Introdução aos Conceitos e Teoria de Processamento de Transações

André Luís Schwerz [1] Rafael Liberato [1]

[1]Universidade Tecnológica Federal do Paraná
Departamento de Computação

Banco de Dados 2
2020/1

Tópicos

Tópicos

Sistemas de processamento de transação

- Definição:
  - Sistemas com grandes bancos de dados
  - Centenas de usuários simultâneos que executam transações de banco de dados
  - Exigem:
    - Alta disponibilidade
    - Tempo de resposta rápido para centenas de usuários
- Exemplos:
  - Reservas de passagens aéreas
  - Sistemas bancários
  - Processamento de cartão de crédito
  - Compras on-line
  - Mercados de ações
  - Caixas de supermercados

Sistemas de Monousuário vs Multiusuário

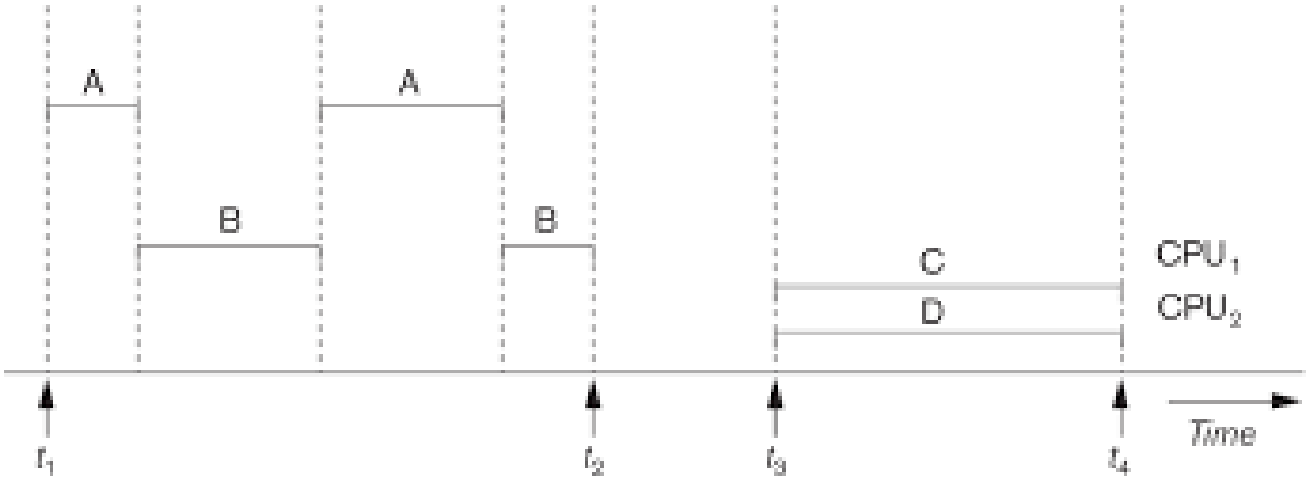
- Monousuário
  - Apenas um usuário por vez pode utilizar o sistema.
- Multiusuário
  - Muitos usuários podem acessar o banco de dados simultaneamente.

Monousuário

Multiusuário

Multiprogramação

- Multiprogramação
  - O sistema operacional executa vários programas (processos) simultaneamente.
  - Processamento intercalado vs processamento paralelo



Tópicos

Transação

- O que é uma Transação?
  - É um programa em execução que forma uma unidade lógica de processamento de banco de dados.
- Uma transação pode incluir uma ou mais operações de acesso ao banco de dados
  - Inserção, exclusão, modificação ou recuperação
  - Pode também ser embutida dentro de um programa
- Limites de uma transação devem ser explícitos:
  - **BEGIN_TRANSACTION**
  - **END_TRANSACTION**
- Um programa pode conter várias transações separadas pelos limites **BEGIN_TRANSACTION** e **END_TRANSACTION**.

Item de Dados

- Um banco de dados é formado por uma coleção de **itens de dados**.
- **Granularidade** de um item de dado pode ser:
  - Um atributo
  - Uma tupla
  - Um bloco de disco
- Conceitos de transações são apresentados **independentemente** da granularidade.

Operações sobre os itens de dados

- Há duas operações básicas de acesso ao banco de dados.
- Mais baixo nível semântico.
- **read(x)**:
  - Lê um item do banco de dados chamado `x` para uma variável do programa.
- **write(x)**:
  - Grava o valor da variável de programa `x` no item de banco de dados chamado `x`.

Operação de leitura

- **read(x)** inclui as seguinte etapas:
  - Encontrar o endereço do bloco de disco que contém o item `x`.
  - Copie esse bloco de disco para um *buffer* na memória principal (caso já não exista).
  - Copie o item `x` do *buffer* para a variável de programa chamada `x`.

