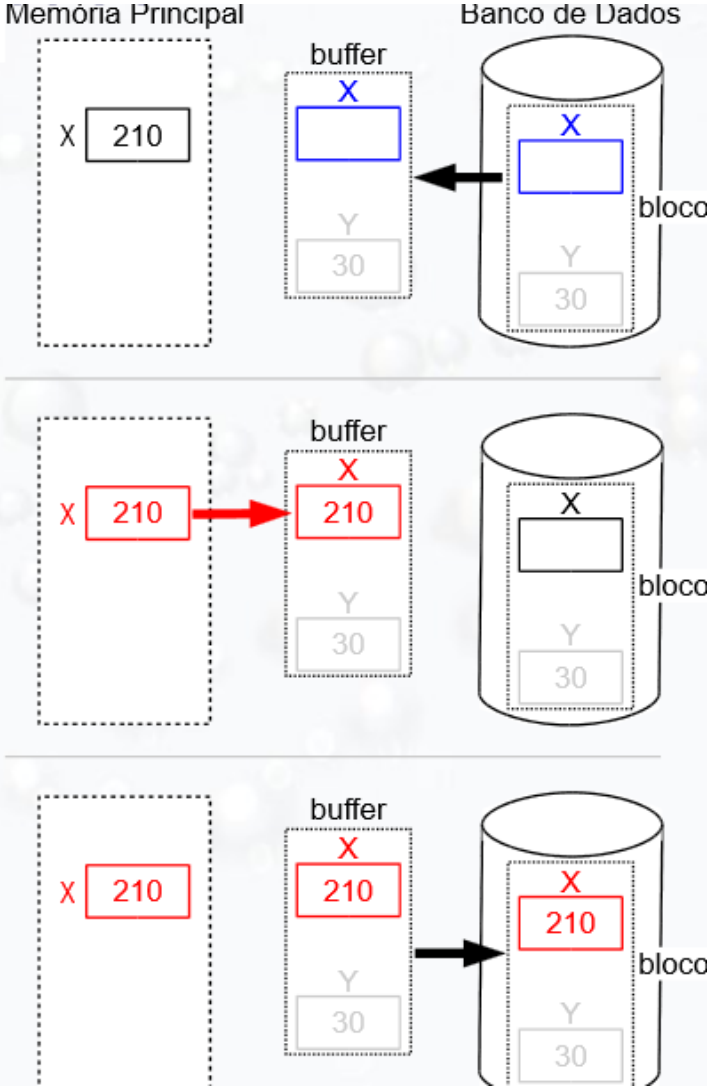
- ler(X)
  - encontra bloco X no disco
  - copia bloco para buffer da memória principal (se ainda não estiver lá)
  - copia o item X do buffer para a variável X da memória principal



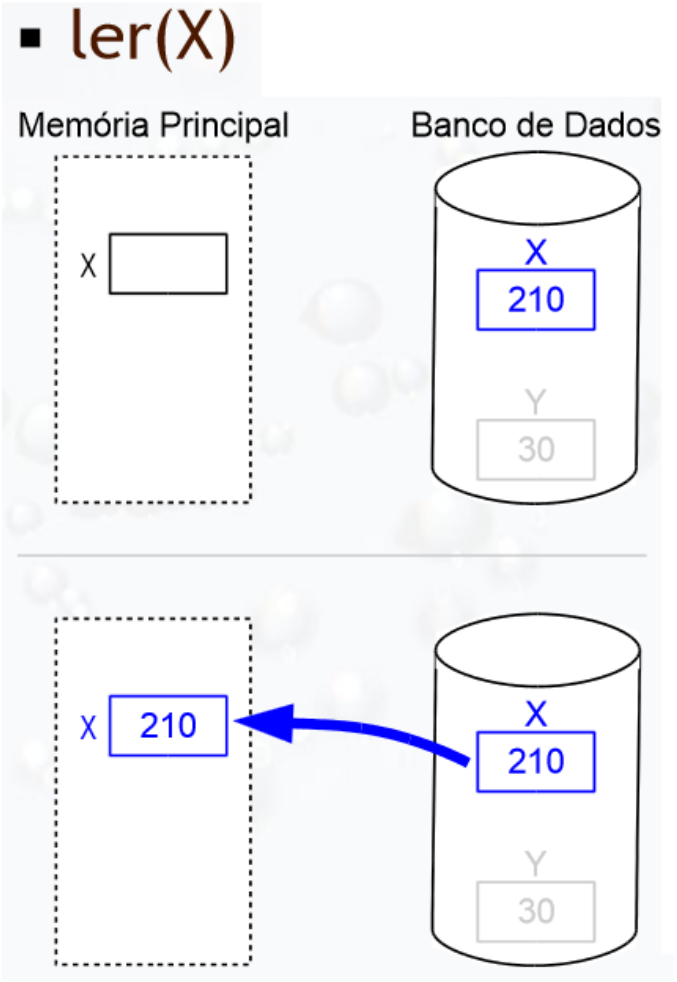
Como acontece a operação de gravação? (Emasri, 2010)

