

# Controle de concorrência

Prof. Heloise Manica P. Teixeira

# Introdução

- Uma das propriedades fundamentais de uma transação é o **isolamento**. Quando diversas transações são executadas de modo concorrente corre-se o risco de violar esta propriedade.
- É necessário que o sistema controle transações concorrentes por meio de uma série de mecanismos, os quais formam o **controle de concorrência**.
- A base dos esquemas de concorrência discutidos aqui tem por base a propriedade de **serialização**, ou seja, todos os esquemas devem garantir que a ordenação de processamento seja serializada.
- → Por enquanto vamos assumir que sistemas não falham!

# Protocolos de Controle de Concorrência

- Oferecem várias **regras** que, se seguidas pelas transações, **garantem a serialização** de todos os escalonamentos nos quais as transações participam

- **Pessimistas**



- baseados na premissa que conflitos entre transações ocorrem com frequência
  - (alta probabilidade)

## Otimistas



baseados na premissa que conflitos entre transações são raros  
(baixa probabilidade)

# Protocolos de Controle de Concorrência

- Pessimistas



- testam as transações antes da execução de suas operações



- *two phase locking*  
*timestamp ordering*

## Otimistas



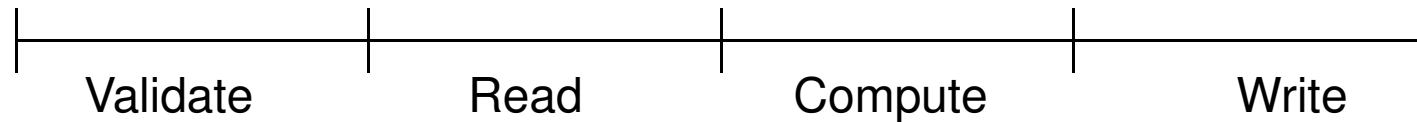
testam as transações após a execução de suas operações



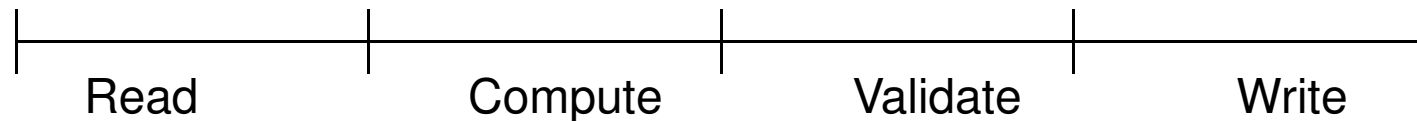
técnicas de validação

# Protocolos de Controle de Concorrência

**Execução pessimista** - Testam as transações **antes** da execução de suas operações



**Execução otimista** - Testam as transações **após** a execução de suas operações



# Protocolos baseados em bloqueio

- **Bloqueio (lock)**: é um mecanismo para controlar o acesso simultâneo a um item de dados
- Itens de dados podem ser bloqueados em dois modos:
  1. **Modo exclusivo (X)**. O item de dados pode ser lido e também escrito. O bloqueio X é solicitado pela instrução lock-X.
  2. **Modo compartilhado (S)**. O item de dados só pode ser lido. O bloqueio S é solicitado pela instrução lock-S.
- As solicitações de bloqueio são feitas ao gerenciador de controle de concorrência. A transação só pode prosseguir após a concessão da solicitação.

# Protocolos baseados em bloqueio (cont.)

- Matriz de compatibilidade de bloqueio:

	S	X
S	true	false
X	false	false

- Uma transação pode receber um bloqueio sobre um item se o bloqueio solicitado **for compatível** com os bloqueios já mantidos sobre o item por outras transações
- Qualquer quantidade de transações pode manter bloqueios compartilhados sobre um item, mas se qualquer transação mantiver um bloqueio exclusivo sobre um item, nenhuma outra pode manter qualquer bloqueio sobre o item.
- Se um bloqueio não puder ser concedido, a transação solicitante deve esperar até que todos os bloqueios incompatíveis mantidos por outras transações tenham sido liberados. O bloqueio é então concedido.

## Protocolos baseados em bloqueio (cont.)

- Exemplo de uma transação realizando bloqueio:

$T_2$ : lock-S(**A**);  
read (**A**);  
unlock(**A**);  
lock-S(**B**);  
read (**B**);  
unlock(**B**);  
display(**A+B**)

- O bloqueio acima não é suficiente para garantir a serialização - pois se **A** e **B** fossem atualizados entre a leitura de **A** e **B**, a soma exibida estaria errada.