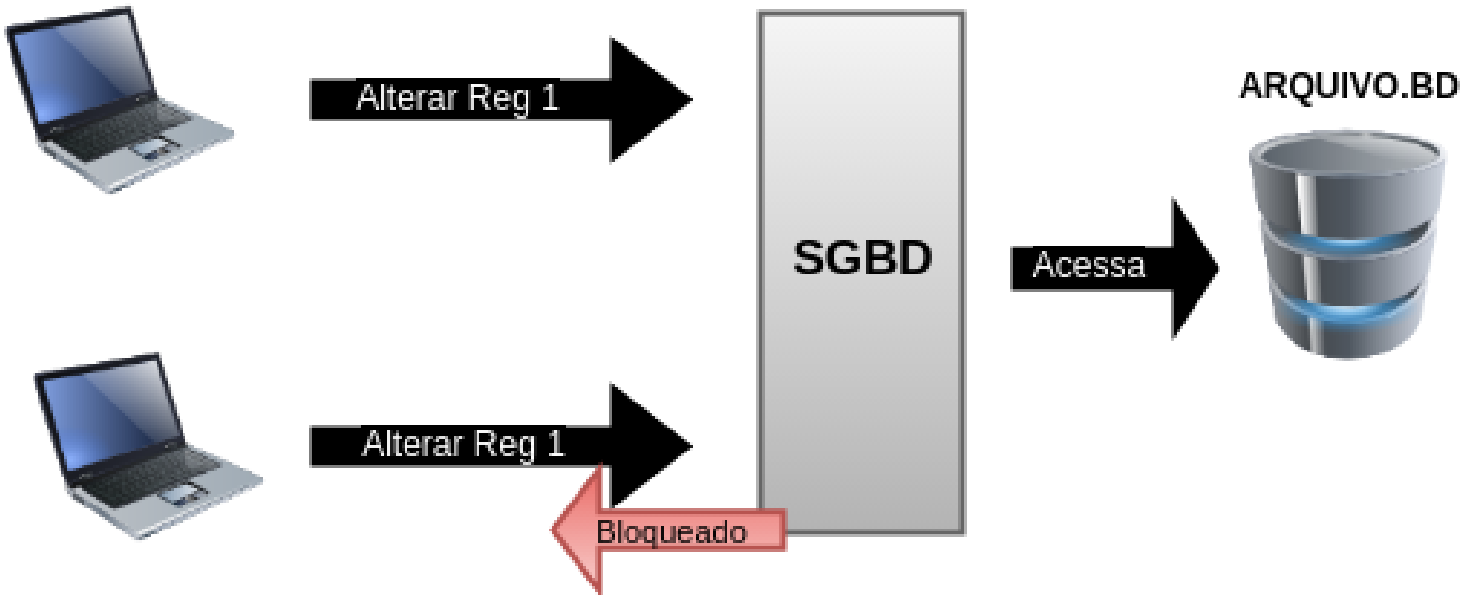
Operação de escrita

- **write(x)** inclui as seguinte etapas:
  - Encontrar o endereço do bloco de disco que contém o item `x`.
  - Copie esse bloco de disco para um *buffer* na memória principal (caso já não exista).
  - Copie o item `x` da variável de programa chamada `x` para o local correto no *buffer*.
  - Armazene o bloco atualizado do *buffer* de volta no disco.

Tópicos

Controle de concorrência

- **Por que o controle de concorrência é necessário?**



Anomalias

- Problema de **atualização perdida** (*lost update*):
  - Esse problema ocorre quando duas transações, que acessam os mesmos itens de dados, têm suas operações intercaladas de modo que isso torna o valor de alguns itens do dados incorreto.

T_1	T_2
read(x)	
x = x - n	
	read(x)
	x = x + m
write(x)	
read(y)	
	write(x)
y = y+ n	
write(y)	

O item x tem um valor incorreto porque sua atualização de T_1 é perdida (sobrescrita).

Anomalias

- Problema de **atualização temporária** (*dirty read*):
  - Esse problema ocorre quando uma transação atualiza um item de dado e depois a transação falha por algum motivo.
  - Problema também conhecido como leitura suja.

T_1	T_2
read(x)	
x = x - n	
write(x)	
	read(x)
	x = x + m
	write(x)
read(y)	
abort()	

A transação T_1 falha e precisa mudar o valor de x de volta a seu valor antigo; enquanto isso, T_2 leu o valor temporário e incorreto de x

Anomalias

- Problema da **leitura não repetitiva** (*unrepeatable read*)
  - Quando uma transação lê duas vezes o mesmo item de dado e os valores são diferentes devido um outra transação ter realizado uma atualização nesse item.

T_1	T_2
read(x)	
x = x + w	
	read(x)
	x = x + m
	write(x)
read(x)	

A transação T_1 lê o mesmo item duas vezes e o item é alterado por outra transação T_2 entre as duas leituras. Logo, T_1 recebe dois valores diferentes para suas leituras do mesmo item.

Anomalias

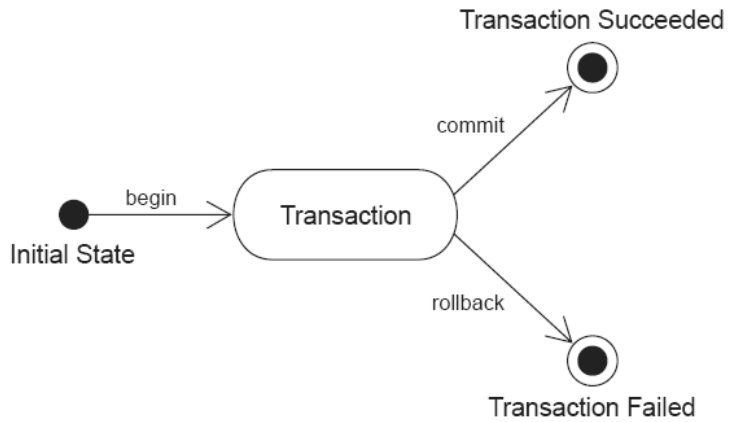
- Problema do **resumo incorreto** (*incorrect summary*)
  - Ocorre com funções de agregação `sum`, `avg`, `count`, `max` e `min`

T_1	T_2
read(x)	
x = x - n	
write(x)	
	read(x)
	s = s + x
	read(y)
	s = s + y
read(y)	
y = y + n	
write(y)	

A transação T_2 lê depois que n é subtraído e lê y antes que n seja somado; um resumo errado é o resultado (defasado por n).