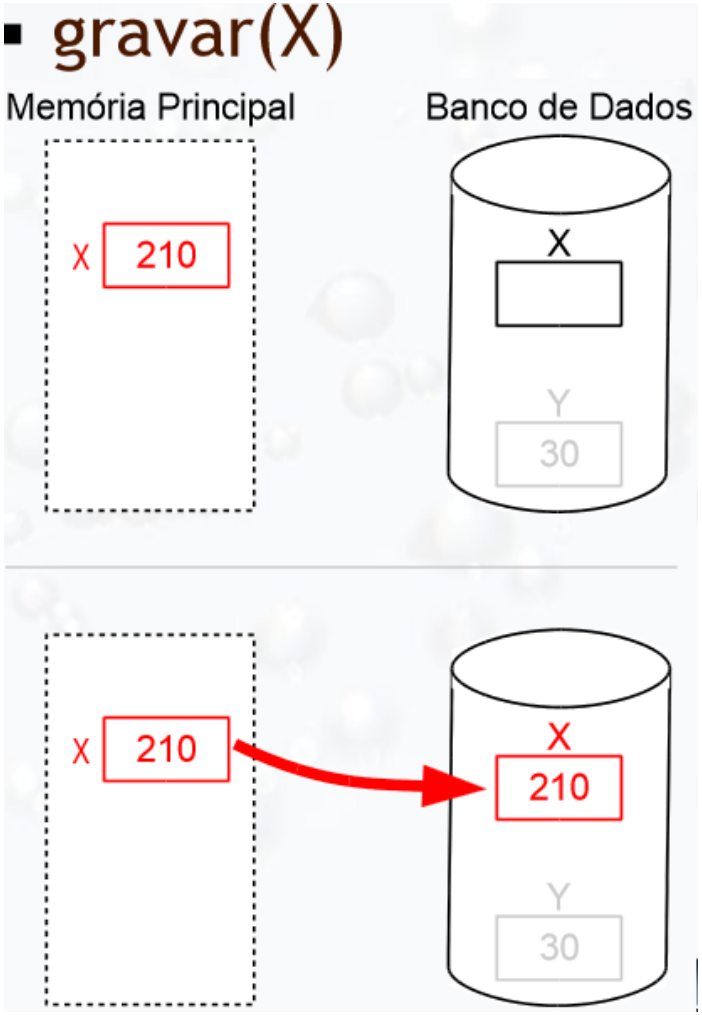
- gravar(X)
  - encontra bloco X no disco
  - copia bloco para buffer da memória principal (se ainda não estiver lá)
  - copia variável X da memória principal para o buffer
  - atualiza o buffer no disco

(Elmasri, 2010)



Operação de Leitura e Gravação: como **abstraímos**?





- → Por enquanto vamos supor que uma operação write atualiza o BD imediatamente

Exemplo de transferência de fundos (cont.)

1. read(***A***)
2. ***A*** **:= *A* – 50**
3. write(***A***)
4. read(***B***)
5. ***B*** **:= *B* + 50**
6. write(***B***)

- Atomicidade
  - Se a transação falhar após a etapa 3 e antes da etapa 6, o sistema deve garantir que suas atualizações não sejam refletidas no BD, ou uma inconsistência irá resultar.
  - Assegurar atomicidade é responsabilidade do próprio SBD
- Consistência
  - A soma de A e B é inalterada pela execução da transação.
  - Responsabilidade do programador (que codifica a transação)

Exemplo de transferência de fundos (cont.)

1. read(***A***)
2. ***A*** **:= *A* – 50**
3. write(***A***)
4. read(***B***)
5. ***B*** **:= *B* + 50**
6. write(***B***)

- Isolamento — Se entre as etapas 3 e 6, outra transação receber permissão de acessar o BD parcialmente atualizado, ele verá um banco de dados inconsistente (a soma *A* + *B* será menor do que deveria ser).
- Isso pode ser trivialmente assegurado executando transações serialmente, ou seja, uma após outra.
- Entretanto, executar múltiplas transações simultaneamente oferece vantagens significativas, como veremos mais adiante.
- Responsabilidade do SBD (controle de concorrência)

Exemplo de transferência de fundos (cont.)

1. read(***A***)
2. ***A*** **:= *A* – 50**
3. write(***A***)
4. read(***B***)
5. ***B*** **:= *B* + 50**
6. write(***B***)

- Durabilidade
  - Quando o usuário é notificado de que a transação está concluída, as atualizações no BD pela transação precisam persistir a pesar de falhas.
  - Assegurar a durabilidade é responsabilidade do SBD (componente de gerenciamento de recuperação)

Exercício (Poscomp, 2008)

Considere que as transações T1 e T2 abaixo possam ocorrer simultaneamente. Analise as duas seguintes situações.

I. A operação Leitura(A) de **T2** é executada após a operação Escrita(A) e antes da operação Leitura(B) de **T1**. Entretanto, a operação Escrita(B) de **T1** causa uma violação de integridade, e a transação **T1** é abortada, sendo suas operações desfeitas.

II. Após as operações da transação **T1** terem sido executadas, é enviada uma mensagem ao usuário informando que a transação foi completada com êxito. Entretanto, antes que os buffers relativos a **T1** sejam descarregados para o meio físico, ocorre uma falha, e os dados não são efetivamente gravados.