- **Protocolo de bloqueio**: conjunto de regras seguidas por todas as transações enquanto solicita e libera bloqueios. Os protocolos de bloqueio restringem o conjunto de schedules possíveis.



# Armadilhas dos protocolos baseados em bloqueio

■ Considere o schedule parcial:

$T_3$	$T_4$
lock-X( $B$ )	
read( $B$ )	
$B := B - 50$	
write( $B$ )	
	lock-S( $A$ )
	read( $A$ )
	lock-S( $B$ )
lock-X( $A$ )	

- Nem  $T_3$  nem  $T_4$  podem ter progresso - a execução de lock-S( $B$ ) faz com que  $T_4$  espere que  $T_3$  libere seu bloqueio sobre  $B$ , enquanto a execução de lock-X( $A$ ) faz com que  $T_3$  espere que  $T_4$  libere seu bloqueio sobre  $A$ .

- Essa situação é chamada de **impasse**.

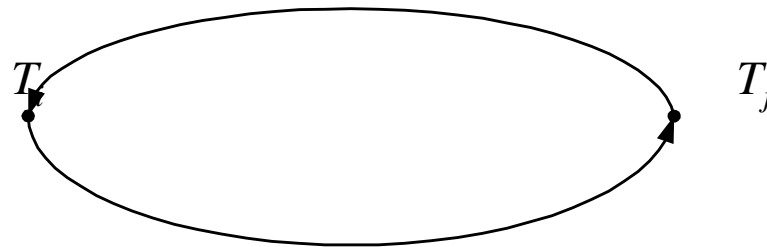
- Para lidar com um impasse, um dentre  $T_3$  ou  $T_4$  precisa ser revertido e seus bloqueios liberados.

## Armadilhas dos protocolos em bloqueio (cont.)

- O potencial para impasse existe na maioria dos protocolos de bloqueio. Os impasses são um **mal necessário**.
- Problema: **Inanição** é possível. Por exemplo:
  - Uma transação pode estar esperando por um bloqueio X sobre um item, enquanto uma seqüência de outras transações solicita e recebe um bloqueio S sobre o mesmo item.
  - A mesma transação é repetidamente revertida, devido aos impasses.

# Deadlock (Impasse)

- Um método utilizado para detecção de deadlock em sistemas distribuídos é analisar o grafo wait-for (GWF).
- Um GWF é um grafo onde os nós representam as transações, e um arco  $T_2 \rightarrow T_1$  significa que a transação  $T_2$  está esperando por  $T_1$ .



## Armadilhas dos protocolos em bloqueio (cont.)

### ■ Exemplo de Inanição

- Suponha que a transação  $T_2$  tenha um bloqueio compartilhado sobre um item de dado e a transação  $T_1$  solicite um bloqueio exclusivo do mesmo item.
- $T_1$  terá que esperar até que o bloqueio compartilhado feito por  $T_2$  seja liberado.
- Enquanto isso, uma transação  $T_3$  pode solicitar o bloqueio compartilhado sobre o mesmo item de dado.
- O bloqueio pedido é compatível com o bloqueio concedido a  $T_2$ , de modo que o bloqueio compartilhado pode ser concedido a  $T_3$ .
- Nesta altura,  $T_2$  pode liberar o bloqueio, mas  $T_1$  terá que esperar até que  $T_3$  termine.
- Novamente, aparece uma transação  $T_4$  que ... ..
- Logo, a transação  $T_1$  poderá nunca ser processada, sendo então chamada de *starved* (ou, em *starvation* – estagnada, em inanição).

## O protocolo de bloqueio em **duas fases**

- Esse é um protocolo que garante schedules seriáveis por conflito.
- Fase 1: **Fase de crescimento**
  - transação pode obter bloqueios
  - transação não pode liberar bloqueios
- Fase 2: **Fase de encurtamento**
  - transação pode liberar bloqueios
  - transação não pode obter bloqueios
- O protocolo garante a serialização.
- O bloqueio em duas fases **não garante** liberdade de impasses

## O protocolo de bloqueio em duas fases (cont.)

- O **bloqueio rigoroso em duas fases** é ainda mais estrito: *todos* os bloqueios são mantidos até um commit/abort.
- Nesse protocolo, as transações podem ser seriadas na ordem em que são confirmadas.
- Protocolos de **prevenção** de impasse
  - Exemplo - Bloqueio 2PL Conservador
    - Cada transação realiza todos os bloqueios antes que iniciar
  - Esquemas Esperar-Morrer e Ferir-Esperar
    - Aborta a transação mais jovem entre duas transações quando as duas possuem operações conflitantes