Controle de Concorrência

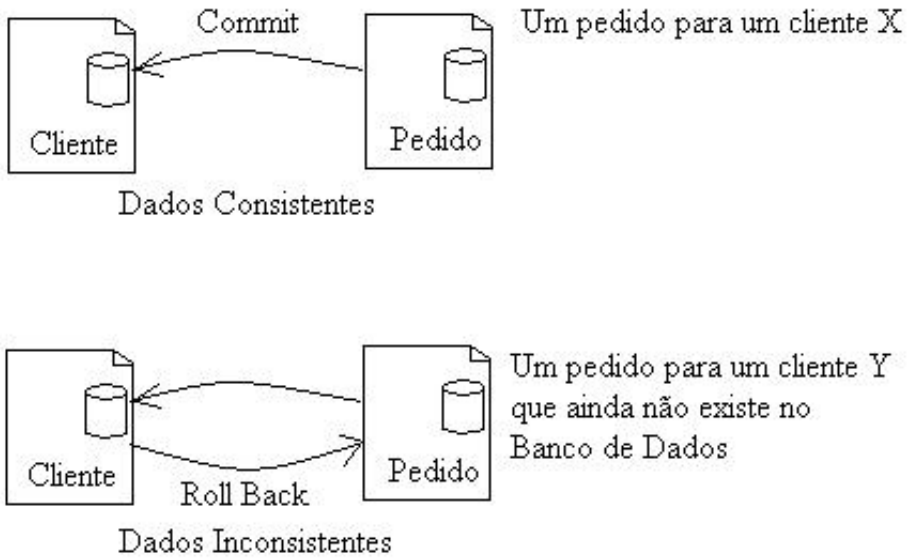
- Regra de ouro:
  - O SGBD NÃO deve permitir que algumas operações de uma transação T sejam aplicadas ao banco de dados enquanto que outras operações de T não o são, pois a transação inteira é uma unidade logica de processamento de banco de dados.




```
read(a);
a := a - 50;
write(a);
read(b);
b := b + 50;
write(b);
```

Recuperação

- Por que a recuperação é necessária?



Recuperação - Falhas

- Possíveis motivos para uma falha:
  - Erro da transação ou do sistema:
    - Erro em operações como estouro de inteiros ou divisão por zero.
    - Interrupção da execução pelo usuário.
  - Erros locais ou condições de execução detectadas pela transação:
    - Situações que requerem o cancelamento da transação.
    - Por exemplo: saldo insuficiente para saque em uma conta bancária.
  - Imposição de controle de concorrência:
    - Abortos por da violação da serialização, ou para resolver um estado de *deadlock*.
  - Problemas físicos e catástrofes:
    - Se refere a uma lista infinita de problemas que incluem falhas de energia, roubo, sabotagem, refrigeração e falhas de discos.

Tópicos

Conceitos de Transação

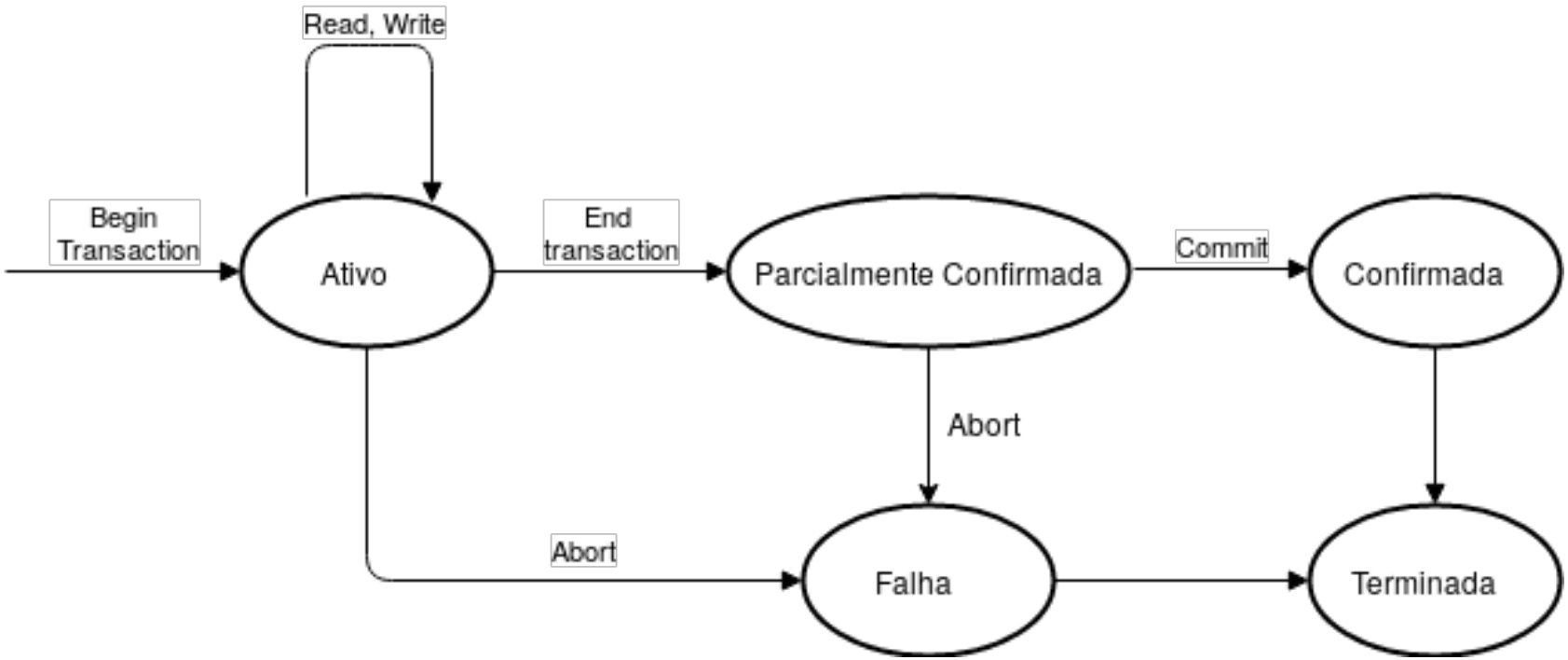
- Para fins de recuperação, o sistema precisa registrar quando cada transação começa, termina, confirma ou aborta.
- Portanto, o gerenciador de recuperação SGBD precisa acompanhar as seguintes operações:
  - **BEGIN_TRANSACTION**
    - Marca o inicio da transação ou execução.
  - **READ ou WRITE**
    - Especificam operações de leitura ou escrita.
  - **END_TRANSACTION**
    - Especifica que operações READ ou WRITE terminaram e marca o final da transação, e pode ser necessário também verificar se as mudanças podem ser aplicadas permanentemente ou se precisam ser abortadas.