As propriedades das transações que foram violadas são, respectivamente,

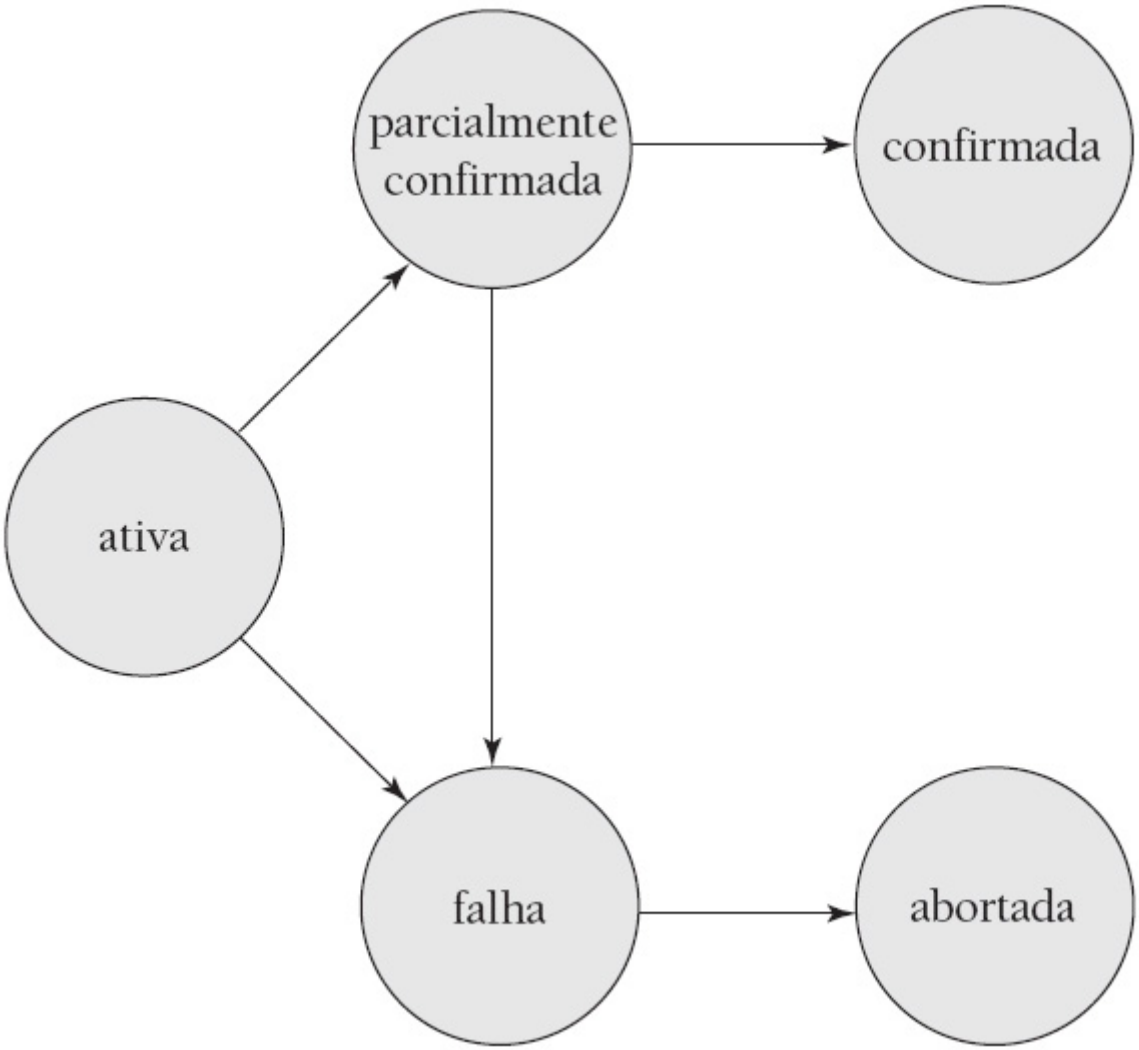
I.Atomicidade e Consistência.
II.Durabilidade e Atomicidade.
III.Atomicidade e Durabilidade.
IV.Durabilidade e Isolamento.
V.Isolamento e Durabilidade.

T1	T2
Leitura(A); A = A + 100; Escrita(A); Leitura(B); B = B - 100; Escrita(B);	Leitura(B); Leitura(A); Print (A+B);

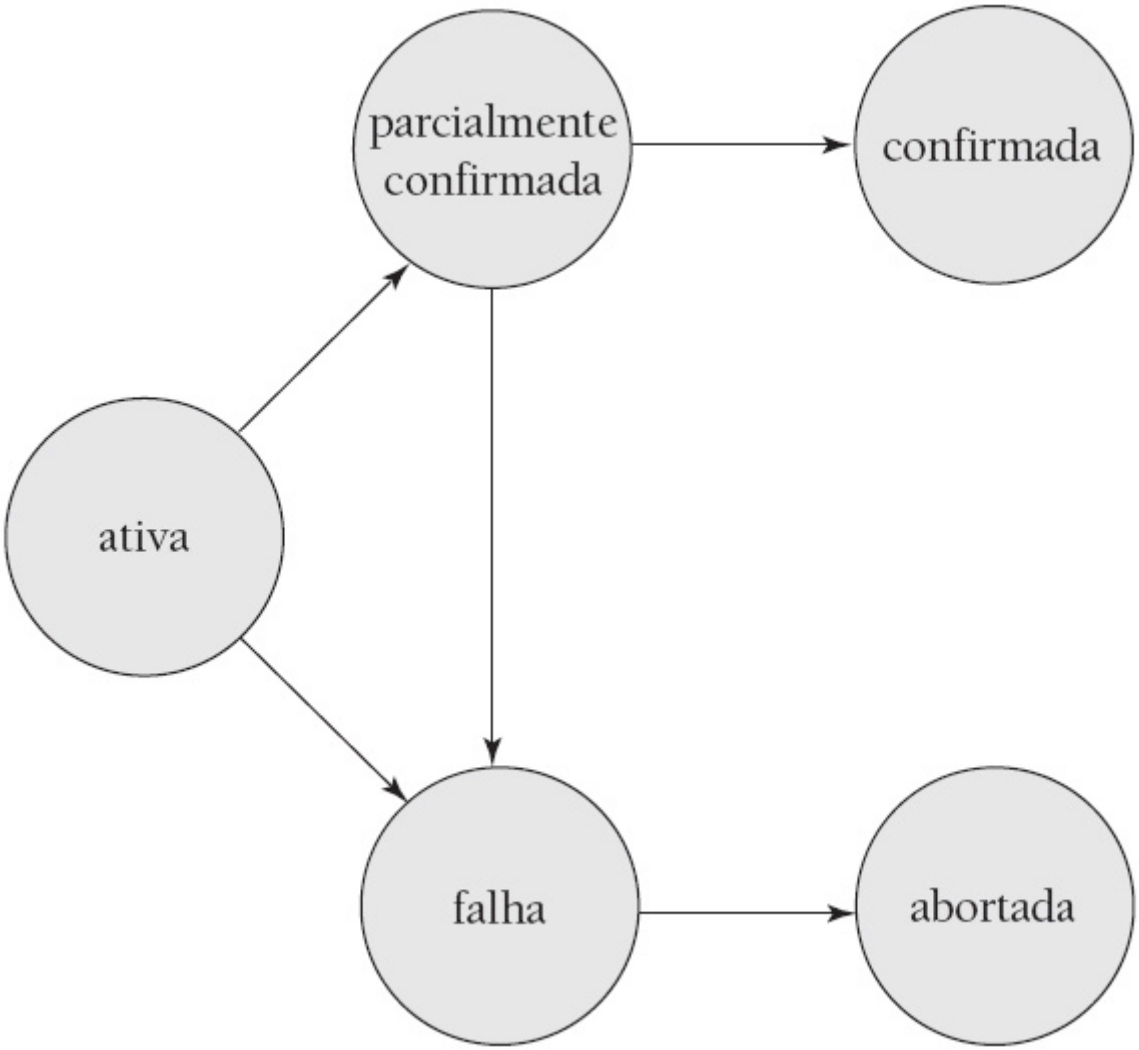
Estado da transação

- **Ativa – O estado inicial; a transação permanece nesse estado enquanto está executando**
- **Parcialmente confirmada – Depois que a instrução final foi executada**
- **Falha – Depois da descoberta de que a execução normal não pode mais prosseguir**
- **Abortada – Depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação. Duas opções após ter sido abortada:**
  - **Reiniciar a transação; pode ser feito apenas se não houver qualquer erro lógico interno**
  - **Excluir a transação**
- **Confirmada – Após o término bem sucedido**

Estado da transação (cont.)



