TRANSAÇÕES - SERIALIZAÇÃO DE CONFLITO

Sérgio Mergen

Versão modificada de Database System Concepts, 5th Ed.

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Conceito de Transação

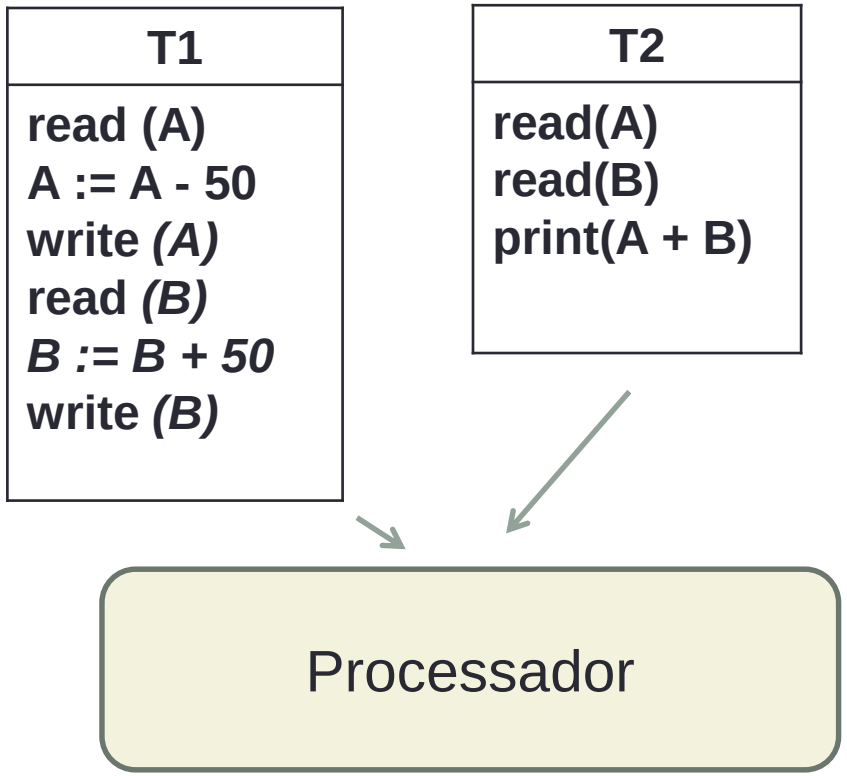
- Uma **transação** é um programa que acessa e possivelmente altera diversos itens de dados.
  - Um item de dados pode ser qualquer objeto que armazene valor

- Ex. a transação ao lado (T1) transfere $50 da conta A para a conta B:
  - Para simplificar, os itens de dados (A e B) são saldos de contas

T1
read (A) **A := A - 50** **write *(A)*** **read *(B)*** **B = B + 50** **write *(B)***