Conceitos de Transação

- (continuação:)
  - **COMMIT_TRANSACTION**
    - Sinaliza um final bem sucedido da transação.
  - **ROLLBACK (ou ABORT)**
    - Sinaliza que a transação encerrada sem sucesso, e que qualquer mudança feita no banco deve ser desfeita.

Conceitos de Transação

- Estado de transações



Tópicos

Registros de log

- $[start\_transaction, T]$
  - Indica que a transação T iniciou sua execução
- $[write\_item, T, x, valor\_antigo, valor\_novo]$
  - Indica que a transação T mudou o valor do item x do banco de dados valor_antigo para valor_novo.
- $[read\_item, T, x]$
  - Indica que a transação T leu o valor do item x no banco de dados.
- $[commit, T]$
  - Indica que a transação T foi concluída com sucesso e indica que seu efeito pode ser confirmado no banco de dados.
- $[abort, T]$
  - Indica que a transação T foi abortada.

Ponto de confirmação (commit)

- Uma transação T alcança seu **ponto de confirmação** quando todas as suas operações, que acessam o banco de dados, tiverem sido executadas com sucesso e os seus efeitos tiverem sido registrados no log.

Tópicos

Propriedades ACID

- **Atomicidade**
  - Uma transação deve ser realizada em sua totalidade ou não ser realizada de forma alguma.
- **Consistência**
  - Uma transação deve levar o banco de dados de um estado consistente para outro estado consistente, sendo executado, do início ao fim, sem interferência de outras transações.
- **Isolamento**
  - A transação não deve ser interferida por quaisquer outras transações que aconteçam simultaneamente.
- **Durabilidade**
  - Mudanças aplicadas ao banco de dados pela transação confirmada precisam persistirem no banco de dados, não podendo serem perdidas por falhas.

Tópicos

Schedules de transações

- A **ordem da execução das operações** de todas as diversas transações que estão sendo executadas simultaneamente é conhecida como *schedule* (ou histórico).
- Um *schedule* S de n transações T_1, T_2, ..., T_n é uma ordenação das operações das transações.
  - Para cada transação T_i que participa no *schedule* S, as operações de T_i em S precisam aparecer na mesma ordem que ocorrem em T_i.

Schedules de transações

- Uma notação abreviada:

Sigla	Operação
b	begin_transaction
r	read_item
w	write_item
e	end_transaction
c	commit
a	abort

Schedules de transações

- Exemplo:

$$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); r_1(Y);$$

T_1	T_2
read(X)	
X = X - N	
	read(X)
	X = X + M
write(X)	
read(Y)	
	write(X)
read(Y)	

Schedules de transações

- Exemplo:

$$S_b : r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

T_1	T_2
read(X)	
X = X - N	
write(X)	
	read(X)
	X = X + M
	write(X)
read(Y)	
abort()	

Schedules recuperáveis

Schedules recuperáveis

Um *schedule* S é recuperável se nenhuma transação T em S for concluída até que todas as transações T' que gravaram dados lidos por T tenham sido concluídas.

Schedules recuperáveis

Não recuperável

T_1	T_2
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 10
	write(A)
	commit()
commit()	

Recuperável

T_1	T_2
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 10
	write(A)
commit()	
	commit()

Schedules recuperáveis

Rollback em cascata

Em um escalonamento recuperável pode ocorrer um fenômeno conhecido como *rollback* em cascata, no qual uma transação não confirmada tenha que ser desfeita porque leu um item de uma transação que falhou.

Rollback em cascata

T_1	T_2
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 10
	write(A)
abort()	

Schedules recuperáveis

Schedule sem cascata

Um *schedules* S é recuperável e evita aborto em cascata se uma T_i em S só puder ler dados que tenham sido atualizados por transações que já concluíram.

Schedules recuperáveis

Schedule sem cascata

T_1	T_2
read(A)	
A = A - 20	
write(A)	
commit()	
	read(A)
	A = A + 10
	write(A)
	...

Schedules recuperáveis

Schedule estrito

Transações não podem ler nem gravar um item x até que a última transação que gravou x tenha sido confirmada (ou cancelada).

