# TRANSAÇÕES
# - SERIALIZAÇÃO DE VISÃO
# - REQUISITOS DE SCHEDULES

Sérgio Mergen

**Versão modificada de Database System Concepts, 5th Ed.**

# Tópicos

- Serialização em visão
- Testando Serialização
- Protocolos de Controle de Concorrência
  - Schedules concorrentes
  - Schedules recuperáveis
  - Schedules livres de rollback em cascata

# Serialização em Visão

- *S* e *S´* são dois schedules contendo as mesmas transações.

- *Para S* e *S´* serem **equivalentes em visão***:*

- Se em um schedule uma transação $T_x$ executou um **read**(*Q)*
  - de um valor produzido por um write de outra transação $T_y$

- No outro schedule a transação $T_x$ também deve ler o mesmo valor de *Q*
  - que foi produzido pelo mesmo **write**(Q) da transação $T_y$ .

# Serialização em Visão

- *S* e *S´* são dois schedules contendo as mesmas transações.

- *Para S* e *S´* serem **equivalentes em visão***:*

- A transação que executa a instrução final **write**(*Q*) do schedule *S*
- também deve executar a instrução final **write**(*Q*) do schedule *S'.*

# Serialização em Visão

- Um schedule *S* é **serializável em visão** *se for equivalente em visão* a um schedule serial.
- O schedule ao lado é serializável em visão.
- A qual schedule serial ele é equivalente?

| T3 | T4 | T6 |
|---|---|---|
| **read (Q)** | | |
| | **write(Q)** | |
| | **Commit** | |
| **write *(Q)*** | | |
| ***Commit*** | | |
| | | **write(Q)** |
| | | **Commit** |

# Serialização em Visão

- Todo schedule serializável em conflito é também serializável em visão.
- Porém, nem todo schedule serializável em visão é também serializável em conflito.
- O schedule ao lado é serializável em visão, mas não é serializável em conflito.

| T3 | T4 | T6 |
|---|---|---|
| **read (Q)** | | |
| | **write(Q)** | |
| | **Commit** | |
| **write *(Q)*** | | |
| ***Commit*** | | |
| | | **write(Q)** |
| | | **Commit** |

# Serialização em Visão

- Perceba que é como se T4 nem executasse
- Sua instrução de escrita é descartada por uma escrita posterior de outra transação
- A isso é chamado **escrita cega**
- Todo schedule serializável em visão que não for serializável em conflito possui **escritas cegas.**

| T3 | T4 | T6 |
|---|---|---|
| **read (Q)** | | |
| | **write(Q)** | |
| | **Commit** | |
| **write *(Q)*** | | |
| ***Commit*** | | |
| | | **write(Q)** |
| | | **Commit** |

# Outras noções sobre Serialização

- O schedule ao lado produz o mesmo resultado que o schedule serial < $T_1$, $T_5$ >
  - mas não é serializável em conflito ou visão

| T1 | T5 |
|---|---|
| **read (A)** | |
| **A := A - 50** | |
| **write(A)** | |
| | **read (B)** |
| | **B := B - 10** |
| | **write(B)** |
| **Read (B)** | |
| **B := B + 50** | |
| **write *(B)*** | |
| ***Commit*** | |
| | **read(A)** |
| | **A := A + 10** |
| | **write *(A)*** |
| | ***Commit*** |

- Determinar essa equivalência exigiria analisar também instruções além das de read e write.

# Tópicos

- Serialização em visão
- Testando Serialização
- Protocolos de Controle de Concorrência
  - Schedules concorrentes
  - Schedules recuperáveis
  - Schedules livres de rollback em cascata

# Testando Serialização

- Pode-se verificar se um schedule é serializável em conflito usando um **grafo de precedência**
  - um grafo direto onde os vértices são as transações.
  - e as arestas são conflitos entre as transações