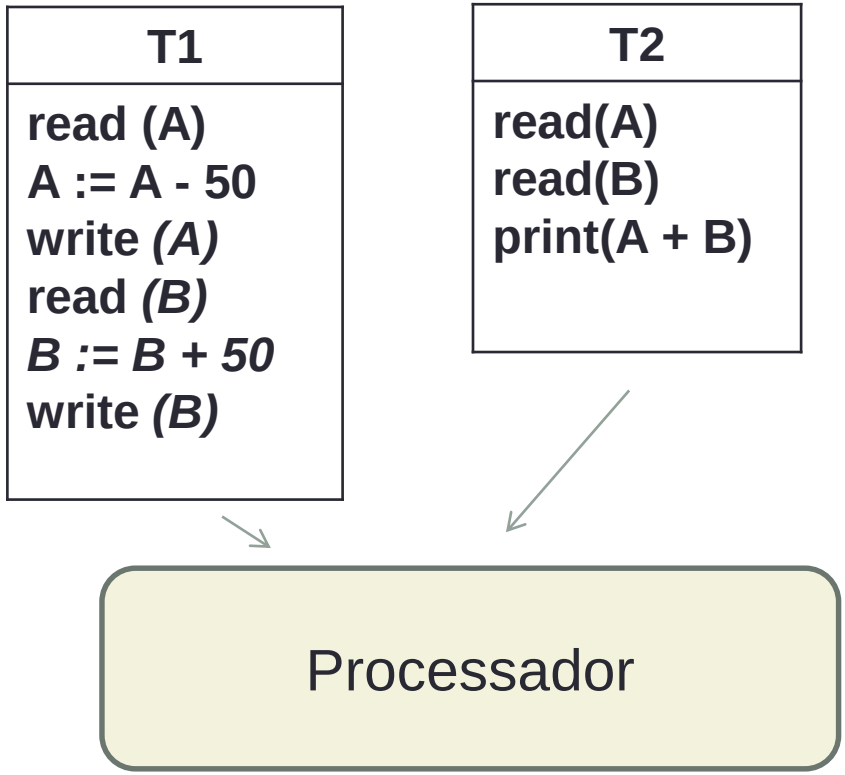
Conceito de Transação

- Várias transações podem ser disparadas ao mesmo tempo
- E disputarem o uso do processador



Conceito de Transação

- Riscos para a manutenção da integridade dos dados
  - Falhas de diversos tipos, como falhas de hardware e travamento de sistemas
  - Execução concorrente de múltiplas transações



Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Propriedades ACID

- Para preservar a integridade do banco de dados, um SGBD deve garantir as propriedades ACID:
  - Atomicidade
  - Consistência
  - Isolamento
  - Durabilidade

Exemplo de Transferência de Fundos

- **Atomicidade**
  - Se a transação falhar depois do passo 3 e antes do passo 6, dinheiro será “perdido”, levando o banco de dados a um estado inconsistente
  - O sistema deve garantir que as atualizações de uma transação parcialmente executada não sejam refletidas no banco de dados

T1
1. read (A) **2. A := A - 50** **3. write *(A)*** **4. read *(B)*** **5. B = B + 50** **6. write *(B)***

Exemplo de Transferência de Fundos

- **Durabilidade**
  - uma vez que o usuário tenha sido notificado que a transação completou (ex., que a transferência foi bem sucedida)
    - as atualizações decorrentes da transação devem persistir mesmo que ocorram falhas posteriores.

T1
1. read (A) **2. A := A - 50** **3. write *(A)*** **4. read *(B)*** **5. B = B + 50** **6. write *(B)***

Exemplo de Transferência de Fundos

- **Consistência**:
  - A execução de uma transação deve deixar o banco em um estado correto
  - Por exemplo, no caso ao lado, o banco ficaria inconsistente se
    - A soma de A e B mudar após a execução da transação
  - Nota: inconsistências podem ocorrer por erros de lógica da transação
    - Nesse caso, não há nada que o banco SGBD possa fazer

T1
1. read (A) **2. A := A - 50** **3. write *(A)*** **4. read *(B)*** **5. B = B + 50** **6. write *(B)***

Exemplo de Transferência de Fundos

- **Isolamento**
  - Requisito necessário quando transações diferentes disputam uso do processador
    - Transações concorrentes
  - Mesmo que as transações usem o processador de forma intercalada
    - A impressão é a de que elas são executadas uma após a outra

T1	T2
1. read (A) **2.A := A - 50** **3. write *(A)*** **4. read *(B)*** **5. *B := B + 50*** **6. write *(B)***	 **1.read(A)** **2.read(B)** **3.print(A + B)**

Exemplo de Transferência de Fundos

- **Isolamento**
  - Entre os passos 3 e 6 em T1, outra transação T2 teve acesso ao banco parcialmente atualizado
    - T2 verá um banco inconsistente
    - a soma *A* + *B* será menor do que deveria ser.

T1	T2
1. read (A) **2.A := A - 50** **3. write *(A)*** **4. read *(B)*** **5. *B := B + 50*** **6. write *(B)***	 **1.read(A)** **2.read(B)** **3.print(A + B)**

Exemplo de Transferência de Fundos

- **Isolamento**
  - pode ser garantido de forma simples pela execução **serial** das transações
    - Isto é, uma após a outra.
  - Entretanto, executar múltiplas transações de forma concorrente traz benefícios, como veremos mais adiante

T1	T2
1. read (A) **2.A := A - 50** **3.write *(A)*** **4.read *(B)*** **5.*B := B + 50*** **6.write *(B)***	**1.read(A)** **2.read(B)** **3.print(A + B)**

Propriedades ACID

- **Atomicidade.** Ou todas as operações de uma transação são refletidas no banco, ou nenhuma é.

- **Consistência.** A execução de uma transação isolada preserva a consistência do banco.

- **Durabilidade.** Depois que uma transações finalizou, as mudanças feitas devem persistir mesmo que ocorram falhas posteriores.

Propriedades ACID

- **Isolamento.** Mesmo que muitas transações executem concorrentemente, cada transação desconhece as demais execuções.