Estado da transação (cont.)



Execuções simultâneas

- Várias transações podem ser executadas simultaneamente no sistema. As vantagens são:
  - Melhor utilização do processador e do disco, levando a um melhor *throughput* de transação: uma transação pode estar usando a CPU enquanto outra está lendo ou escrevendo no disco
  - Tempo de médio de resposta reduzido para transações: as transações curtas não precisam esperar atrás das longas
- Esquemas de controle de concorrência – mecanismos para obter isolamento; ou seja, para controlar a interação entre as transações concorrentes a fim de evitar que elas destruam a consistência do banco de dados

  → Estudaremos mais adiante

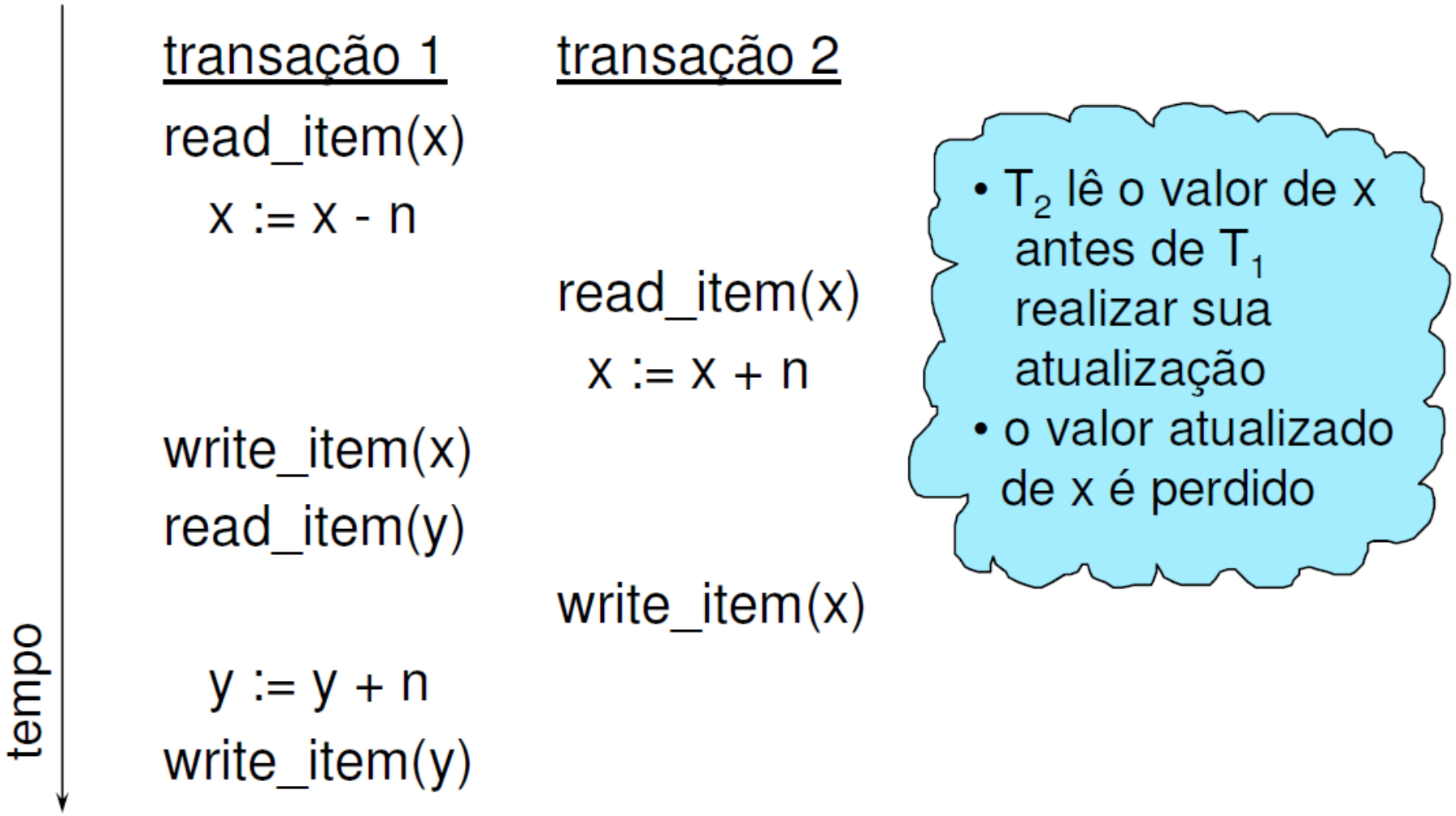
Por que é Necessário o Controle de Concorrência?