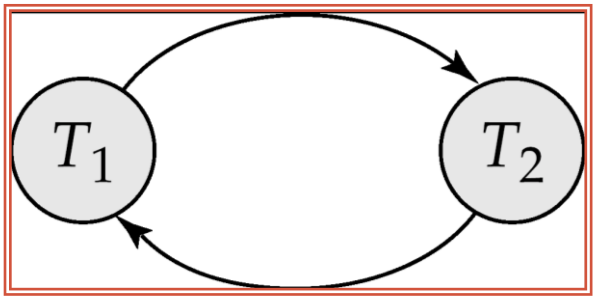
# Testando Serialização

- Considere um schedule de um conjunto de transações $T_1$, $T_2$, ..., $T_n$

- É desenhado um arco de $T_x$ para $T_y$
  - *se as duas transações* conflitarem,
  - e $T_x$ acessou antes o item onde ocorreu o conflito.

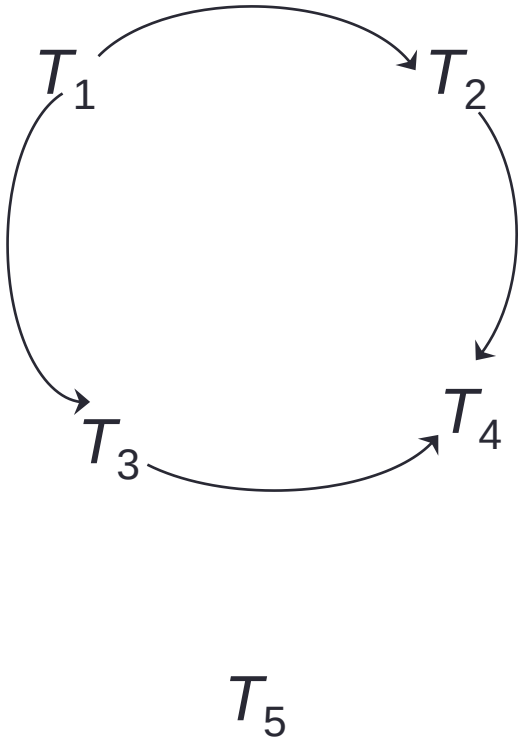
- A aresta pode ser rotulada com o nome do item de dado acessado.

- **Exemplo 1**



# Exemplo

| T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|
| **write(A)** | **read (X)** | | | |
| **read(Z)** | | | | **read(*V*)** |
| | | | | **read(*W*)** |
| | **read(A)** | | | |
| | **write(*Y*)** | | | |
| | | **write(*Z*)** | | |
| **read (U)** | | | | |
| | | | **read(*Y*)** | |
| | | | **write(*Y*)** | |
| | | | **read(*Z*)** | |
| **read(U)** | | | | |
| **write(U)** | | | | |



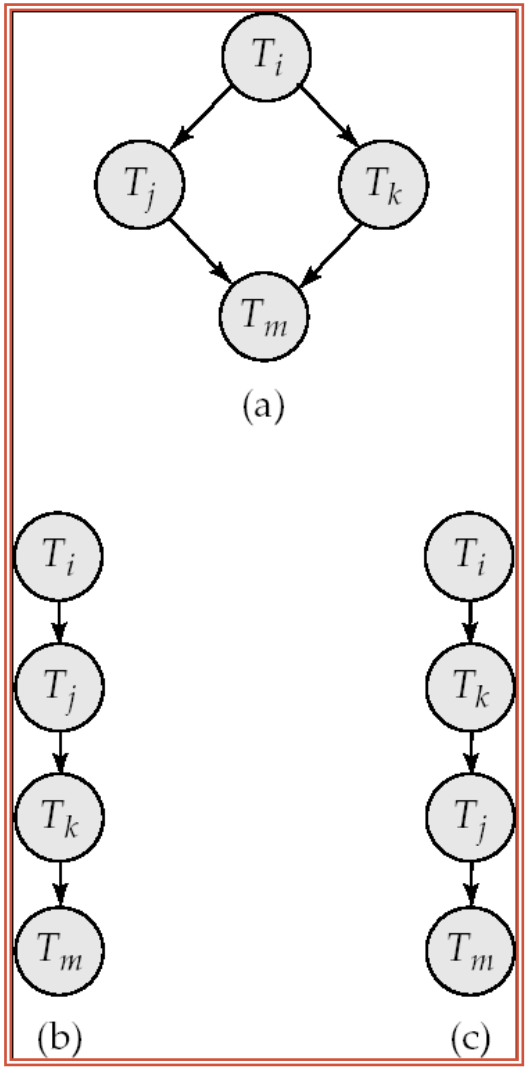
# Teste para Serialização em Conflito

- Um schedule é serializável em conflito se e somente se seu grafo de precedência for acíclico.

