

# Gerenciamento de Transações

## Controle de Concorrência

# Controle de Concorrência

- SGBD
  - sistema multiusuário em geral
    - diversas transações executando simultaneamente
- Garantia de **isolamento** de Transações
  - 1ª solução: uma transação executa por vez
    - **escalonamento serial** de transações
    - solução bastante ineficiente!
      - várias transações podem esperar muito tempo para execução
      - CPU pode ficar muito tempo ociosa
        - » enquanto uma transação faz I/O, por exemplo, outras transações poderiam ser executadas

# Controle de Concorrência

- Solução mais eficiente
  - execução concorrente de transações de modo a preservar o isolamento
    - escalonamento (*schedule*) não-serial e íntegro
  - responsabilidade do subsistema de controle de concorrência ou *scheduler*

execução  
serial

T1	T2
read(X)	
X = X - 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

execução  
não-serial  
ou concorrente

T1	T2
read(X)	
X = X - 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

# Scheduler

- Responsável pela definição de escalonamentos não-seriais de transações
- “Um escalonamento  $E$  define uma ordem de execução das operações de várias transações, sendo que *a ordem das operações de uma transação  $T_x$  em  $E$  aparece na mesma ordem na qual elas ocorrem isoladamente em  $T_x$* ”
- Problemas de um escalonamento não-serial mal definido (inválido)
  - atualização perdida (*lost-update*)
  - leitura suja (*dirty-read*)

# Atualização Perdida

- Uma transação  $T_y$  grava em um dado atualizado por uma transação  $T_x$

T1	T2
read(X)	
$X = X - 20$	
	read(Z)
	$X = Z + 10$
write(X)	
read(Y)	
	write(X)
$Y = X + 30$	
write(Y)	

← a atualização de X  
por T1 foi perdida!

# Leitura Suja

- $T_x$  atualiza um dado  $X$ , outras transações posteriormente lêem  $X$ , e depois  $T_x$  falha

T1	T2
read(X)	
$X = X - 20$	
write(X)	
	read(X)
	$X = X + 10$
	write(X)
read(Y)	
Rollback()	

← T2 leu um valor de  $X$  que não será mais válido!

# Scheduler

- Deve evitar escalonamentos inválidos
  - exige análise de operações em conflito
    - operações que pertencem a transações ativas diferentes
    - transações acessam o mesmo dado
    - pelo menos uma das operações é *write*
  - tabela de situações de conflito de transações
    - podem gerar um estado inconsistente no BD

		Ty	
Tx		read(X)	write(X)
	read(X)		✓
	write(X)	✓	✓

# *Scheduler X Recovery*

- *Scheduler* deve cooperar com o *Recovery*!
- Categorias de escalonamentos considerando o grau de cooperação com o *Recovery*
  - recuperáveis X não-recuperáveis
  - permitem *rollback* em cascata X evitam *rollback* em cascata
  - estritos X não-estritos



# Escalonamento Recuperável

- Garante que, se  $Tx$  realizou *commit*,  $Tx$  não irá sofrer UNDO
  - o *recovery* espera sempre esse tipo de escalonamento!
- Um escalonamento  $E$  é recuperável se nenhuma  $Tx$  em  $E$  for concluída até que todas as transações que gravaram dados lidos por  $Tx$  tenham sido concluídas

T1	T2
read(X)	
$X = X - 20$	
write(X)	
	read(X)
	$X = X + 10$
	write(X)
	<i>commit( )</i>
<i>rollback( )</i>	

escalonamento  
não-recuperável

T1	T2
read(X)	
$X = X - 20$	
write(X)	
	read(X)
	$X = X + 10$
	write(X)
<i>commit( )</i>	
	<i>commit( )</i>

escalonamento  
recuperável

# Escalonamento sem *Rollback* em Cascata

- Um escalonamento recuperável pode gerar *rollback* de transações em cascata
  - consome muito tempo de *recovery*!
- Um escalonamento  $E$  é recuperável e evita *rollback* em cascata se uma  $Tx$  em  $E$  só puder ler dados que tenham sido atualizados por transações que já concluíram

escalonamento  
recuperável  
com *rollback*  
em  
cascata

T1	T2
read(X)	
X = X - 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
rollback( )	...

escalonamento  
recuperável  
sem *rollback* em  
cascata

T1	T2
read(X)	
X = X - 20	
write(X)	
commit( )	
	read(X)
	X = X + 10
	write(X)
	...