- Alguns problemas:
  - Atualizações perdidas
  - Leitura Incorreta
  - Somas incorretas

Problema de alterações perdidas

- Ocorre quando duas transações que acessam os mesmos dados do BD têm suas operações intercaladas de modo a gerar um valor incorreto de algum item de dado
- Exemplo:
  - dois programadores escrevem o mesmo programa ao mesmo tempo
  - cada cópia é alterada independentemente
    - a cópia 1 substitui a versão original
    - a cópia 2 substitui a versão 1
  - cópia 1: alterações perdidas

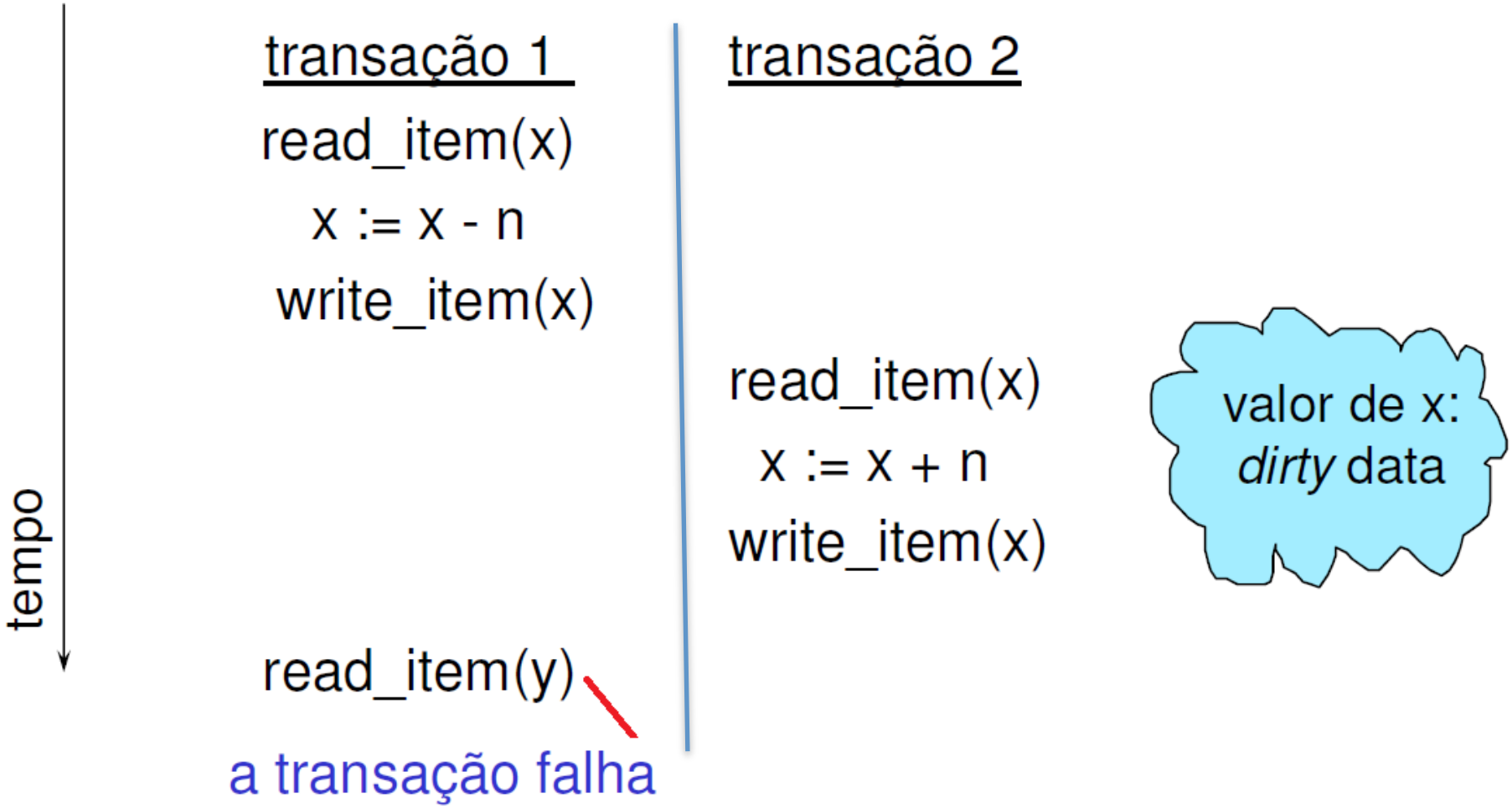
Problema de alterações perdidas



Problema da Leitura Incorreta

- Cenário
  - quando uma transação primeiro atualiza o valor de um item de dado do BD e em seguida falha
  - o item de dado alterado é acessado por outra transação antes de voltar a possuir seu valor original