- Existem algoritmos de detecção de ciclo de complexidade linear O(n+e)
  - *n* é o número de vértices
  - e é o número de arestas.

# Teste para Serialização em Conflito

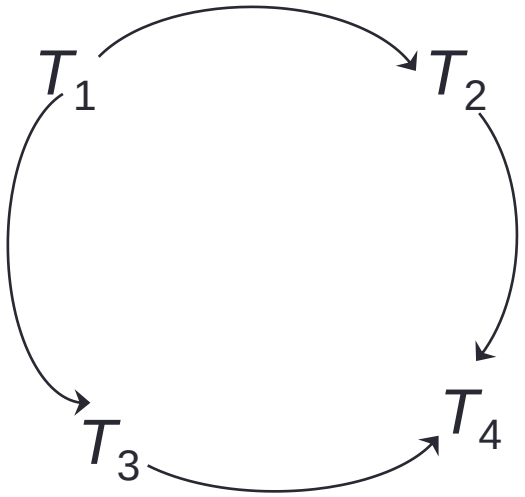
- Se o grafo de precedência for acíclico, a ordem de serialização pode ser obtida por uma *ordenação topológica* do grafo.
  - Essa é uma ordem linear consistente com a ordem parcial do grafo.



# Teste para Serialização em Conflito

- Uma possível ordenação topológica do grafo ao lado seria
  $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

- Existem outras?




$T_5$

# Tópicos

- Serialização em visão
- Testando Serialização
- Protocolos de Controle de Concorrência
  - Schedules concorrentes
  - Schedules recuperáveis
  - Schedules livres de rollback em cascata

# Protocolos de Controle de Concorrência

- Quando as aplicações enviam comandos SQL para o SGBD, esses comandos devem obter acesso aos dados
  - Esses comandos podem ser vistos como instruções de READ e WRITE

- Ex. considere que a conta bancária 890-01 seja identificada pelo apelido A
  - o comando *SELECT * FROM conta WHERE numConta = '890-01'*
    - seria traduzido para *READ(A)*

# Protocolos de Controle de Concorrência

- Quando muitas instruções de READ e WRITE desejam acessar os itens de dados, como controlar o acesso a esses itens?

- O mecanismo usado para isso é chamado de protocolo de controle de concorrência
  - A ordem com que o acesso é concedido vem a se tornar o schedule

# Protocolos de Controle de Concorrência

- O objetivo de um protocolo de controle de concorrência é gerar schedules
  - Concorrentes (e consistentes)
  - Recuperáveis
  - Sem rollbacks em  cascata .

# Schedules Concorrentes

- Lembrando:
  - as transações concorrentes disputam o uso do processador de forma intercalada

- Vantagens em ter schedules 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.

# Protocolos de Controle de Concorrência

- O objetivo de um protocolo de controle de concorrência é gerar schedules
  - Concorrentes (e consistentes)
  - Recuperáveis
  - Sem rollbacks em cascata .

# Schedules Recuperáveis

- **Schedule recuperável** — se uma transação ler um item de dados previamente escrito por outra transação
  - o commit da transação que fez a escrita deve aparecer antes do commit daquela que fez a leitura

- O schedule ao lado não é recuperável

| T8 | T9 |
|---|---|
| **read (A)**<br>**write(A)**<br><br><br><br>**read (B)**<br>***Commit*** | <br><br><br>**read(A)**<br>***Commit*** |

# Schedules Recuperáveis

- **Schedule recuperável** — se uma transação ler um item de dados previamente escrito por outra transação
  - o commit da transação que fez a escrita deve aparecer antes do commit daquela que fez a leitura

- Se $T_8$ abortar, $T_9$ teria lido (e possivelmente mostrado ao usuário) um banco inconsistente.