# Escalonamento Estrito

- Garante que, se  $T_x$  deve sofrer UNDO, basta gravar a *before image* dos dados atualizados por ela
- Um escalonamento  $E$  é recuperável, evita *rollback* em cascata e é estrito se uma  $T_x$  em  $E$  só puder ler ou atualizar um dado  $X$  depois que todas as transações que atualizaram  $X$  tenham sido concluídas

T1	T2
read(X)	
$X = X - 20$	
write(X)	
	read(Y)
	$X = Y + 10$
	write(X)
	commit( )
rollback( )	

escalonamento  
recuperável  
sem *rollback* em  
cascata e  
não-estricto

T1	T2
read(X)	
$X = X - 20$	
write(X)	
commit( )	
	read(Y)
	$X = Y + 10$
	write(X)
	commit( )

escalonamento  
recuperável  
sem *rollback* em  
cascata e  
estricto

# Teoria da *Serializabilidade*

- Garantia de escalonamentos não-seriais **válidos**
- Premissa
  - “*um escalonamento não-serial de um conjunto de transações deve produzir resultado **equivalente** a alguma execução **serial** destas transações*”

entrada:	<b>T1</b>	<b>T2</b>
$X = 50$	read(X)	
$Y = 40$	$X = X - 20$	
	write(X)	
execução serial	read(Y)	
	$Y = Y + 20$	
	write(Y)	
saída:		read(X)
$X = 40$		$X = X + 10$
$Y = 60$		write(X)

entrada:	<b>T1</b>	<b>T2</b>
$X = 50$	read(X)	
$Y = 40$	$X = X - 20$	
	write(X)	
execução não-serial		read(X)
<b>serializável</b>		$X = X + 10$
		write(X)
saída:	read(Y)	
$X = 40$	$Y = Y + 20$	
$Y = 60$	write(Y)	

# Verificação de Serializabilidade

- Duas técnicas principais
  - equivalência de conflito
  - equivalência de visão
- Equivalência de Conflito
  - *Dado um escalonamento não-serial  $E'$  para um conjunto de Transações  $T$ ,  $E'$  é serializável em conflito se  $E'$  for equivalente em conflito a algum escalonamento serial  $E$  para  $T$ , ou seja, a ordem de quaisquer 2 operações em conflito é a mesma em  $E'$  e  $E$ .*

# Equivalência de Conflito - Exemplo

escalonamento serial  $E$

T1	T2
read(X)	
X = X - 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

escalonamento não-serial  $E1$

T1	T2
read(X)	
X = X - 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

escalonamento não-serial  $E2$

T1	T2
read(X)	
X = X - 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	

- $E1$  equivale em conflito a  $E$
- $E2$  não equivale em conflito a nenhum escalonamento serial para T1 e T2
- $E1$  é serializável e  $E2$  não é serializável

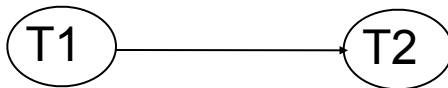
# Verificação de Equivalência em Conflito

- Construção de um grafo direcionado de precedência
  - nós são IDs de transações
  - arestas rotuladas são definidas entre 2 transações T1 e T2 se existirem operações em conflito entre elas
    - direção indica a ordem de precedência da operação
      - origem indica onde ocorre primeiro a operação
- Um grafo com ciclos indica um escalonamento não-serIALIZÁVEL em conflito!