- Os resultados intermediários de uma transação devem ser escondidos das demais transações em execução.

- Isto é, para cada par de transações T_x e T_y
  - *De ponto de vista de T_y , parece que*
    - *T_x, finalizou antes de T_y* iniciar
    - ou T_x iniciou depois de T_y finalizar.

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Estados de uma Transação

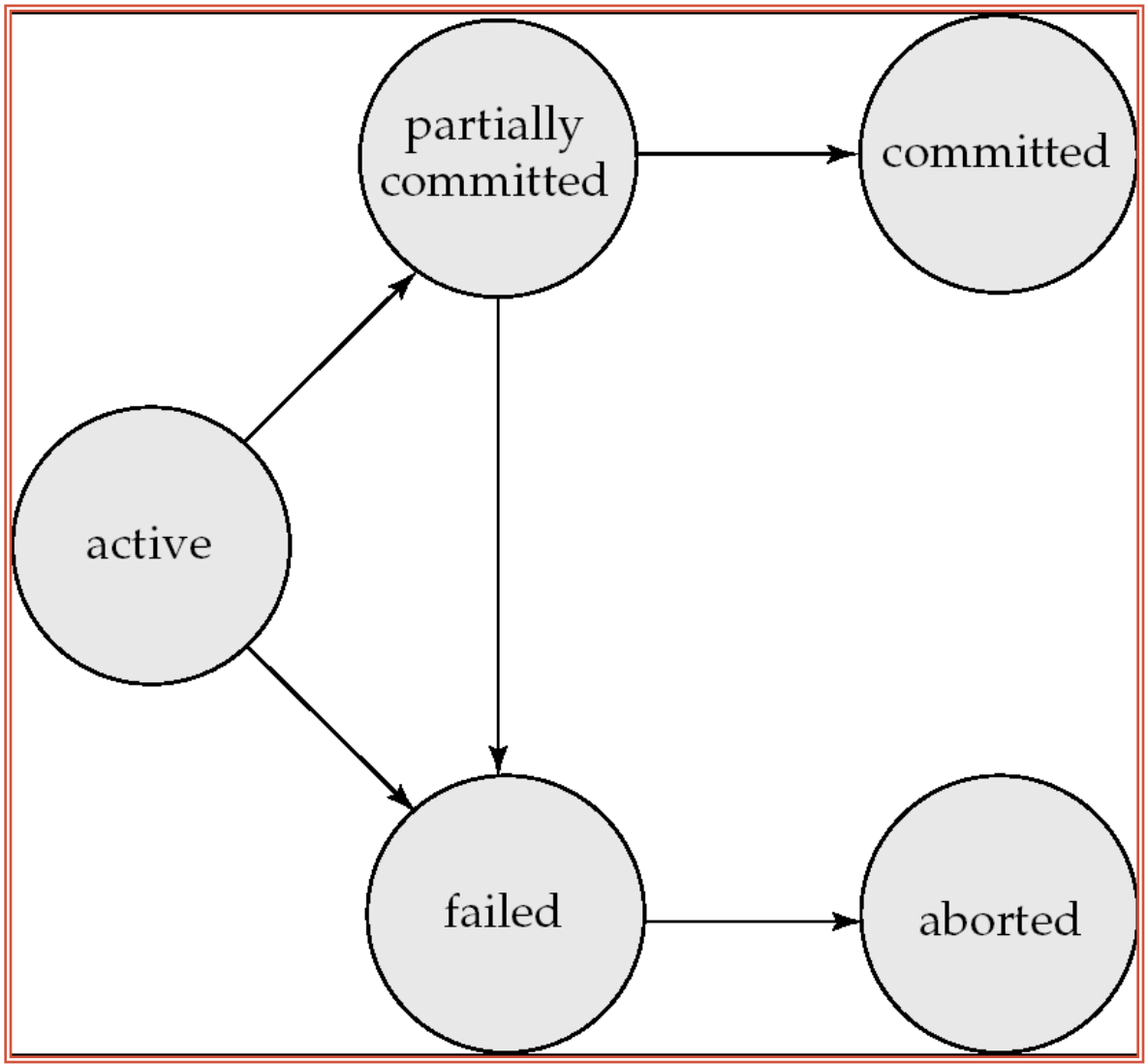
- **Active**
  - o estado inicial; a transação fica nesse estado enquanto estiver executando
- **Partially committed**
  - depois que o último comando for executado.
- **Failed**
  - depois da descoberta que a execução normal não pode prosseguir.