Schedules recuperáveis

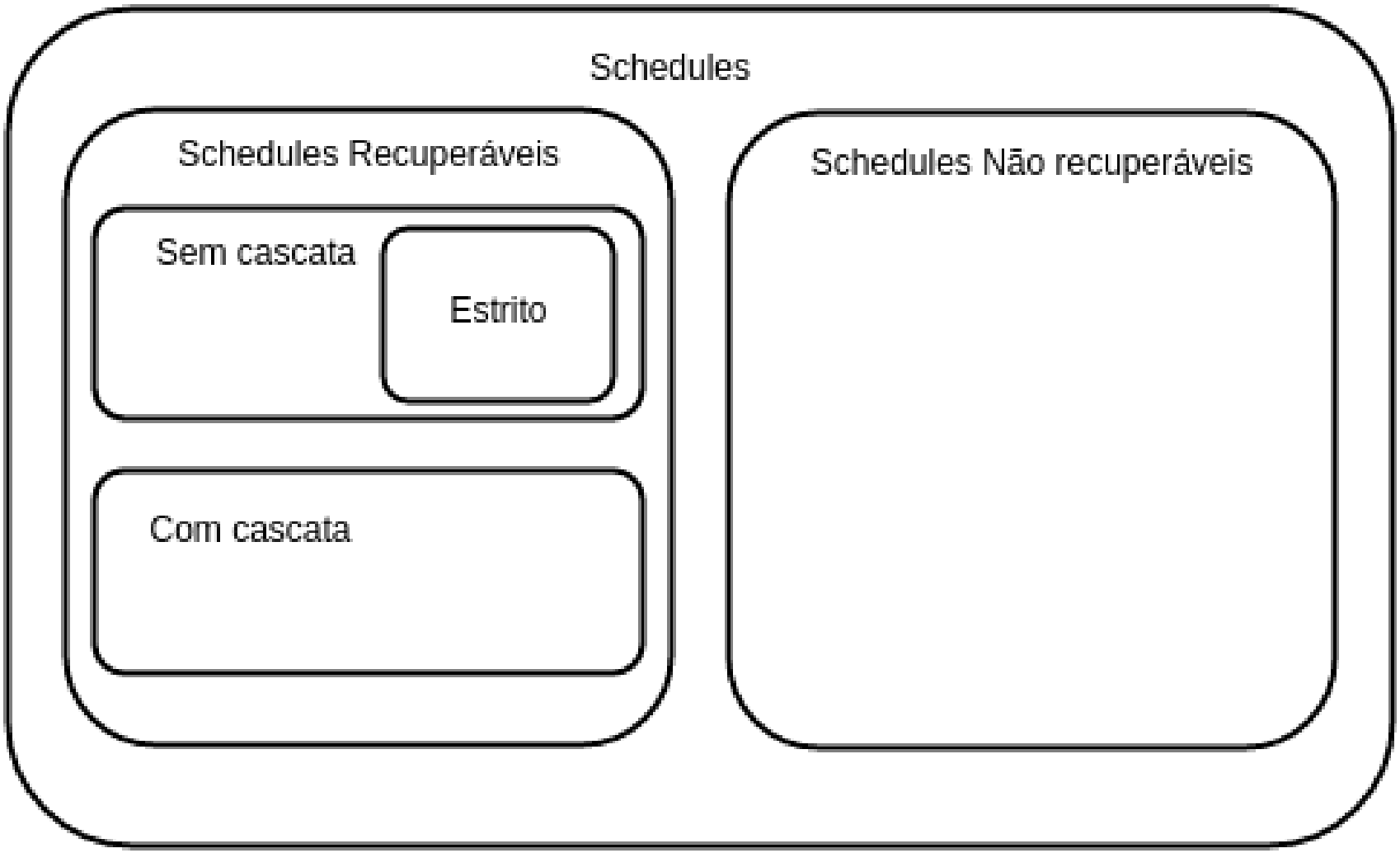
Não estrito

T_1	T_2
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 10
	write(A)
commit()	
	commit()

Estrito

T_1	T_2
read(A)	
A = A - 20	
write(A)	
commit()	
	read(A)
	A = A + 10
	write(A)
	commit()

Resumo


Schedules
Schedules Recuperáveis
Sem cascata
Estrito
Com cascata
Schedules Não recuperáveis

Tópicos

Schedules

- Até agora, caracterizamos *schedules* com base em suas propriedades de recuperação.
- Agora, caracterizaremos os tipo de *schedules* que sempre são considerados corretos quando transações concorrentes são executadas.

Schedule serial

Schedule serial

Um *schedule* é chamado serial se as operações de cada transação forem executadas consecutivamente, sem quaisquer operações intercaladas.

Schedules seriais

T_1	T_2
read(X)	
X = X - N	
write(X)	
read(Y)	
Y = Y + N	
write(Y)	
	read(X)
	X = X + M
	write(X)

T_1 « T_2

T_1	T_2
	read(X)
	X = X + M
	write(X)
read(X)	
X = X - N	
write(X)	
read(Y)	
Y = Y + N	
write(Y)	

T_2 « T_1

Schedules não seriais

T_1	T_2
read(X)	
X = X - N	
write(X)	
	read(X)
	X = X + M
	write(X)
read(Y)	
Y = Y + N	
write(Y)	

T_1	T_2
read(X)	
X = X - N	
	read(X)
	X = X + M
write(X)	
read(Y)	
	write(X)
Y = Y + N	
write(Y)	

Schedules serializáveis

Schedules serializáveis

Um *schedule* S é serializável se ele é **equivalente** com algum *schedule* serial S' com as mesmas n transações.