# Conversões de bloqueio

- Bloqueio em **duas fases** com **conversões** de bloqueio:
  - **Primeira fase:**
    - pode adquirir um bloqueio-S sobre o item
    - pode adquirir um bloqueio-X sobre o item
    - pode converter um bloqueio-S para um bloqueio-X (upgrade)
  - **Segunda fase:**
    - pode liberar um bloqueio-S
    - pode liberar um bloqueio-X
    - pode converter um bloqueio-X para um bloqueio-S (downgrade)
- Esse protocolo garante a serialização. Mas ainda conta com o programador para inserir as diversas instruções de bloqueio.

# Aquisição automática de bloqueios

- Uma transação  $T_i$  emite a instrução de leitura/escrita padrão, sem chamadas de bloqueio explícitas.
- A operação  $\text{read}(D)$  é processada como:

if  $T_i$  tem um bloqueio sobre  $D$

then

**$\text{read}(D)$**

else

begin

**se necessário, espera até que nenhuma outra**

**transação tenha um bloqueio-X sobre  $D$**

**concede a  $T_i$  um bloqueio-S sobre  $D$ ;**

**$\text{read}(D)$**

end



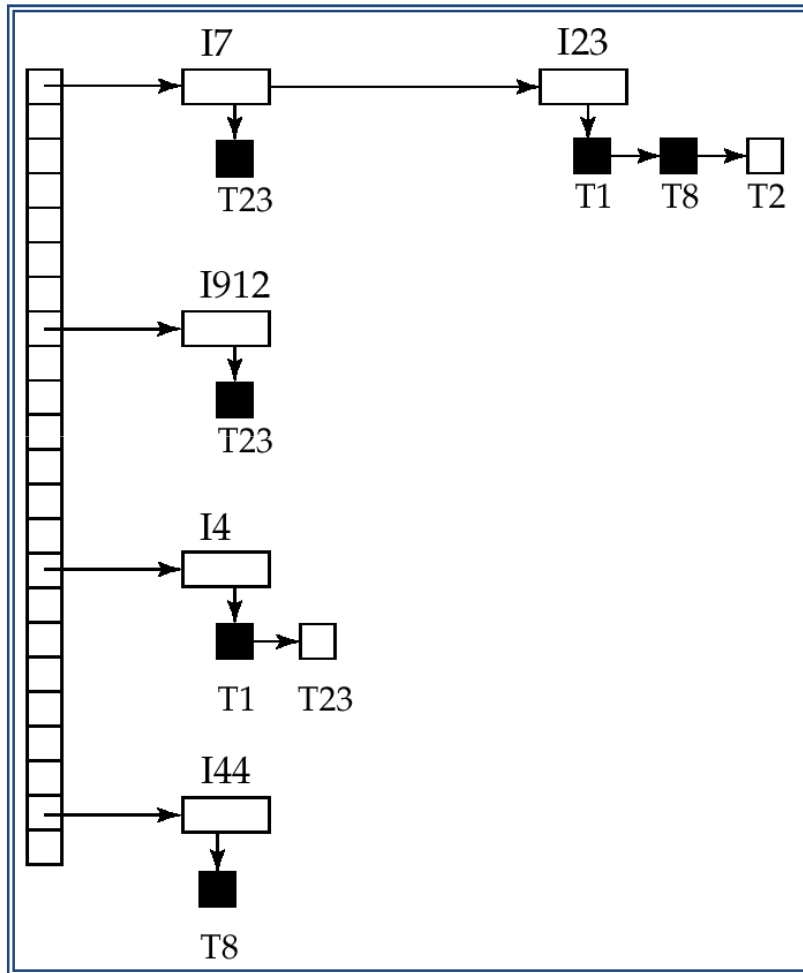
## Aquisição automática de bloqueios (cont.)