Estados de uma Transação

- **Aborted**
  - depois que a transação foi revertida e o banco foi restaurado ao estado anterior a execução.
  - Existem duas formas de prosseguir:
    - Reiniciar a transação
      - faz sentido somente se o “abort” não foi devido a erros lógicos
    - Matar a transação

- **Committed**
  - depois da execução bem sucedida.

Estados de uma Transação



Tópicos

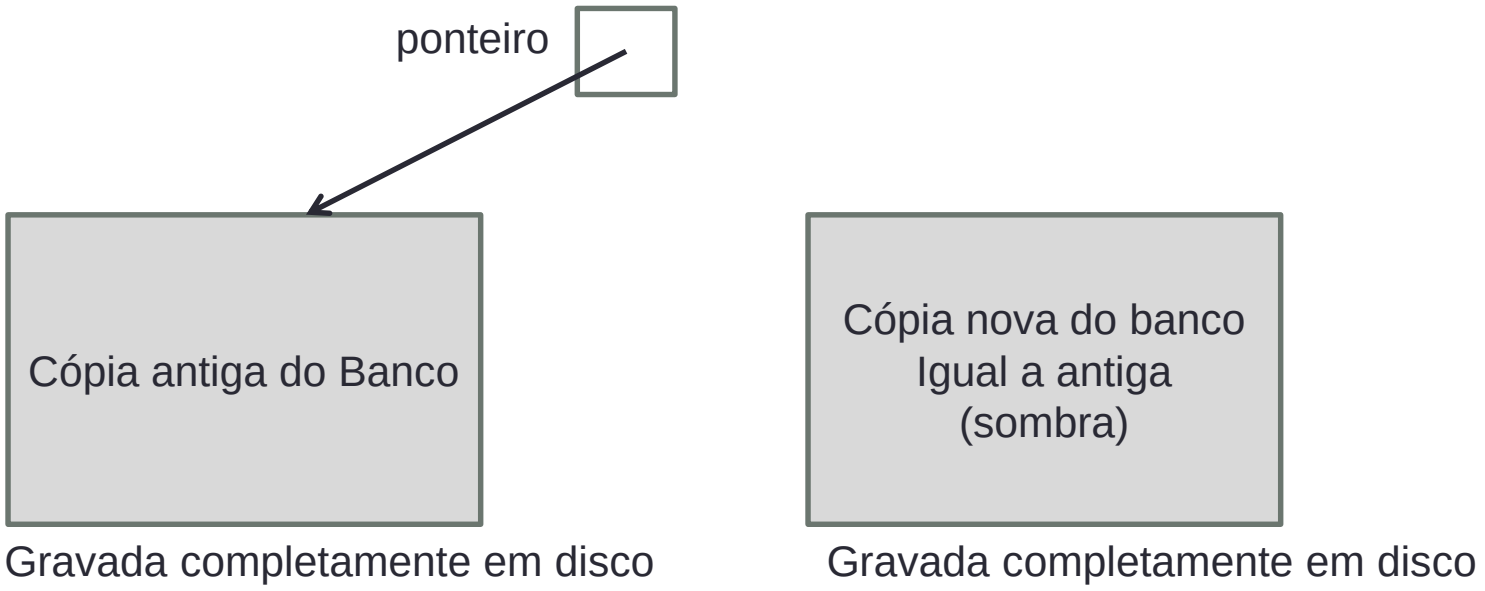
- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Cópia sombra

- Cópia Sombra (shadow-database)
- Trata-se de um modelo simples de garantia de atomicidade e durabilidade
  - Baseado na realização de alterações em uma cópia exata do banco de dados