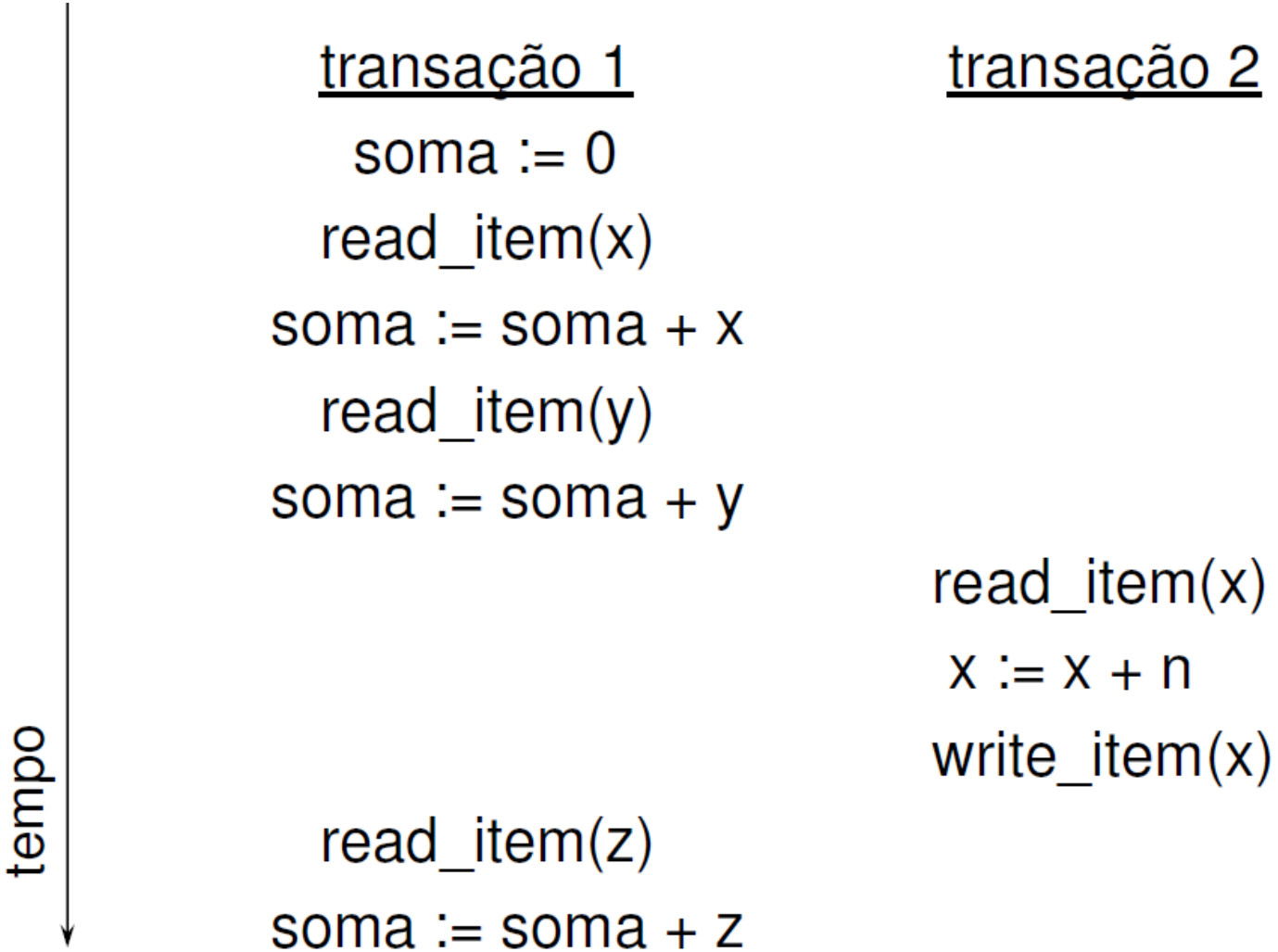
Problema da Leitura Incorreta



Problema das somas incorretas

- Cenário
  - quando uma transação está realizando uma soma de vários itens de dados, enquanto que outras transações estão alterando alguns deste itens
  - a soma pode utilizar valores dos itens de dados antes destes serem alterados e depois destes serem alterados

Problema das somas incorretas



Schedules (Escalas)

- **Schedules (escalas de execução)**
  - **Sequências de instruções que especificam a ordem cronológica em que as instruções das transações simultâneas são executadas**
  - **Um schedule para um conjunto de transações precisa consistir em todas as instruções dessas transações**
  - **Precisam preservar a ordem em que as instruções aparecem em cada transação individual**
- **Uma transação que completa com sucesso sua execução terá uma instrução commit como a última instrução (será omitida se for óbvia)**
- **Uma transação que não completa com sucesso sua execução terá instrução abort como a última instrução (será omitida se for óbvia)**

Schedule 1

- **Suponha que T_1 transfere US$ 50 de *A* para *B* e T_2 transfere 10% do saldo de *A* para *B*. A seguir está um schedule serial em que T_1 é seguido de T_2.**

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- **Sejam T_1 e T_2 as transações definidas anteriormente. O schedule a seguir é uma escala concorrente, mas é equivalente ao Schedule 1.**

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 3

- **O seguinte schedule concorrente não preserva o valor da soma *A* + *B*.**

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serialização

- **Suposição básica – Cada transação preserva a consistência do banco de dados**

Portanto, a execução serial de um conjunto de transações preserva a consistência do banco de dados

Um schedule (possivelmente simultâneo) é serializável se for equivalente a um schedule serial.

1. **Seriação de conflito**
2. **Seriação de view**

- **Uma transação pode realizar uma sequência qualquer de operações sobre a cópia de um item de dado X que esta residindo no buffer local da transação.**

- **Assim, ignoramos todas as operações exceto read e write, writes, as únicas operações significativas.**

Instruções conflitantes

- **As instruções I_i e I_j das transações T_i e T_j respectivamente, estão em conflito se e somente se algum item *Q* é acessado por I_i e por I_j e pelo menos uma destas instruções escreveram *Q*.**

  1. I_i = read(*Q*), I_j = read(*Q*). I_i e I_j não estão em conflito
  2. I_i = read(*Q*), I_j = write(*Q*). Estão em conflito (ordem importa)
  3. I_i = write(*Q*), I_j = read(*Q*). Estão em conflito (ordem importa)
  4. I_i = write(*Q*), I_j = write(*Q*). Estão em conflito (ordem importa)

  – **Assim, somente no caso em que I_i e I_j são instruções read, a ordem relativa de sua execução não importa**

Seriação de conflito

- **Diz-se que duas instruções entram em conflito se elas são operações pertencentes a transações diferentes, agindo no mesmo item de dado, e pelo menos uma dessas instruções é uma operação de escrita.**

- **Exemplo:**

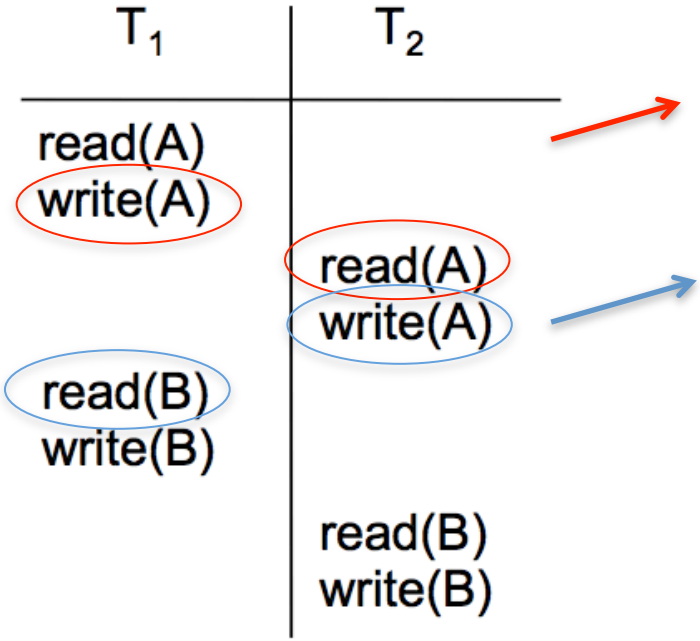
I_i = read(Q) e I_j = write(Q)

Se I_i vier antes de I_j, então T_i não lê o valor de Q que é escrito por T_j na instrução I_j.