- **write( $D$ ) é processado como:**
  - if  $T_i$  tem um bloqueio-X sobre  $D$  then**
    - write( $D$ )**
  - else**
    - begin**
      - se for preciso, espera até que nenhuma outra transação tenha um bloqueio sobre  $D$ ,**
      - if  $T_i$  tem um bloqueio-S sobre  $D$** 
        - then**
          - upgrade do bloqueio sobre  $D$  para bloqueio-X**
        - else**
          - concede a  $T_i$  um bloqueio-X sobre  $D$**
      - write( $D$ )**
    - end;**
- **Todos os bloqueios são liberados após o commit ou abort**

# Implementação do bloqueio

- Um **gerenciador de bloqueio** pode ser implementado como um processo separado para o qual as transações enviam solicitações de bloqueio e desbloqueio
- O gerenciador de bloqueio **responde** a uma solicitação de bloqueio enviando uma mensagem de **concessão** de bloqueio (ou uma mensagem pedindo à transação para **reverter**, no caso de um impasse)
- A transação solicitante **espera** até que sua solicitação seja respondida
- O gerenciador de bloqueio mantém uma estrutura de dados chamada **tabela de bloqueio** para registrar bloqueios concedidos e solicitações pendentes
- A tabela de bloqueio normalmente é implementada como uma **tabela de hash** na memória indexada sobre **o nome do item** de dados sendo bloqueado

# Tabela de bloqueio



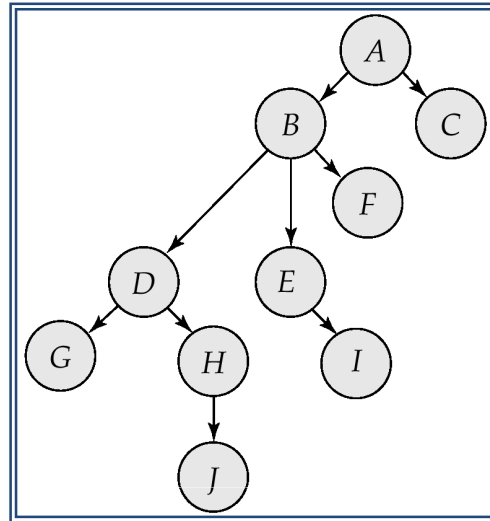
Exemplo de Tabela de Bloqueio

- Retângulos pretos indicam bloqueios concedidos, brancos indicam solicitações aguardando
- I7, I23, etc. representam os itens de dados
- A tabela de bloqueio também registra o tipo de bloqueio concedido ou solicitado
- A nova solicitação é acrescentada ao final da fila de solicitações para o item de dados, e concedida se for compatível com todos os bloqueios anteriores
- As solicitações de desbloqueio resultam na solicitação sendo excluída e solicitações posteriores são verificadas para saber se agora podem ser concedidas
- Se a transação abortar, todas as solicitações aguardando ou concedidas da transação são excluídas

# Protocolos baseados em gráfico

- Os protocolos baseados em gráfico são uma **alternativa** ao bloqueio em duas fases
- Exige conhecimento **anterior** sobre a **ordem** em que os **itens** de dados serão acessados.
- Com essa informação é possível construir protocolos de bloqueios que não são de duas fases mas que **garante** a seriação de conflito.
- Imponha uma ordenação parcial  $\rightarrow$  sobre o conjunto  $D = \{d_1, d_2, \dots, d_h\}$  de todos os itens de dados.
  - Se  $d_i \rightarrow d_j$  então qualquer transação acessando  $d_i$  e  $d_j$  precisa acessar  $d_i$  antes de acessar  $d_j$ .
  - Implica que o novo conjunto  $D$  agora pode ser visto como um gráfico acíclico direcionado, chamado **gráfico de banco de dados**.

# Protocolo de árvore



- O **protocolo de árvore** é um tipo simples de protocolo de gráfico.
- O **primeiro** bloqueio por  $T_i$  pode ser sobre **qualquer** item de dados.
- Subseqüentemente, um dado  $Q$  pode ser **bloqueado** por  $T_i$  somente se o **pai** de  $Q$  for atualmente bloqueado por  $T_i$ .
- Os itens de dados podem ser desbloqueados a qualquer momento.