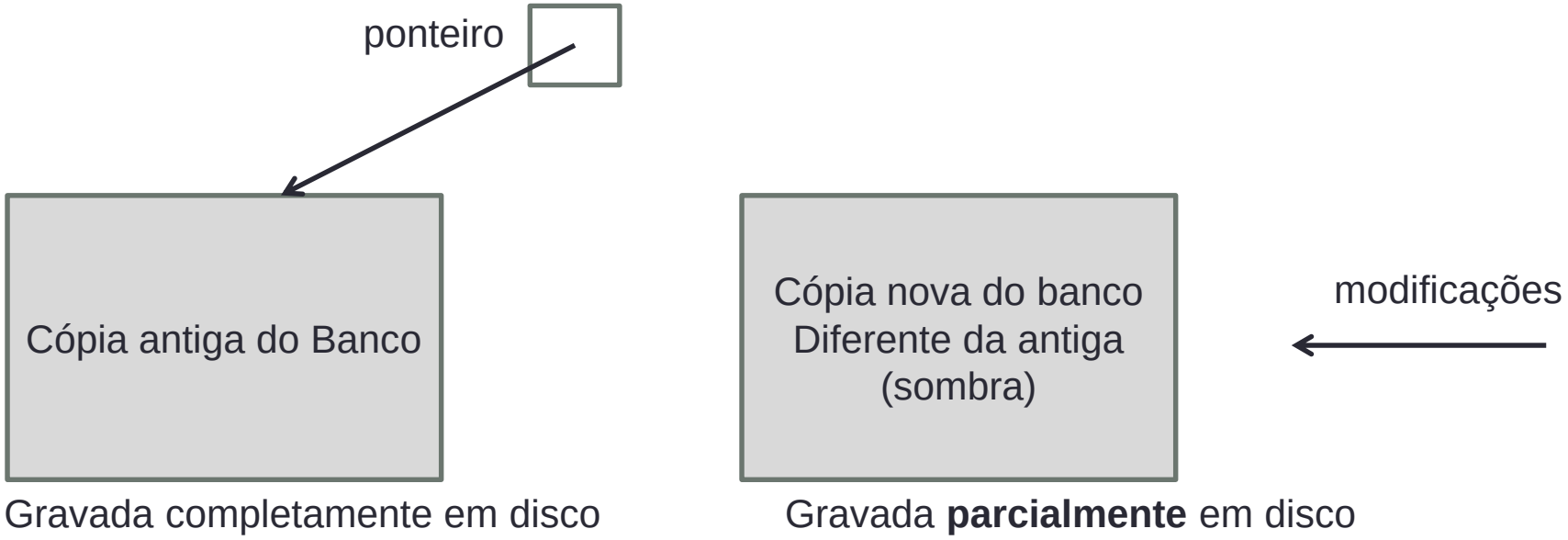
Esquema da Cópia Sombra

- Antes de iniciar a atualização, é criada uma cópia nova do banco (sombra)
- Ponteiro continua apontando para a cópia antiga