# Grafo de Precedência

escalonamento serializável  $E1$

T1	T2
read(X)	
$X = X - 20$	
write(X)	
	read(X)
	$X = X + 10$
	write(X)
read(Y)	
$Y = Y + 20$	
write(Y)	



escalonamento não-serIALIZÁVEL  $E2$

T1	T2
read(X)	
$X = X - 20$	
	read(X)
	$X = X + 10$
write(X)	
read(Y)	
	write(X)
$Y = Y + 20$	
write(Y)	





# Verificação de Serializabilidade

- Técnicas propostas (em conflito e de visão) são difíceis de serem testadas
  - exige que se tenha um conjunto fechado de transações para fins de verificação
- Na prática
  - conjunto de transações executando concorrentemente é muito dinâmico!
    - novas transações estão sendo constantemente submetidas ao SGBD para execução
  - logo, a serializabilidade é garantida através de técnicas (ou protocolos) de controle de concorrência que não precisam testar os escalonamentos

# História

- Representação seqüencial da execução entrelaçada de um conjunto de transações concorrentes

escalonamento não-serIALIZÁVEL  $E_2$

– operações consideradas

- *read* (r), *write* (w),  
*commit* (c), *rollback* (a)

- Exemplo

$H_{E_2} = r1(x) \ r2(x) \ w1(x) \ w2(x) \ w1(y) \ c1 \ c2$

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	
commit( )	
	commit( )

# Técnicas de Controle de Concorrência

- Pessimistas

- supõem que sempre ocorre interferência entre transações e garantem a serializabilidade enquanto a transação está ativa
- técnicas
  - bloqueio (*locking*)
  - *timestamp*

- Otimistas

- supõem que quase nunca ocorre interferência entre transações e verificam a serializabilidade somente ao final de uma transação
- técnica
  - validação

# Técnicas Baseadas em Bloqueio

- Técnicas mais utilizadas pelos SGBDs
- Princípio de funcionamento
  - controle de operações *read(X)* e *write(X)* e postergação (através de bloqueio) de algumas dessas operações de modo a evitar conflito
- Todo dado possui um **status de bloqueio**
  - **liberado** (*Unlocked* - U)
  - com **bloqueio compartilhado** (*Shared lock* - S)
  - com **bloqueio exclusivo** (*eXclusive lock* - X)

# Modos de Bloqueio

- **Bloqueio Compartilhado (S)**
  - solicitado por uma transação que deseja realizar leitura de um dado D
    - várias transações podem manter esse bloqueio sobre D
- **Bloqueio Exclusivo (X)**
  - solicitado por uma transação que deseja realizar leitura+atualização de um dado D
    - uma única transação pode manter esse bloqueio sobre D
- **Matriz de Compatibilidade de Bloqueios**

	S	X
S	verdadeiro	falso
X	falso	falso

- **Informações de bloqueio são mantidas no DD**  
<ID-dado, status-bloqueio, ID-transação>

# Operações de Bloqueio na História

- O *Scheduler* gerencia bloqueios através da invocação automática de operações de bloqueio conforme a operação que a transação deseja realizar em um dado
- Operações
  - $Is(D)$ : solicitação de bloqueio compartilhado sobre D
  - $Ix(D)$ : solicitação de bloqueio exclusivo sobre D
  - $u(D)$ : libera o bloqueio sobre D

# Exemplo de História com Bloqueios

T1	T2
lock-S(Y)	
read(Y)	
unlock(Y)	
	lock-S(X)
	lock-X(Y)
	read(X)
	read(Y)
	unlock(X)
	write(Y)
	unlock(Y)
	commit( )
lock-X(X)	
read(X)	
write(X)	
unlock(X)	
commit( )	

$H = ls1(Y) \ r1(Y) \ u1(Y) \ ls2(X) \ lx2(Y) \ r2(X) \ r2(Y) \ u2(X) \ w2(Y) \ u2(Y) \ c2$   
 $lx1(X) \ r1(X) \ w1(X) \ u1(X) \ c1$

# Implementação das Operações

- Solicitação de bloqueio compartilhado

```
lock-S(D, Tx)
início
    se lock(D) = 'U' então
        início
            insere Tx na lista-READ(D);
            lock(D) ← 'S';
        fim
    senão se lock(D) = 'S' então insere Tx na lista-READ(D)
        senão /* lock(D) = 'X' */ insere (Tx, 'S') na fila-WAIT(D);
fim
```

status de bloqueio de D

lista de transações com bloqueio compartilhado sobre D

fila de transações aguardando a liberação de um bloqueio conflitante sobre D

Obs.: supor que os métodos de inclusão/exclusão de elementos nas EDs automaticamente alocam/desalocam a ED caso ela não exista/se torne vazia

fila de transações aguardando a liberação de um bloqueio conflitante sobre D



# Uso de Bloqueios “S” e “X”

- Não garantem escalonamentos serializáveis
- Exemplo

$H_{N-SR} = ls1(Y) \text{ r1}(Y) u1(Y) ls2(X) \text{ r2}(X) u2(X) lx2(Y) r2(Y)$   
 $\text{w2}(Y) u2(Y) c2 lx1(X) r1(X) \text{ w1}(X) u1(X) c1$