## Protocolos baseados em gráfico (cont.)

- O desbloqueio pode ocorrer **mais cedo** do que no protocolo de bloqueio em duas fases.
  - tempos de espera mais curtos, aumento na concorrência
  - protocolo é livre de impasse
- Porém, uma transação pode **ter** que bloquear itens de dados que ela **não acessa**.
  - maior sobrecarga de bloqueio, e tempo de espera adicional. Exemplo: uma transação que precisa acessar itens de dados A e J no gráfico precisa bloquear B, D e H
  - diminuição em potencial na concorrência

## Protocolos baseados em estampa de tempo

- Cada **transação** tem uma **estampa de tempo** emitida quando entra no sistema.
  - Se uma transação antiga  $T_i$  tem a estampa de tempo  $TS(T_i)$ , uma nova transação  $T_j$  recebe a estampa de tempo  $TS(T_j)$  de modo que  $TS(T_i) < TS(T_j)$ .
  - O protocolo gerencia a execução concorrente tal que as **estampas de tempo** determinam a **ordem de seriação**.
  - Para garantir esse comportamento, o protocolo mantém para cada dado  $Q$  dois valores de estampa de tempo:
    - $W\text{-timestamp}(Q)$  é a maior estampa de tempo de qualquer transação que executou  $\text{write}(Q)$  com sucesso.
    - $R\text{-timestamp}(Q)$  é a maior estampa de tempo de qualquer transação que executou  $\text{read}(Q)$  com sucesso.
- Esses timestamp são atualizados sempre que uma nova instrução read ou write é executada

## Protocolos baseados em estampa de tempo (cont.)

- O protocolo de ordenação de estampa de tempo garante que quaisquer operações read e write em conflito sejam executadas na ordem de estampa de tempo.
- Suponha que uma transação  $T_i$  emita um **read( $Q$ )**
  1. Se  **$TS(T_i) \geq W\text{-timestamp}(Q)$** , então a operação read é executada, e  $R\text{-timestamp}(Q)$  é definido como o máximo de  $R\text{-timestamp}(Q)$  e  $TS(T_i)$ .
  2. Se  **$TS(T_i) \leq W\text{-timestamp}(Q)$** , então  $T_i$  precisa ler um valor de  $Q$  que já foi modificado. Logo, a operação read é rejeitada, e  $T_i$  é revertida.

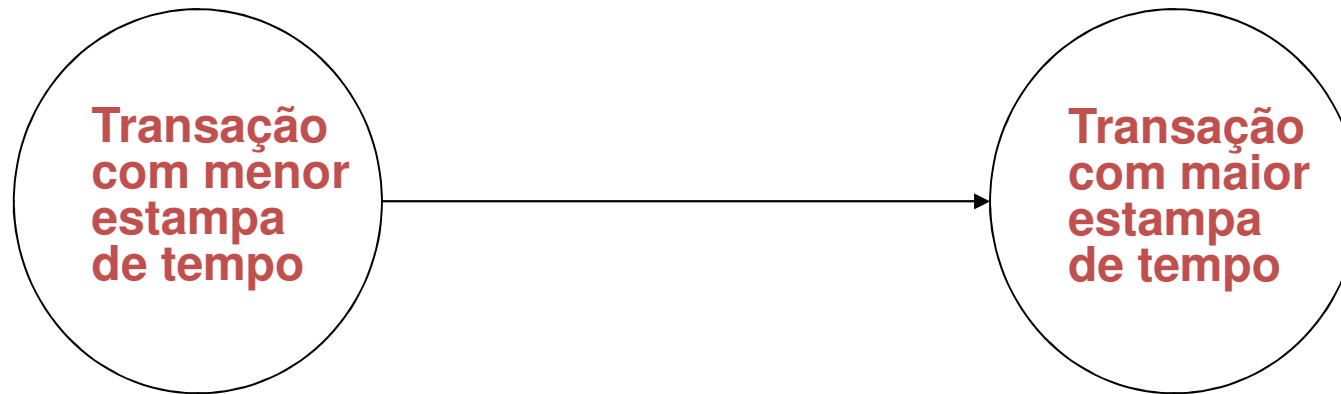


## Protocolos baseados em estampa de tempo (cont.)

- Suponha que a transação  $T_i$  emita **write( $Q$ )**.
- Se  **$TS(T_i) < R\text{-timestamp}(Q)$** , então a operação write é rejeitada, e  $T_i$  é revertido.
- Se  **$TS(T_i) < W\text{-timestamp}(Q)$** , então essa operação write é rejeitada, e  $T_i$  é revertida.
- Caso contrário, a operação write é **executada**, e  $W\text{-timestamp}(Q)$  é definida como  $TS(T_i)$ .

## Exatidão do protocolo de ordenação de estampa de tempo

- O protocolo de ordenação de estampa de tempo garante a seriação, pois todos os arcos no gráfico de precedência são da forma:



**Assim, não haverá ciclos no gráfico de precedência**

- O protocolo de estampa de tempo garante liberdade de impasse, pois nenhuma transação precisa esperar.