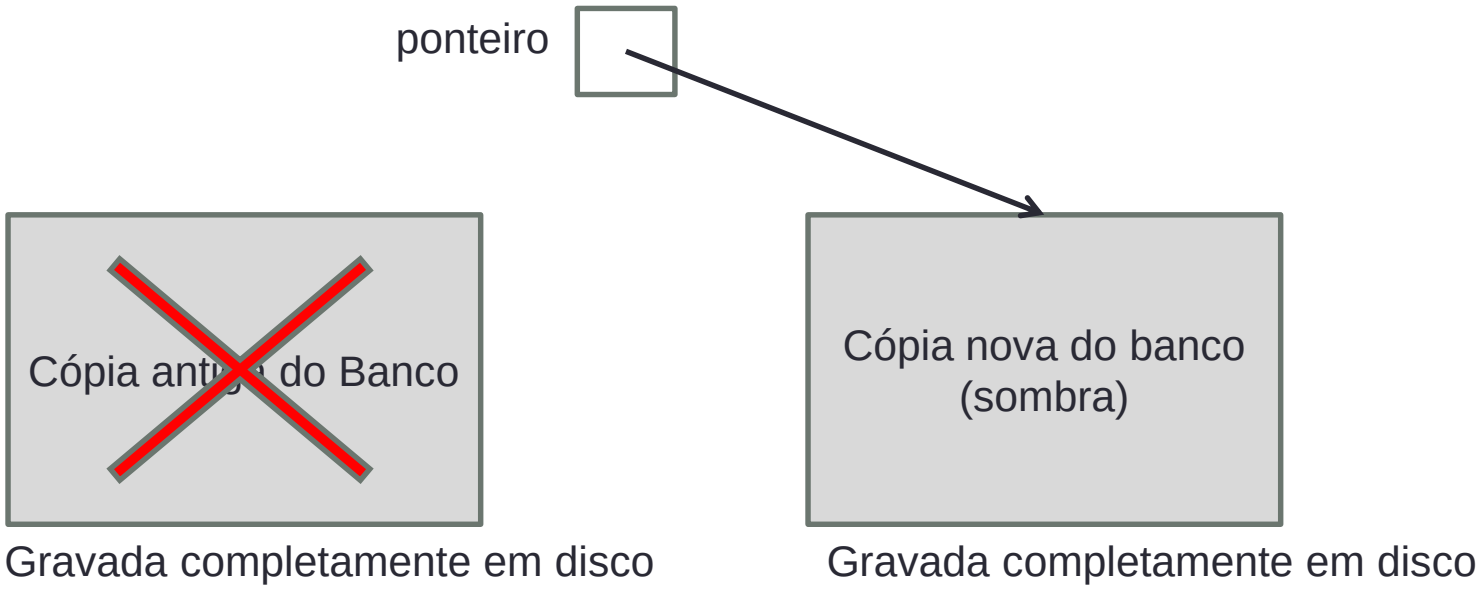
Esquema da Cópia Sombra

- Todas atualizações são feitas na cópia sombra



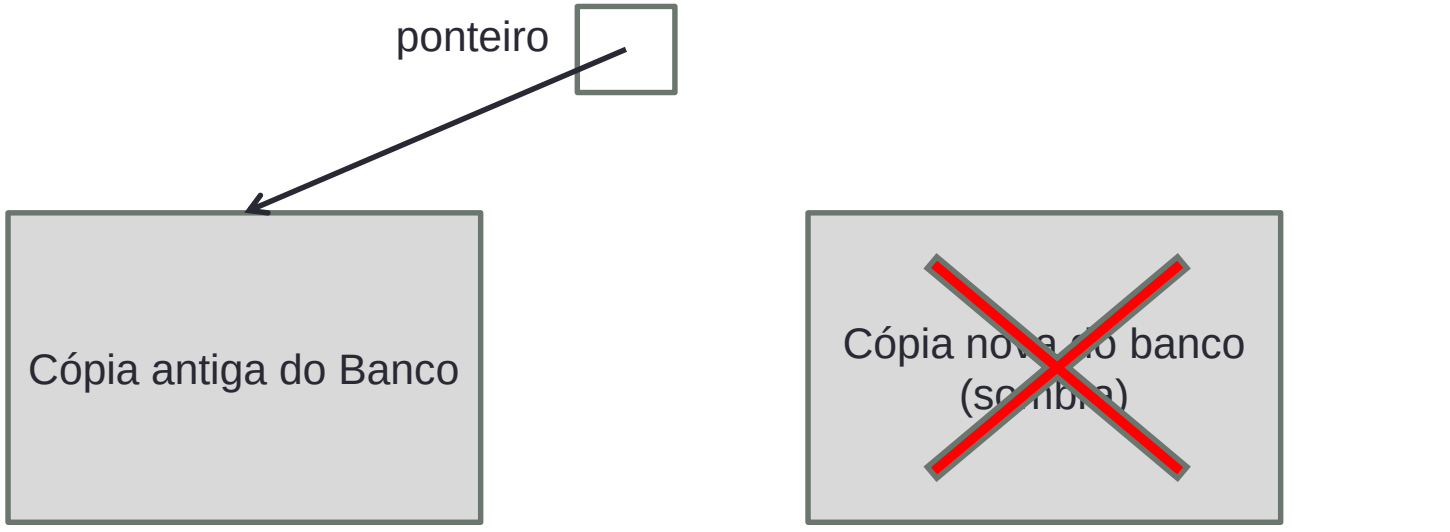
Esquema da Cópia Sombra

- O **ponteiro** aponta para a sombra depois que
  - A transação chegar ao partial commit e
  - Todas páginas de atualização forem descarregadas em disco
  - A cópia antiga é removida



Esquema da Cópia Sombra

- O ponteiro sempre aponta para um estado consistente do banco.
  - Caso a transação falhe, o ponteiro continuará apontando para a cópia antiga, e a sombra poderá ser removida.



Gravada completamente em disco

Gravada parcialmente em disco

Cópia Sombra

- O esquema da cópia sombra:
  - Assume que apenas uma transação está ativa.
    - Ou seja, não trata transações concorrentes
  - Útil para editores de texto, mas
    - Extremamente ineficiente para grandes bancos (porque?)
- Nas próximas aulas, veremos esquemas mais adequados
  - Atômicos
  - Duráveis
  - E sobretudo concorrentes

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Execuções Concorrentes

- Quais as vantagens em ter acessos concorrentes?
  - **Maior utilização do disco e do processador**, levando a uma maior vazão (*throughput)* das transações
    - Ex. Uma transação pode usar o CPU enquanto a outra está lendo ou gravando do disco
  - **Redução do tempo médio de resposta**: transações curtas não esperam mais tanto atrás de transações longas.