- Necessita-se de uma técnica mais rigorosa de bloqueio para garantir a serializabilidade
  - técnica mais utilizada
    - bloqueio de duas fases (*two-phase locking* – 2PL)

# Bloqueio de 2 Fases – 2PL

- Premissa
  - *“para toda transação Tx, todas as operações de bloqueio de dados feitas por Tx precedem a primeira operação de desbloqueio feita por Tx”*
- Protocolo de duas fases
  1. Fase de expansão ou crescimento
    - Tx pode obter bloqueios, mas não pode liberar nenhum bloqueio
  2. Fase de retrocesso ou encolhimento
    - Tx pode liberar bloqueios, mas não pode obter nenhum bloqueio

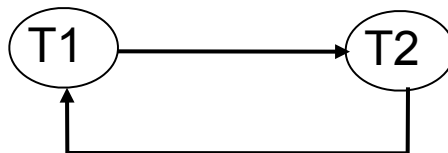
# Scheduler 2PL - Exemplo

- T1: r(Y) w(Y) w(Z)
- T2: r(X) r(Y) w(Y) r(Z) w(Z)

## Contra-Exemplo

$H_{N-2PL} = lx1(Y) r1(Y) ls2(X) r2(X) u2(X) w1(Y) u1(Y) lx2(Y)$   
 $r2(Y) w2(Y) u2(Y) lx2(Z) r2(Z) w2(Z) c2 lx1(Z) w1(Z)$   
 $u1(Z) c1$

não é 2PL!

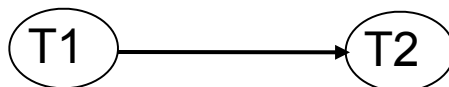


não garantiu SR!

## Exemplo

$H_{2PL} = ls2(X) r2(X) lx1(Y) r1(Y) lx1(Z) w1(Y) u1(Y) lx2(Y)$   
 $r2(Y) w1(Z) u1(Z) c1 w2(Y) lx2(Z) u2(X) u2(Y) w2(Z)$   
 $u2(Z) c2$

é SR!



$P_{max}(T1)$

$P_{max}(T2)$

# Scheduler 2PL - Crítica

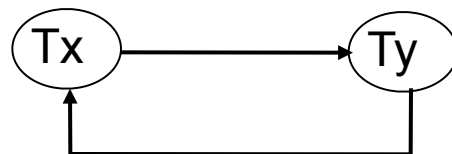
- Vantagem
  - técnica que sempre garante escalonamentos SR sem a necessidade de se construir um grafo de dependência para teste!
    - se  $T_x$  alcança  $P_{max}$ ,  $T_x$  não sofre interferência de outra transação  $T_y$ , pois se  $T_y$  deseja um dado de  $T_x$  em uma operação que poderia gerar conflito com  $T_x$ ,  $T_y$  tem que esperar (evita ciclo  $T_y \rightarrow T_x$ !)
    - depois que  $T_x$  liberar os seus dados, não precisará mais deles, ou seja,  $T_x$  não interferirá nas operações feitas futuramente nestes dados por  $T_y$  (evita também ciclo  $T_y \rightarrow T_x$ !)

# Scheduler 2PL - Crítica

- Desvantagens
  - limita a concorrência
    - um dado pode permanecer bloqueado por  $T_x$  muito tempo até que  $T_x$  adquira bloqueios em todos os outros dados que deseja
  - 2PL básico (técnica apresentada anteriormente) não garante escalonamentos
    - livres de *deadlock*
      - $T_x$  espera pela liberação de um dado bloqueado por  $T_y$  de forma conflitante e vice-versa
    - adequados à recuperação pelo *recovery*

# Deadlock (Impasse) de Transações

- Ocorrência de *deadlock*
  - $T_y$  está na Fila-WAIT(D1) de um dado D1 bloqueado por  $T_x$
  - $T_x$  está na Fila-WAIT(D2) de um dado D2 bloqueado por  $T_y$
- Pode ser descoberto através de um **grafo de espera de transações**
  - se o grafo é **cíclico** existe *deadlock*!