Exemplos

Schedule serializável

T_1	T_2
read(X)	
X = X - N	
write(X)	
	read(X)
	X = X + M
	write(X)
read(Y)	
Y = Y + N	
write(Y)	

Schedule não serializável

T_1	T_2
read(X)	
X = X - N	
	read(X)
	X = X + M
write(X)	
read(Y)	
	write(X)
Y = Y + N	
write(Y)	

O que é ser um *schedule* equivalente?

- Há duas **noções de equivalência** de *schedules*
  - Equivalente no resultado.
  - Equivalente em conflito.

Schedules equivalentes

Schedules equivalentes no resultado

Dois *schedules* são chamados de equivalentes no resultado se eles produzem o mesmo estado final no banco de dados.

- Problema em ser equivalentes no resultados:

x = 100

T_1
read(X)
X = X + 10
write(X)

T_2
read(X)
X = X * 1,1
write(X)

Operações em conflito

- Para entender *schedules* equivalentes em conflito, primeiramente, temos que entender **operações em conflito**.
- Duas operações em um *schelude* S são consideradas **em conflito** se satisfazem todas as três condições a seguir:
  1. Pertencem a diferentes transações.
  2. Acessam o mesmo item x.
  3. Pelo menos uma das operações é um `write_item(X)`.

Tabela: Tabela de conflito

	$r_j(x)$	$w_j(x)$
$r_i(x)$	false	true
$w_i(x)$	true	true

Operações em conflito

$$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); r_1(Y);$$

Ideia intuitiva

Duas operações estão em conflito se a alteração de sua ordem tiver efeitos diferentes.

Operações em conflito

Conflito de Leitura-Gravação

$$\begin{array}{ccc} r_i(x) & \times & w_j(x) \\ & \downarrow & \\ w_j(x) & \times & r_i(x) \end{array}$$

Conflito de Gravação-Gravação

$$\begin{array}{ccc} w_i(x) & \times & w_j(x) \\ & \downarrow & \\ w_j(x) & \times & w_i(x) \end{array}$$

Schedule completo

- Um *schedule* S é dito ser completo se:
  - As operações em S são exatamente aquelas operações em $T_1, T_2, ..., T_n$ incluindo uma operação de confirmação ou aborto como última operação em cada transação no *schedule*.
  - Para qualquer par de operações da mesma transação T_i, sua ordem de aparecimento relativa em S é a mesma que sua ordem de aparecimento em T_i.
  - Para duas operações quaisquer em conflito, uma das duas precisa ocorrer antes da outra no *schedule*.

Schedules equivalentes em conflito

Schedules equivalentes em conflito

Dois *schedules* são ditos ser equivalentes em conflito se a ordem de quaisquer operações conflitantes é a mesma em ambos *schedules*.

Schedules equivalentes em conflito

- Exemplo de dois *schedules* S e S' equivalentes em conflito

Schedule S

T_1	T_2
read(X)	
X = X - N	
write(X)	
read(Y)	
Y = Y + N	
write(Y)	
	read(X)
	X = X + M
	write(X)

Schedule S'

T_1	T_2
read(X)	
X = X - N	
write(X)	
	read(X)
	X = X + M
	write(X)
read(Y)	
Y = Y + N	
write(Y)	

Schedules serializáveis em conflito

- Agora, com a noção de equivalência em conflito, nós podemos melhorar nossa definição de *schedule* serializável.

Schedules serializáveis

Um *schedule* S é serializável se ele é **equivalente** com algum *schedule* serial S' com as mesmas n transações.

Schedules serializáveis em conflito

Um *schedule* S é serializável em conflito se ele é **equivalente em conflito** com algum *schedule* serial S' com as mesmas n transações.

Resumo

- Ser serializável **não** é o mesmo de ser serial
- Ser serializável implica que o *schedule* é um *schedule* **correto**.
  - Ele levará o banco de dados para um estado consistente.
  - A intercalação é apropriada e resultará em um estado como se as transações fossem executadas serialmente, ainda assim, alcançarão eficiência devido a execução concorrente.
- Verificar se um *schedule* é seriável é computacionalmente complexo
  - A intercalação das operações ocorre no sistema operacional através de algum escalonador.
  - Difícil determinar de antemão como as operações em uma transação serão intercaladas.

Tópicos

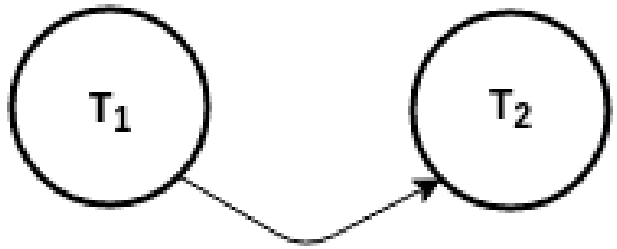
Testando a serialização por conflito em schedules

1. Para cada transação T_i participante no *schedule* S, crie um nó rotulado com T_i no grafo de precedência.
2. Para cada caso onde T_j executa um `read(x)` depois de T_i executar um `write(x)`, crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
3. Para cada caso onde T_j executa um `write(x)` após T_i executar um `read(x)`, crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
4. Para cada caso onde T_j executa um `write(x)` após T_i executar um `write(x)`, crie uma aresta $(T_i \rightarrow T_j)$ no grafo de precedência.
5. O *schedule* S é serializável se, e somente se, o grafo de precedência não tiver ciclos.