Se I_j vier antes de I_i, então T_i lê o valor de Q que é escrito por T_j.

Assim, a ordem de I_i e I_j importa.

Seriação de conflito (cont.)



A instrução write(A) de T_1 entra em conflito com a instrução read(A) de T_2.

Porém, a instrução write(A) e T_2 não está em conflito com a instrução read(B) de T_1.

Seriação de conflito

- Seja I_i e I_j instruções consecutivas de uma escala de execução *S*.
- Se I_i e I_j são instruções de transações diferentes e não entram em conflito, então podemos trocar a ordem de I_i e I_j para produzir uma nova escala de execução *S'*.
- Diz-se que *S* e *S'* são equivalentes já que todas as instruções aparecem na mesma ordem em ambas as escalas de execução com exceção de I_i e I_j, cuja ordem não importa.

Seriação de conflito (cont.)

- **Se um schedule S puder ser transformado em um schedule S' por uma série de trocas de instruções não conflitantes, dizemos que S e S' são equivalentes em conflito**
- **Um schedule S é serial de conflito se for equivalente em conflito a um schedule serial.**
- **Exemplo de um schedule que não é serial de conflito:**

T_3	T_4
read(**Q**)	
	write(**Q**)
write(**Q**)	

Pois não podemos trocar instruções no schedule acima para obter o schedule serial < T_3, T_4 >, ou o schedule serial < T_4, T_3 >.

Exercício – Adptado do Poscomp, 2011

Considere as escalas S1 de execução de transações (T).

S1

T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

S2

T1	T2	T3
Read (A)		
	Write (A)	
Write (A)		
		Write (A)

S1 é serializável no conflito?

S2 é serializável no conflito?

Seriação de view

- **Sejam *S* e *S´* dois schedules com o mesmo conjunto de transações. *S* e *S´* são equivalentes em view se as três condições a seguir forem satisfeitas:**

  1. **Para cada item de dados *Q*, se a transação T_i ler o valor inicial de *Q* no schedule *S*, então, T_i precisa, no schedule *S´*, também ler o valor inicial de *Q*.**
  2. **Para cada item de dados *Q*, se a transação T_i executar** read(***Q***) **no schedule *S*, e se esse valor lido foi produzido pela transação T_j (se houver), então a transação T_i, no schedule *S'*, também precisa ler o valor de *Q* que foi produzido pela transação T_j.**
  3. **Para cada item de dados *Q*, a transação (se houver) que realiza a operação** write(***Q***) **final no schedule *S* precisa realizar a operação** write(***Q***) **final no schedule *S'*.**

→ **Como podemos ver, a equivalência em view também é baseada unicamente em** reads **e** writes **isolados.**

A escala 1 **não** é equivalente em visão à escala 2, já que na escala 1 o valor de A lido pela transação T2 foi produzido por T1, enquanto isso não ocorre na escala 2.

Escala 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Escala 2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	

Seriação de view (cont.)

- Um schedule *S* é serial de view se ele for em equivalente view a um schedule serial
- Todo schedule serial de conflito também é serial de view
- Exemplo de um schedule *não é* serial de conflito, mas é serial de view, pois
  - é equivalente à escala <T3, T4, T6> serial, já que um read(Q) lê o valor inicial de Q em ambas as escalas e T6 executa a escrita final de Q em ambas as escalas

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Todo schedule que é serial de view que não é serial de conflito possui escritas cegas

Exercício – Adptado do Poscomp, 2011

Considere as escalas S1 de execução de transações (T).

S1

T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

S2

T1	T2	T3
Read (A)		
	Write (A)	
Write (A)		
		Write (A)

S1 é serializável na visão?

S2 é serializável na visão?

Recuperação

- Se uma transação T1 falhar, para garantir atomicidade, seus efeitos devem ser desfeitos
  - Também deve-se desfazer os efeitos de qualquer transação T2 que leu dados escritos por T1

- Para alcançar essa segurança, é necessário permitir apenas escalas de execução recuperáveis
  - Uma escala é recuperável se para cada par de transações Ti e Tj, tal que Tj leia itens de dados de Ti, a operação de efetivação de Ti apareça antes da efetivação de Tj.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recuperação

- Para o sistema se recuperar corretamente da falha de uma transação Ti pode ser necessário desfazer diversas transações (retorno em cascata)

- Rollback em cascata – Uma única falha de transação leva a uma série de rollbacks de transação.

Recuperação

- **Considere o seguinte schedule onde nenhuma das transações ainda foi confirmada (portanto, o schedule é recuperável)**

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

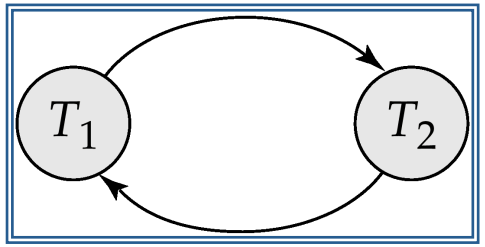
- **Se T_{10} falhar, T_{11} e T_{12} também precisam ser revertidos**
- **Pode chegar a desfazer uma quantidade de trabalho significativa**

Testando a seriação

- **Esquemas de controle de concorrência devem ter escalas serializáveis**
- **Como saber se uma escala é serializável?**
  - **Serialização de conflito – existe algoritmo eficiente**
  - **Serialização de visão – não existe algoritmo eficiente**

Testando a seriação de Conflito

- G = (V,E) em que V é um conjunto de vértices e E é um conjunto de arestas.
  - O conjunto de vértices é composto por todas as transações que participam da escala.
  - O conjunto de arestas consiste em todas as arestas $T_i \rightarrow T_j$ para as quais uma das seguintes condições é verdadeira:
    - T_i executa **write**(*Q*) antes de T_j executar **read**(*Q*);
    - T_i executa **read**(*Q*) antes de T_j executar **write**(*Q*);
    - T_i executa **write**(*Q*) antes de T_j executar **write**(*Q*);