# Tratamento de *Deadlock*

- Protocolos de Prevenção
  - abordagens pessimistas
    - *deadlocks* ocorrem com frequência!
    - impõem um *overhead* no processamento de transações
      - controles adicionais para evitar *deadlock*
    - tipos de protocolos pessimistas
      - técnica de bloqueio 2PL conservador
      - técnicas baseadas em *timestamp* (*wait-die* e *wound-wait*)
      - técnica de *espera-cautelosa* (*cautious-waiting*)
  - uso de *timeout*
    - se tempo de espera de  $T_x > \text{timeout} \Rightarrow \text{Rollback}(T_x)$

# Técnicas Baseadas em *Timestamp*

*tempo de start de Tx*

- *Timestamp*
  - rótulo de tempo associado à  $T_x$  ( $TS(T_x)$ )
- Técnicas
  - consideram que  $T_x$  deseja um dado bloqueado por outra transação  $T_y$
  - Técnica 1: esperar-ou-morrer (*wait-die*)
    - se  $TS(T_x) < TS(T_y)$  então  $T_x$  espera  
senão início  
 $rollback(T_x)$   
 $start(T_x)$  com o mesmo TS  
fim



# Técnicas Baseadas em *Timestamp*

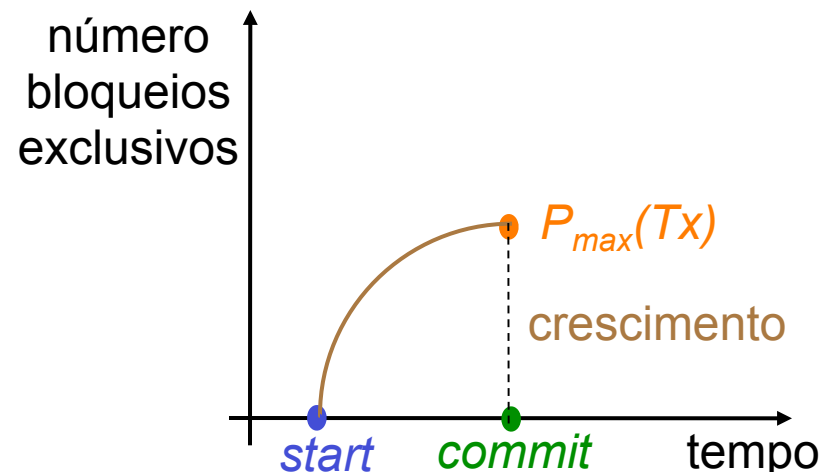
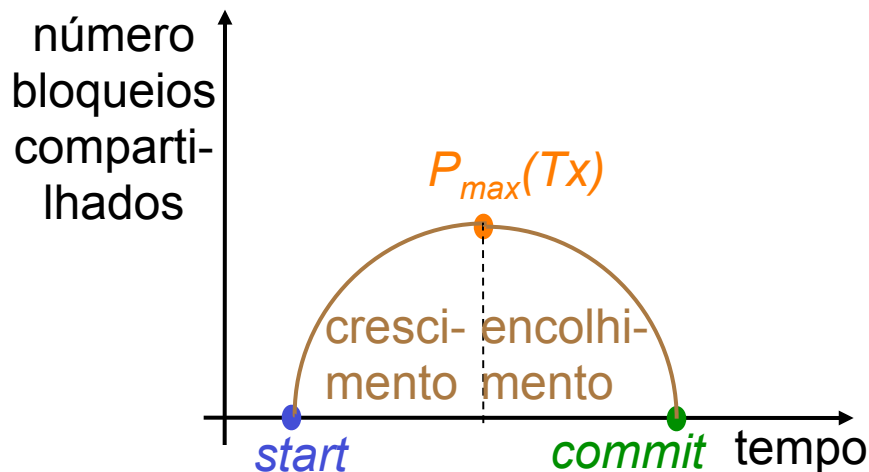
- Técnicas (cont.)
  - Técnica 2: *ferir-ou-esperar* (*wound-wait*)
    - se  $TS(T_x) < TS(T_y)$  então  
início  
     $rollback(T_y)$   
     $start(T_y)$  com o mesmo TS  
fim  
senão  $T_x$  espera
  - vantagem das técnicas
    - evitam *starvation* (espera indefinida) de uma  $T_x$ 
      - quanto mais antiga for  $T_x$ , maior a sua prioridade
  - desvantagem das técnicas
    - muitos *rollbacks* podem ser provocados, sem nunca ocorrer um *deadlock*