■ **Problema** com protocolo de ordenação de estampa de tempo:

- Suponha que  $T_i$  aborte, mas  $T_j$  tenha lido um item de dados escrito por  $T_i$
- Então,  $T_j$  precisa abortar; se  $T_j$  tivesse permitido o commit anterior, o schedule não seria recuperável.
- Além do mais, qualquer transação que tenha lido um item de dados escrito por  $T_j$  precisa abortar
- Isso pode levar ao **rollback em cascata** - ou seja, uma cadeia de rollbacks

■ **Solução:**

- Uma transação é estruturada de modo que suas **escritas** sejam todas realizadas **no final de seu processamento**
- Todas as **escritas** de uma transação formam uma **ação atômica**; e nenhuma transação pode ser executada enquanto uma transação estiver sendo escrita
- Uma transação que aborta é reiniciada com uma nova estampa de tempo

# Regra do write de Thomas

- Versão modificada do protocolo de ordenação de estampa de tempo em que as operações **write** obsoletas podem **ser ignoradas** sob certas circunstâncias.
- Quando  $T_i$  tenta escrever o item de dados  $Q$ , se  $TS(T_i) < W\text{-timestamp}(Q)$ , então  $T_i$  está tentando escrever um valor obsoleto de  $\{Q\}$ . Logo, **em vez de reverter**  $T_i$  como o protocolo de ordenação de estampa de tempo teria feito, essa operação {write} **pode ser ignorada**.
- Caso contrário, esse protocolo é igual ao protocolo de ordenação de estampa de tempo.

# Protocolo baseado em validação

- A execução da transação  $T_i$  é feita em três fases.
  1. **Fase de leitura e execução:** A transação  $T_i$  escreve apenas em variáveis locais temporárias
  2. **Fase de validação:** A transação  $T_i$  realiza um “teste de validação” para determinar se as variáveis locais podem ser escritas sem violar a serialização.
  3. **Fase de escrita:** Se  $T_i$  for validada, as atualizações são aplicadas ao banco de dados; caso contrário,  $T_i$  é revertida.
- As três fases da execução simultânea de transações podem ser intercaladas, mas cada transação precisa passar pelas três fases nessa ordem.
- Também chamado controle de concorrência **otimista**, pois a transação é executada totalmente na esperança de que tudo correrá bem durante a validação

## Protocolo baseado em validação (cont.)

- Cada transação  $T_i$  possui 3 estampas de tempo
  - $\text{Start}(T_i)$ : momento em que  $T_i$  iniciou sua execução
  - $\text{Validation}(T_i)$ : momento em que  $T_i$  entrou em sua fase de validação
  - $\text{Finish}(T_i)$  : momento em que  $T_i$  concluir sua fase de escrita
- A ordem de serialização é determinada pela estampa de tempo dada na hora da validação, para aumentar a concorrência.
- Esse protocolo é útil e oferece maior grau de concorrência se a **probabilidade de conflitos for baixa**.