Exemplo de schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5	
	read(X)				
read(Y)					
read(Z)					
					read(V)
					read(W)
					read(W)
	read(Y)				
	write(Y)				
		write(Z)			
read(U)					
				read(Y)	
				write(Y)	
				read(Z)	
				write(Z)	
read(U)					
write(U)					

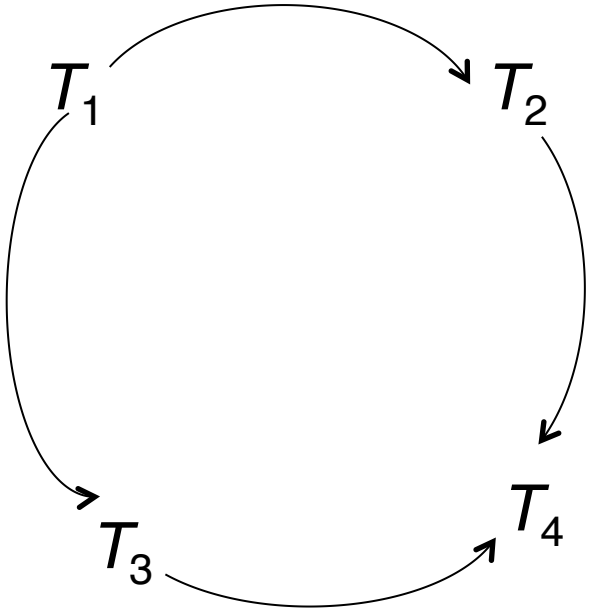
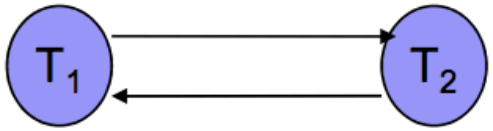


Gráfico de precedência

Teste para seriação de conflito

- **Um schedule é serial de conflito se e somente se seu gráfico de precedência for acíclico**

T_1	T_2
read(A)	
A := A - 50	
	read(A)
	temp := A * 0,1
	A := A – temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B := B + temp;
	write(B)



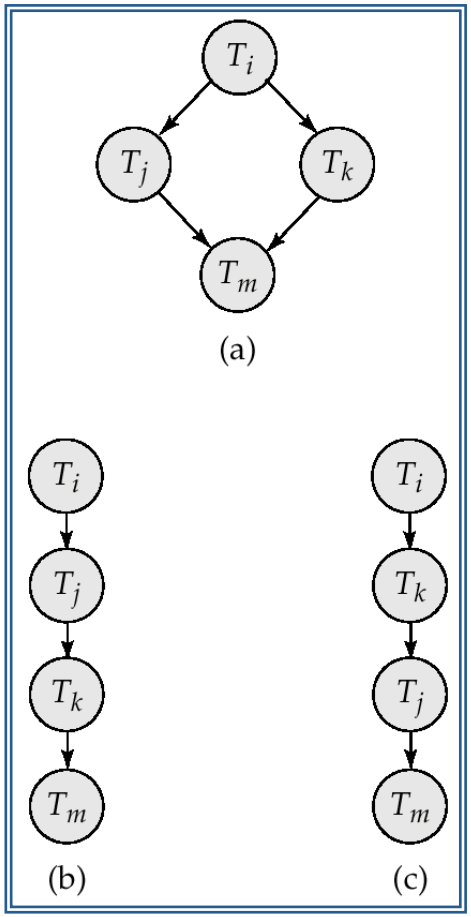
Ciclo no grafo: Se o grafo de precedência possui ciclo, então a escala *S* não é serializável por conflito.

A ordem de serialização pode ser obtida por meio da classificação topológica, que estabelece uma ordem linear para a escala consistente com a ordem parcial do grafo de precedência.

Teste para seriação de conflito

- **Existem algoritmos de detecção de ciclo que assumem a ordem cronológica n^2, onde n é o número de vértices no gráfico. (Algoritmos melhores assumem a ordem $n + e$, onde e é o número de arestas.)**
- **Se o gráfico de precedência for acíclico, a ordem de seriação pode ser obtida por meio da *classificação topológica* do gráfico. Essa é a ordem linear com a ordem parcial do gráfico.**

Ilustração da classificação topológica



Teste para seriação de view

- O teste do gráfico de precedência precisa ser modificado para se aplicar a um teste para seriação de view
- O problema de verificar se um schedule é serial de view entra na classe dos problemas *não procedurais* completos.
- Portanto, embora a existência de um algoritmo eficiente seja improvável, os algoritmos práticos que simplesmente verificam algumas *condições suficientes* para a seriação de view ainda podem ser usados.

Transações em SQL

- Na SQL, uma transação inicia implicitamente
  - primeiro comando SQL executável
- Uma transação na SQL termina por:
  - Commit
    - Finaliza a transação com sucesso
    - Atualiza os dados do BD (torna persistente as alterações feitas pela transação);
  - Rollback
    - Finaliza a transação sem sucesso
    - Desfaz os efeitos das operações da transação

Transações em SQL

- Níveis de consistência especificados pela SQL-92:
  - Serializable — padrão
  - Repeatable read
  - Read committed
  - Read uncommitted

Níveis de consistência na SQL-92

- Serializable
- Repeatable read — Apenas registros confirmados podem ser lidos; reads repetidos do mesmo registro precisam retornar o mesmo valor.
- Read committed — Apenas registros confirmados podem ser lidos, mas reads sucessivos do registro podem retornar valores diferentes (mas confirmados)
- Read uncommitted — Mesmo registros não confirmados podem ser lidos

→ Graus de consistência mais baixos são úteis para coletar informações aproximadas sobre o banco de dados; por exemplo, estatística para otimizador de consulta.

Bibliografia

- Sistemas de Banco de Dados (Cap. 15). Abraham Silberchatz, Henry F. Korth, S Sudarshan. 5a Ed. Elsevier, 2006.