# Tratamento de *Deadlock*

- Protocolos de Detecção
  - abordagens otimistas
    - *deadlocks* não ocorrem com frequência!
      - são tratados quando ocorrem
    - mantém-se um grafo de espera de transações
    - se há *deadlock*, seleciona-se uma transação vítima *Tx* através de um ou mais critérios
      - quanto tempo *Tx* está em processamento
      - quantos itens de dado *Tx* já leu/escreveu
      - quantos itens de dado *Tx* ainda precisa ler/escrever
      - quantas outras transações serão afetadas pelo *Rollback(Tx)*

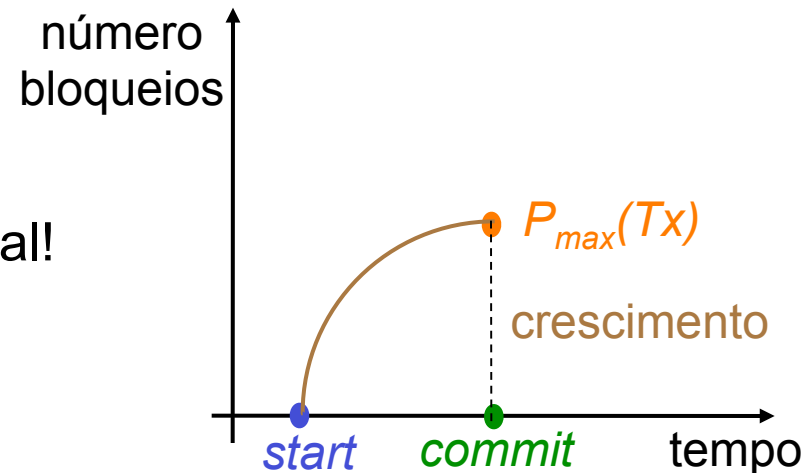
# Outras Técnicas de Bloqueio 2PL

- *Scheduler 2PL Conservador ou Estático*
  - evita *deadlock*, porém *Tx* pode esperar muito para executar
- *Scheduler 2PL Estrito* (muito usado pelos SGBDs)
  - *Tx* só libera seus bloqueios exclusivos após executar *commit* ou *rollback*



# Outras Técnicas de Bloqueio 2PL


- *Scheduler 2PL Estrito*
  - vantagem: garante escalonamentos estritos
  - desvantagem: não está livre de *deadlocks*
- *Scheduler 2PL (Estrito) Rigoroso*
  - Tx só libera seus bloqueios após executar *commit* ou *rollback*
  - vantagem
    - menos *overhead* para Tx
      - Tx libera tudo apenas no final!
  - desvantagem
    - limita mais a concorrência




# Scheduler Baseado em *Timestamp*

- Técnica na qual toda transação  $T_x$  possui uma marca *timestamp* ( $TS(T_x)$ )
- Princípio de funcionamento (TS-Básico)
  - “no acesso a um item de dado  $D$  por operações conflitantes, a ordem desse acesso deve ser equivalente à ordem de  $TS$  das transações envolvidas”
    - garante escalonamentos serializáveis através da ordenação de operações conflitantes de acordo com os  $TS$ s das transações envolvidas
  - cada item de dado  $X$  possui um registro  $TS$  ( $R-TS(X)$ )  
<ID-dado,  $TS-Read$ ,  $TS-Write$ >