# Schedule produzido por validação

## ■ Exemplo de schedule produzido usando validação

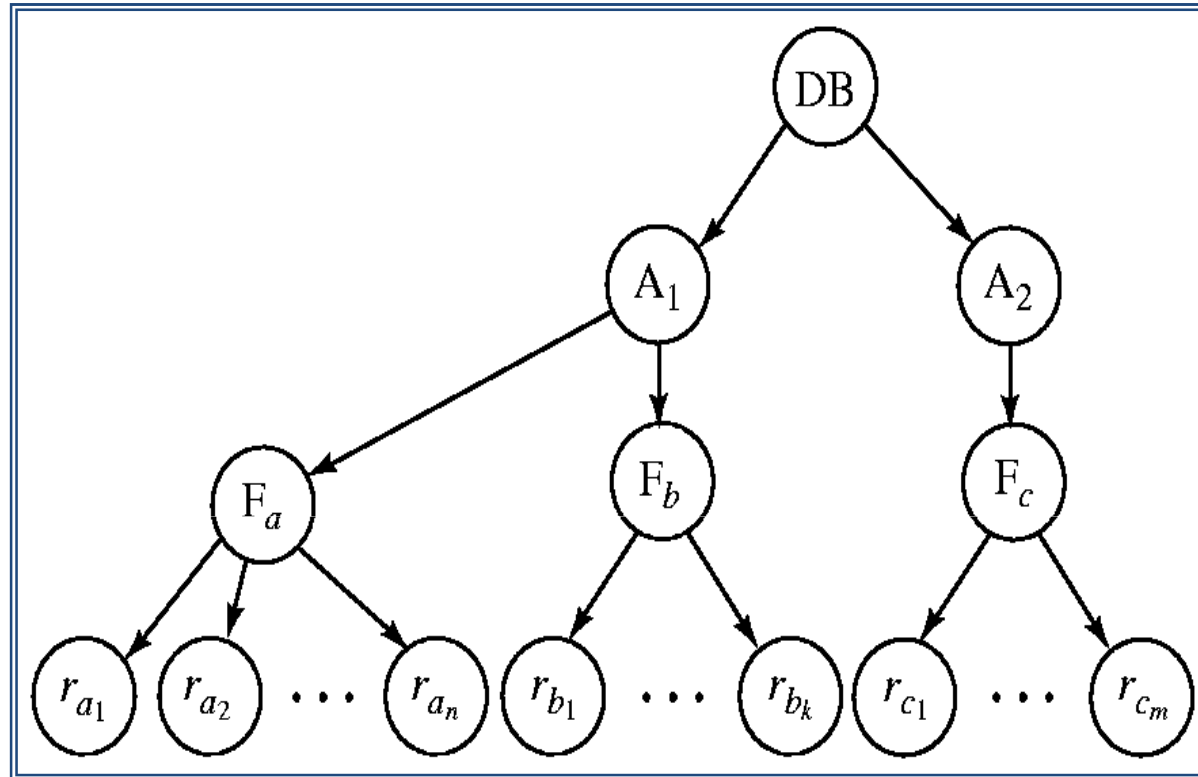
$T_{14}$	$T_{15}$
<b>read</b> ( $B$ )	<b>read</b> ( $B$ ) $B:- B-50$
<b>read</b> ( $A$ ) ( <i>validate</i> ) <b>display</b> ( $A+B$ )	<b>read</b> ( $A$ ) $A:- A+50$
	( <i>validate</i> ) <b>write</b> ( $B$ ) <b>write</b> ( $A$ )

# Granularidades múltiplas

- Nos esquemas descritos foi utilizado item de dado **individual** como unidade de sincronismo
- Em algumas circunstancias pode ser interessante **agrupar dados** e trata-los como uma unidade individual
- Podem ser representadas graficamente como **árvore**
- Quando uma transação bloqueia um nó na árvore *explicitamente*, ela *implicitamente* bloqueia os descendentes do nó no mesmo modo.
- Granularidade do bloqueio:
  - *granularidade menor* (mais baixo na árvore): alta concorrência
  - *granularidade maior* (mais alto na árvore): baixa concorrência



## Exemplo de hierarquia de granularidade



- O nível mais alto na hierarquia de exemplo é o banco de dados inteiro.
- Os níveis abaixo são do tipo *área*, *arquivo* e *registro*, nessa ordem.

## Esquema de bloqueio com granularidade múltipla

- A transação  $T_i$  pode bloquear um nó  $Q$ , usando as seguintes regras:
  1. A matriz de compatibilidade de bloqueio precisa ser observada.
  2. A raiz da árvore precisa ser bloqueada primeiro, e pode ser bloqueada em qualquer modo.
  3. Um nó  $Q$  pode ser bloqueado por  $T_i$  em modo S ou IS somente se o pai de  $Q$  for atualmente bloqueado por  $T_i$  no modo IX ou IS.
  4. Um nó  $Q$  pode ser bloqueado por  $T_i$  no modo X, SIX ou IX somente se o pai de  $Q$  estiver atualmente bloqueado por  $T_i$  no modo IX ou SIX.
  5.  $T_i$  só pode bloquear um nó se ele não tiver previamente desbloqueado qualquer nó (ou seja,  $T_i$  é em duas fases).
  6.  $T_i$  só pode desbloquear um nó  $Q$  se nenhum dos filhos de  $Q$  estiver atualmente bloqueado por  $T_i$ .
- Observe que os bloqueios são adquiridos na ordem da raiz à folha, enquanto são liberados na ordem da folha à raiz.

# Esquemas multiversão

- Esquemas **multiversão** mantêm versões antigas do item de dados para aumentar a concorrência.
- Cada write bem sucedido resulta na criação de uma nova versão do item de dados escrito.
- Usa **estampas** de tempo para **rotular** versões.
- Quando uma operação  $\text{read}(Q)$  for emitida, selecione uma **versão apropriada** de  $Q$  com base na estampa de tempo da transação, e retorne o valor da versão selecionada.