Execuções Concorrentes

- **Esquemas de controle de concorrência**
  - mecanismo que permite o isolamento
    - controla a interação entre as transações concorrentes
    - prevenindo que elas destruam a consistência do banco de dados

- Alguns esquemas serão vistos nas próximas aulas

- Para compreendê-los, é importante conhecer os seguintes conceitos
  - Schedule
  - Serialização

Schedules

- **Schedule** – uma sequência de instruções que especificam a ordem cronológica com que as instruções de transações concorrentes são executadas
  - Um schedule para um conjunto de transações deve
    - possuir todas as instruções dessas transações
    - preservar a ordem com que as instruções aparecem em cada transação.

n.	T1	T2
1	**read (A)**	
2	**A := A - 50**	
3	**write *(A)***	
4		**read(A)**
5		**read(B)**
6		**print(A + B)**
7	**read *(B)***	
8	***B := B + 50***	
9	**write *(B)***	

Schedules

- Uma transação que termina com sucesso terá a instrução de **commit** como a última instrução

T1	T2
read (A) **A := A - 50** **write *(A)***	
	read(A) **read(B)** **print(A + B)**
read *(B)*	
	commit
B := B + 50 **write *(B)*** ***commit***	

Schedules

- Uma transação que falha terá a instrução de **abort** como a última instrução

T1	T2
read (A) **A := A - 50** **write *(A)***	
	read(A) **read(B)** **abort**
read *(B)*	
B := B + 50 **write *(B)*** ***commit***	

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Schedules Equivalentes

- Diferentes schedules podem levar um banco de dados ao mesmo estado após a execução

- Esse schedules que geram o mesmo resultado são chamados de equivalentes

Schedules Equivalentes

- Considere as transações ao lado
- T_1 transfere \$50 de *A* para *B*
- T_2 transfere 10% do saldo de *A* para *B.*

T1
read (A) **A := A - 50** **write *(A)*** **read *(B)*** ***B := B + 50*** **write *(B)***

T2
read(A) **Temp:= A * 0.1** **A := A - temp** **write(A)** **read(B)** **B := B + temp** **write(B)**

Schedules Equivalentes

- O schedule ao lado é serial
- T_1 executa antes de T_2
  - <T1, T2>

T1	T2
read (A) **A := A - 50** **write *(A)*** **read *(B)*** ***B := B + 50*** **write *(B)*** ***commit***	**read(A)** **Temp:= A * 0.1** **A := A - temp** **write(A)** **read(B)** **B := B + temp** **write(B)** **commit**

Schedule 1

Schedules Equivalentes

- O schedule ao lado é serial
- T_2 executa antes de T_1
  - <T2, T1>

- Mas não é equivalente ao schedule 1
  - os valores das contas A e B mudam dependendo do schedule que for usado

T1	T2
	read(A) **Temp:= A * 0.1** **A := A - temp** **write(A)** **read(B)** **B := B + temp** **write(B)** **commit**
read (A) **A := A - 50** **write *(A)*** **read *(B)*** ***B := B + 50*** **write *(B)*** ***commit***	

Schedule 2

Schedules Equivalentes

- O schedule ao lado não é serial

- Mas é *equivalente* ao Schedule 1: <T1, T2>
  - Gera os mesmos resultados

T1	T2
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)*	
commit	
	read(B)
	B := B + temp
	write(B)
	commit

Schedule 3

Schedules Equivalentes

- O schedule ao lado não é serial

- E nem equivalente a algum dos schedules seriais
  - <T1,T2>
  - <T2,T1>

T1	T2
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)*	
commit	
	B := B + temp
	write(B)
	commit

Schedule 4

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Serialização de Schedules

- **Presunção Básica** – Cada transação por si só preserva consistência.
  - Então a execução serial de um conjunto de transação também preserva consistência.