TS da transação mais recente  
que leu o dado



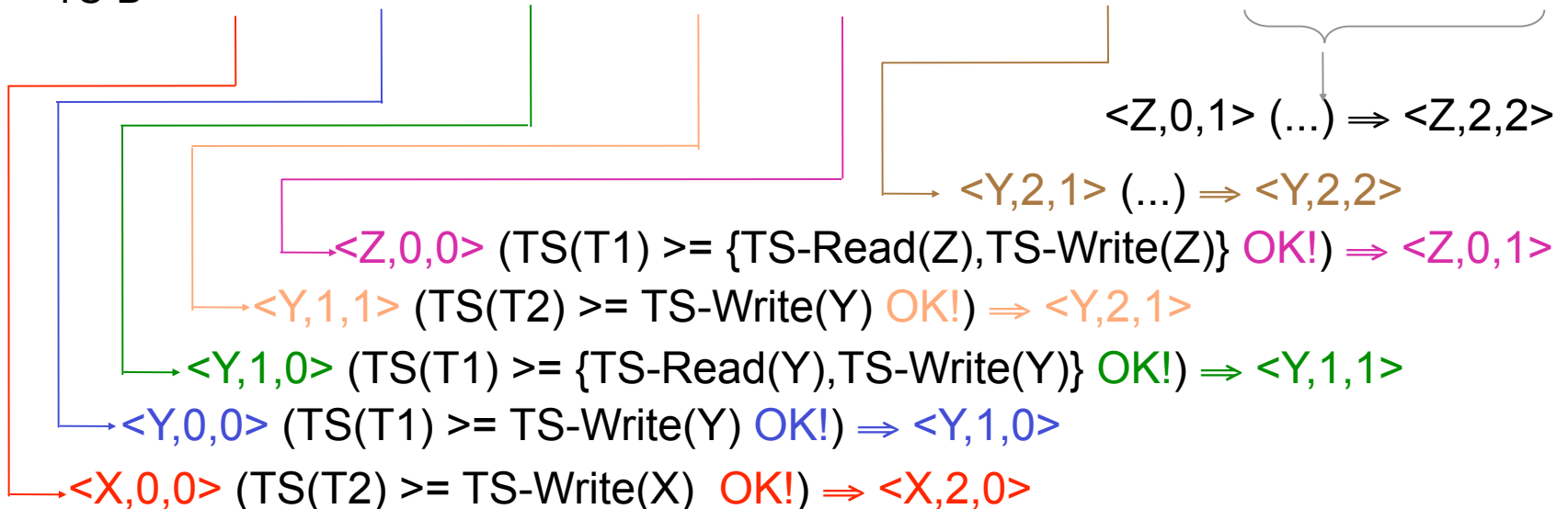
TS da transação mais recente  
que atualizou o dado



# Técnica TS-Básico - Exemplo

- **T1**:  $r(Y) \ w(Y) \ w(Z)$   $\rightarrow TS(T1) = 1$
- **T2**:  $r(X) \ r(Y) \ w(Y) \ r(Z) \ w(Z)$   $\rightarrow TS(T2) = 2$
- Registros iniciais de TS de X, Y e Z:
  - $\langle X, 0, 0 \rangle; \langle Y, 0, 0 \rangle; \langle Z, 0, 0 \rangle$
- Exemplo de escalonamento serializável por TS

$H_{TS-B} = r2(X) \ r1(Y) \ w1(Y) \ r2(Y) \ w1(Z) \ c1 \ w2(Y) \ r2(Z) \ w2(Z) \ c2$



# Algoritmo TS-Básico

```
TS-Básico(Tx, dado, operação)
início
  se operação = 'READ' então
    se TS(Tx) < R-TS(dado).TS-Write então
      início rollback(Tx);
      restart(Tx) com novo TS;
    fim
  senão início executar read(dado);
    se R-TS(dado).TS-Read < TS(Tx) então
      R-TS(dado).TS-Read ← TS(Tx);
    fim
  senão início /* operação = 'WRITE' */
    se TS(Tx) < R-TS(dado).TS-Read OU
      TS(Tx) < R-TS(dado).TS-Write então
      início rollback(Tx);
      restart(Tx) com novo TS;
    fim
    senão início executar write(dado);
      R-TS(dado).TS-Write ← TS(Tx);
    fim
  fim
fim
```