É imprescindível que o SGBD gere schedules recuperáveis.

| T8 | T9 |
|---|---|
| **read (A)** | |
| **write(A)** | |
| | **read(A)** |
| | ***Commit*** |
| **read (B)** | |
| ***Abort*** | |

# Protocolos de Controle de Concorrência

- O objetivo de um protocolo de controle de concorrência é gerar schedules
  - Concorrentes (e consistentes)
  - Recuperáveis
  - Sem rollbacks em cascata .

# Schedules sem rollback em cascata

- Rollback é a ação que é tomada quando uma transação aborta
  - Significa que uma transação precisa ser desfeita (revertida)
  - Isso pode vir a ser necessário para assegurar o requisito de atomicidade
    - Ou tudo é executado ou nada é

- Rollbacks em cascata
  - Ocorre quando o abort em uma transação leva a um abort de outra transação
  - que leva ao abort de outra transação
  - e assim por diante
  - gerando um efeito de rollbacks em cascata

# Schedules sem rollback em cascata

- No schedule ao lado nenhuma das transações “comitou” ainda
  - ou seja, o schedule ainda é recuperável

| T10 | T11 | T12 |
|---|---|---|
| **read (A)** | | |
| **read (B)** | | |
| | **read(B)** | |
| **write *(A)*** | | |
| | **read(A)** | |
| | **write(A)** | |
| | | **read(A)** |

# Schedules sem rollback em cascata

- Se $T_{10}$ falhar, $T_{11}$ e $T_{12}$ também devem ser revertidas (roll back).
  - Para garantir a consistência do banco
- Pode levar a reversão de uma quantidade grande de trabalho
- E a reversão tem um custo relativamente alto

| T10 | T11 | T12 |
|---|---|---|
| **read (A)** | | |
| **read (B)** | | |
| | **read(B)** | |
| **write *(A)*** | | |
| | **read(A)** | |
| | **write(A)** | |
| | | **read(A)** |
| ***Abort*** | | |
| | ***Abort (cascata)*** | |
| | | ***Abort (cascata)*** |

# Schedules sem rollback em cascata

- **Schedules sem cascata**
  - Se uma transação ler um item de dados previamente escrito por outra transação
    - A transação que fez a escrita deve comitar antes que a outra transação tenha feito a leitura

- Schedules sem cascata previnem rollbacks em cascata
  - É desejável restringir os schedules para os que são sem cascata

# Schedules sem rollback em cascata

- O schedule ao lado é livre de cascata
- E também é recuperável
- Todo schedule sem cascata é também recuperável

| T10 | T11 | T12 |
|---|---|---|
| **read (A)** | | |
| **read (B)** | | |
| | **read(B)** | |
| **write *(A)*** | | |
| ***Commit*** | | |
| | **read(A)** | |
| | **write(A)** | |
| | ***Commit*** | |
| | | **read(A)** |
| | | **.** |
| | | **.** |

# Finalizando

- Como vimos, um SGBD deve ter um mecanismo que assegura que todos os schedules possíveis sejam
  - Serializáveis em conflito ou visão, e
  - Recuperáveis e preferencialmente sem cascata

- A estratégia que permite apenas schedules seriais leva a um grau pobre de concorrência
  - Ou seja, schedules concorrentes são desejáveis

# Finalizando

- O teste de serialização de um schedule é inviável
  - Para testar, é necessário conhecer todas instruções das transações
  - Para saber qual instrução de uma transação vem a seguir, o gerenciador deve executar a instrução anterior.
  - Para conhecer todas as instruções de uma transação, o gerenciador deve executar todas as instruções
  - Testar se um schedule é serializável *depois* de ele ter sido executado é tarde demais!
- Como resolver esse problema?

# Finalizando

- De modo geral, os protocolos de controle de concorrência analisam uma instrução de cada vez de cada instrução

- Os protocolos impõem regras que evitam schedules não serializáveis.
  - como veremos na próxima aula.