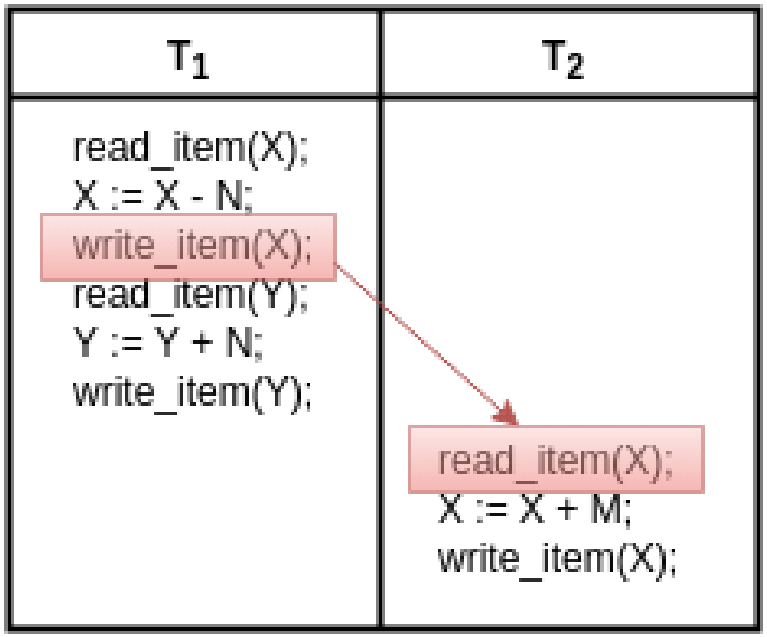
Algoritmo para construir um grafo de precedência

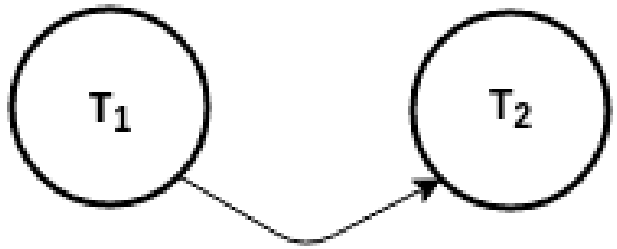
T_1	T_2

read_item(X);
X := X - N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
	read_item(X);
	X := X + M;
	write_item(X);

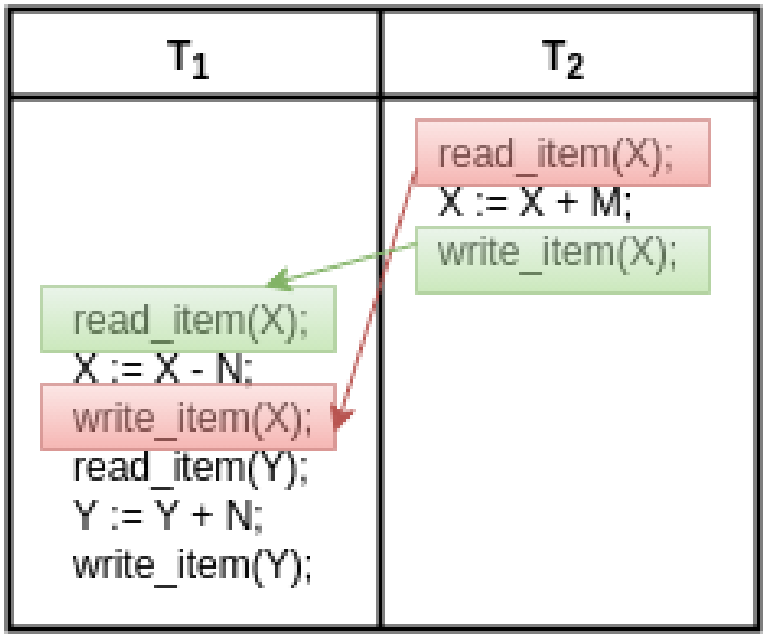


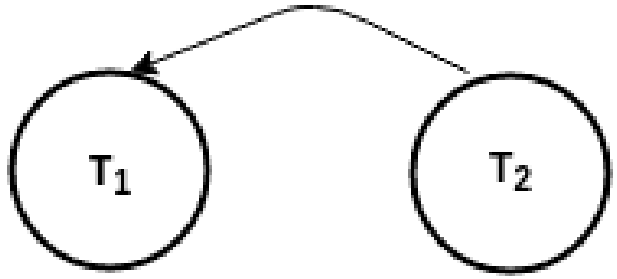
Algoritmo para construir um grafo de precedência