# Técnica TS-Básico

- Vantagens

- técnica simples para garantia de serializabilidade (não requer bloqueios)
- não há *deadlock* (não há espera)

- Desvantagens

- gera muitos *rollbacks* de transações
  - passíveis de ocorrência quando há conflito
- pode gerar *rollbacks* em cascata
  - não gera escalonamentos adequados ao *recovery*

- Para minimizar essas desvantagens

- técnica de *timestamp* rigoroso (TS-Estrito)



# Técnica TS-Rigoroso

- Garante escalonamentos **serializáveis** e **estritos**
  - passíveis de *recovery* em caso de falha
- Funcionamento
  - baseado no TS-básico com a seguinte diferença
    - “se  $T_x$  deseja  $read(D)$  ou  $write(D)$  e  $TS(T_x) > R-TS(D).TS-Write$ , então  $T_x$  **espera** pelo *commit* ou *rollback* da transação  $T_y$  cujo  $R-TS(D).TS-Write = TS(T_y)$ ”
    - exige *fila-WAIT(D)*
    - não há risco de *deadlock*
      - nunca há ciclo pois somente transações mais novas esperam pelo *commit/rollback* de transações mais antigas
  - *overhead* no processamento devido à espera

# Técnica TS-Rigoroso - Exemplo

- **T1**:  $r(X)$   $w(X)$   $w(Z)$   $\rightarrow TS(T1) = 1$
- **T2**:  $r(X)$   $w(X)$   $w(Y)$   $\rightarrow TS(T2) = 2$
- Exemplo de escalonamento TS-Estrito

$H_{TS-E} = r1(X) w1(X) r2(X) w1(Z) c1 r2(X) w2(X) w2(Y) c2$

↓  
T2 espera por T1, pois  
 $TS(T2) > R-TS(X).TS-write$   
( $r2(X)$  não é executado  
e T2 é colocada na  
Fila-WAIT(X))

↓  
T1 já *commitou*!  
T2 pode executar  
agora  $r2(X)$   
(tira-se T2 da  
fila-WAIT(X))

# Schedulers Otimistas

- Técnicas pessimistas
  - *overhead* no processamento de transações
    - executam verificações e ações antes de qualquer operação no BD para garantir a serializabilidade (solicitação de bloqueio, teste de TS)
- Técnicas otimistas
  - não realizam nenhuma verificação durante o processamento da transação
    - pressupõem nenhuma ou pouca interferência
    - verificações de violação de serializabilidade feitos somente ao final de cada transação
    - técnica mais conhecida: Técnica de Validação

# Scheduler Baseado em Validação

- Técnica na qual atualizações de uma transação *Tx* são feitas sobre cópias locais dos dados
- Quando *Tx* solicita *commit* é feita a sua validação
  - *Tx* violou a serializabilidade?
    - **SIM**: *Tx* é abortada e reiniciada posteriormente
    - **NÃO**: atualiza o BD a partir das cópias dos dados e encerra *Tx*

# Técnica de Validação

- Cada transação  $T_x$  passa por 3 fases:

## 1. Leitura

- $T_x$  lê dados de transações *committed* do BD e atualiza dados em cópias locais

## 2. Validação

- análise da manutenção da serializabilidade de conflito caso as atualizações de  $T_x$  sejam efetivadas no BD

## 3. Escrita

- se fase de Validação for OK, aplica-se as atualizações de  $T_x$  no BD e  $T_x$  encerra com sucesso; caso contrário,  $T_x$  é abortada

# Scheduler Baseado em Validação

- Vantagens

- reduz o *overhead* durante a execução de  $T_x$
- evita *rollback* em cascata
  - $T_x$  não grava no BD antes de suas atualizações serem validadas em memória
    - se  $T_x$  interfere em outra  $T_y$  *committed* ou em validação, suas atualizações são descartadas

- Desvantagem

- se houve interferência entre  $T_x$  e outras transações (isso não é esperado pois a técnica é otimista), isso é descoberto somente ao final da execução de  $T_x$  (na validação) e só após essa validação  $T_x$  pode ser reiniciada