- Um schedule concorrente é serializável se ele for **equivalente** a um schedule serial.
  - Isso significa que ele também preservará a consistência do banco de dados

Serialização de Schedules

- O schedule ao lado é serial
- T_1 executa antes de T_2
  - <T1, T2>
- Como o schedule é serial, ele também é consistente
  - O valor retirado de uma conta equivale ao valor depositado na outra conta

T1	T2
read (A) **A := A - 50** **write *(A)*** **read *(B)*** ***B := B + 50*** **write *(B)*** ***commit***	**read(A)** **Temp:= A * 0.1** **A := A - temp** **write(A)** **read(B)** **B := B + temp** **write(B)** **commit**

Schedule 1

Serialização de Schedules

- O schedule ao lado é serial
- T_2 executa antes de T_1
  - <T2, T1>

- Como o schedule é serial, ele também é consistente
  - O valor retirado de uma conta é todo ele depositado na outra conta

T1	T2
	read(A) **Temp:= A * 0.1** **A := A - temp** **write(A)** **read(B)** **B := B + temp** **write(B)** **commit**
read (A) **A := A - 50** **write *(A)*** **read *(B)*** ***B := B + 50*** **write *(B)*** ***commit***	

Schedule 2

Serialização de Schedules

- O schedule ao lado não é serial
- Mas é *equivalente* ao Schedule 1.
  - Gera os mesmos resultados
- Ou seja, ele é serializável
  - O que implica em ser também consistente

T1	T2
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)*	
commit	
	read(B)
	B := B + temp
	write(B)
	commit

Schedule 3

Serialização de Schedules

- O schedule ao lado não é serial
- E nem equivalente a algum dos schedules seriais
  - <T1,T2>
  - <T2, T1>
- Ou seja, não é serializável
- Tampouco é consistente
  - o valor retirado de A não bate com o valor depositado em B

T1	T2
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)*	
commit	
	B := B + temp
	write(B)
	commit

Schedule 4

Tópicos

- Conceito de Transação
- Propriedades ACID
- Estados de uma transação
- Cópia Sombra
- Conceito de Schedule
- Schedules Equivalentes
- Serialização de schedules
- Serialização em conflito

Serialização em Conflito

- Dado um schedule concorrente, como saber se ele é equivalente a um schedule serial?
  - Sem precisar analisar todas as instruções que fazem parte de cada transação?

- Uma das formas de fazer isso é através da técnica chamada de serialização em conflito

Serialização em Conflito

- Simplificando as transações
  - Ignora-se instruções que não sejam **read** e **write**
  - Assume-se que cálculos intermediários sejam feitos usando a memória principal.
  - Por isso os schedules possuirão apenas instruções de **read** e **write**.

Serialização em Conflito

- Instruções I_x e I_j das transações T_x e T_y respectivamente, **conflitam**
  - se existir algum item Q acessado tanto por I_x quanto por I_y
  - e pelo menos uma dessas instruções escreve Q.

Instrução de Tx	Instrução de Ty	resultado
read(Q)	read(Q)	sem conflito
read(Q)	write(Q)	conflito
write(Q)	read(Q)	conflito
write(Q)	write(Q)	conflito

Serialização em Conflito

Instrução de Tx	Instrução de Ty	resultado
read(Q)	read(Q)	sem conflito
read(Q)	write(Q)	conflito
write(Q)	read(Q)	conflito
write(Q)	write(Q)	conflito

- Intuitivamente, um conflito entre duas instruções força uma ordem temporal entre elas.
  - Se as instruções são consecutivas em um schedule e não conflitam
    - Elas poderiam ser trocados de lugar no schedule
    - E ainda assim seus resultados seriam iguais

Serialização em Conflito

- Se um schedule *S* pode ser transformado em um schedule *S´ por uma série de* trocas de instruções não conflitantes
  - dizemos que *S* e *S´* são **equivalentes em conflito**.

- Se um schedule S for equivalente em conflito a um schedule serial
  - dizemos que ele é **serializável em conflito**

Serialização em Conflito

- O schedule A pode ser transformado no schedule serial B, por uma série de trocas de instruções não conflitantes.
  - Então, o schedule A é serializável em conflito.

T1	T2
read (A)	
write *(A)*	
	read(A)
	write(A)
read *(B)*	
write *(B)*	
Commit	
	read(B)
	write(B)
	Commit

Schedule A

T1	T2
read (A)	
write *(A)*	
read *(B)*	
write *(B)*	
Commit	
	read(A)
	write(A)
	read(B)
	write(B)
	Commit

Schedule B

Serialização em Conflito

- Exemplo de um schedule que não é serializável em conflito:

T3	T4
read (Q)	
	write(Q)
	Commit
write *(Q)*	
Commit	

- Não se pode trocar instruções desse schedule para obter o schedule serial < T_3, T_4 > ou o schedule serial < T_4, T_3 >.