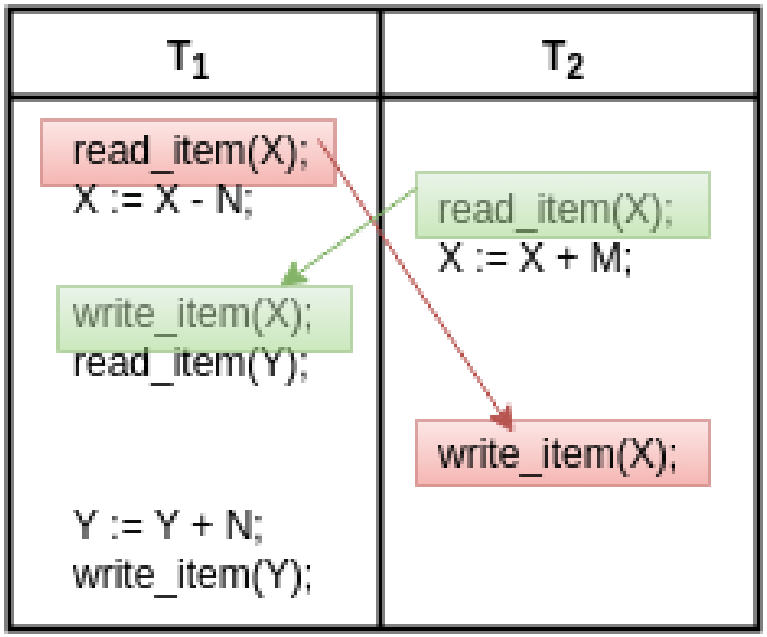


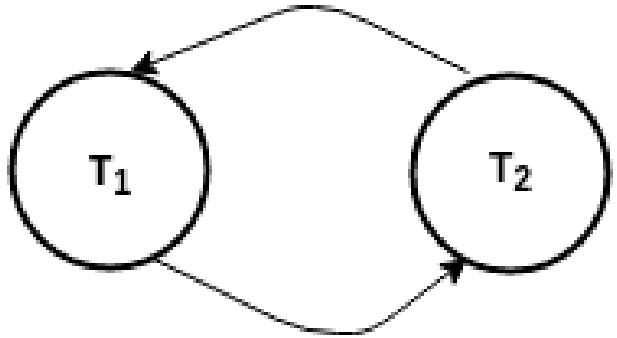
Algoritmo para construir um grafo de precedência



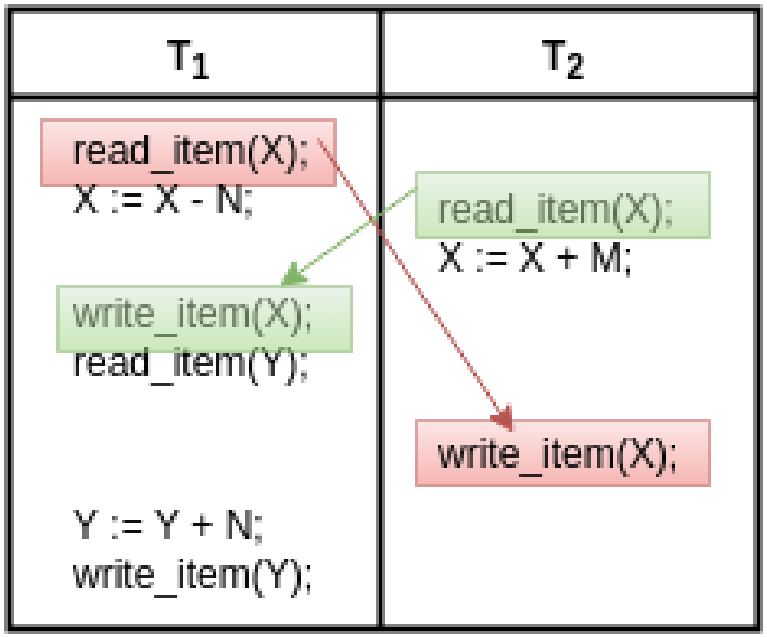


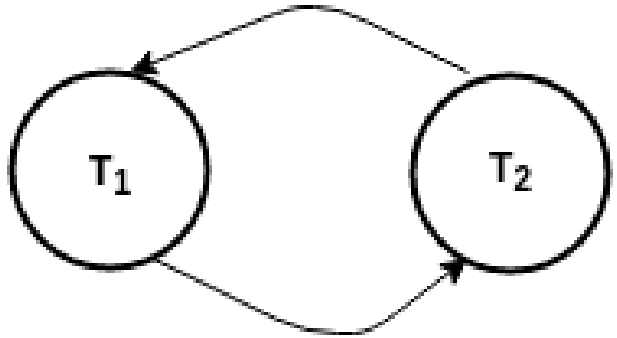
Algoritmo para construir um grafo de precedência





Algoritmo para construir um grafo de precedência





Exercício

- Qual dos seguintes *schedules* é serializável? Para cada *schedule* serializável, determine os *schedules* seriais equivalentes.
  a) $r_1(x); r_3(x); w_1(x); r_2(x); w_3(x);$
  b) $r_1(x); r_3(x); w_3(x); w_1(x); r_2(x);$
  c) $r_3(x); r_2(x); w_3(x); r_1(x); w_1(x);$
  d) $r_3(x); r_2(x); r_1(x); w_3(x); w_1(x);$

Exercício

- Considere as três transações T_1, T_2 e T_3 e os *schedules* S_1 e S_2 a seguir. Desenhe os grafos de serialização (precedência) para S_1 e S_2 e indique se cada *schedule* é serializável ou não. Se um *schedule* for serializável, escreva o(s) schedule(s) serial(is) equivalente(s).

$\mathtt{T_1 : r_1(x); r_1(z); w_1(x);}$
$\mathtt{T_2 : r_2(z); r_2(y); w_2(z); w_2(y);}$
$\mathtt{T_3 : r_3(x); r_3(y); w_3(y);}$

$\mathtt{S_1 : r_1(x); r_2(z); r_1(z); r_3(x); r_3(y); w_1(x); w_3(y); r_2(y); w_2(z); w_2(y);}$
$\mathtt{S_2 : r_1(x); r_2(z); r_3(x); r_1(z); r_2(y); r_3(y); w_1(x); w_2(z); w_3(y); w_2(y);}$

Referência

- ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados.** 6. ed. São Paulo, SP: Pearson Addison-Wesley, c2011. xviii, 788 p. ISBN 8588639173.
- Capítulo 21 - Introdução aos conceitos e teoria de processamento de transações.
- Exercícios recomendados: 1 a 7; 9; 14 a 19; 22 a 24.