# Bloqueio em duas fases multiversão

- Diferencia entre transações somente leitura e transações de atualização
- *Transações **de atualização*** adquirem bloqueios de leitura e escrita, e mantêm todos os bloqueios ativos até o final da transação. Ou seja, transações de atualização seguem o bloqueio rigoroso em **duas fases**.
- *Transações **somente de leitura*** recebem uma **estampa** de tempo lendo o valor atual de ts-counter antes de começar a execução; elas seguem o protocolo de ordenação de estampa de tempo multiversão para realizar leituras.

# Tratamento de impasse

- Considere as duas transações a seguir:

$T_1$ :	write ( $X$ )	$T_2$ :	write( $Y$ )
	write( $Y$ )		write( $X$ )

- Schedule com impasse

$T_1$	$T_2$
<b>lock-X</b> on $X$ write ( $X$ )  wait for <b>lock-X</b> on $Y$	<b>lock-X</b> on $Y$ write ( $X$ ) wait for <b>lock-X</b> on $X$

# Tratamento de impasse

- O sistema está em impasse se houver um conjunto de transações de modo que cada transação no conjunto esteja esperando por outra transação no conjunto.
- Protocolos de *prevenção de impasse* garantem que o sistema *nunca* entrará em um estado de impasse.
- Algumas estratégias de prevenção:
  - Exige que cada transação bloqueie todos os seus itens de dados **antes** de iniciar a execução (pré-declaração).

# Mais estratégias de prevenção de impasse

- Os seguintes esquemas usam estampas de tempo de transação para fins de prevenção de impasse apenas.
- Esquema **esperar-morrer — não preemptivo**
  - a transação **mais antiga** pode esperar que a mais recente libere o item de dado.
  - as transações **mais recentes** nunca esperam pelas mais antigas; em vez disso, elas são revertidas.
  - uma transação pode morrer várias vezes antes de adquirir o item de dados necessário
- Esquema **ferir-esperar — preemptivo**
  - a transação mais antiga **ferre** (força o rollback) da transação mais nova em vez de esperar por ela. Transações mais novas **podem** esperar pelas mais antigas.
  - pode ter menos rollbacks que o esquema *esperar-morrer*.

# Prevenção de impasse (cont.)

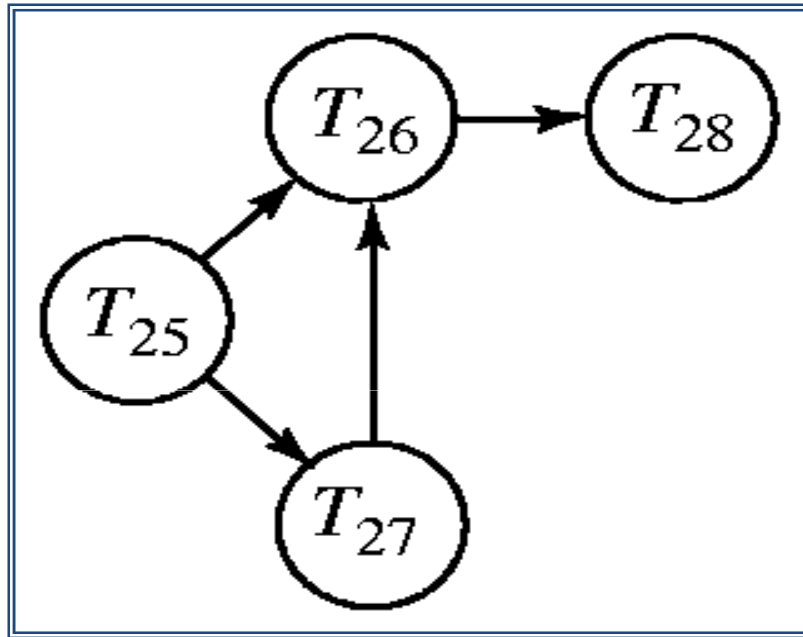
- Nos esquemas *esperar-morrer* e *ferir-esperar*, uma transação revertida é reiniciada com sua estampa de tempo original. Transações mais antigas, assim, têm precedência em relação às mais novas, e a inanição é evitada.
- Esquemas baseados em tempo limite:
  - uma transação só espera por um bloqueio por um **certo tempo** especificado. Depois disso, a espera esgota o tempo limite e a transação é revertida.
  - simples de implementar; mas a inanição é possível. Também difícil de determinar um bom valor do intervalo de tempo limite.



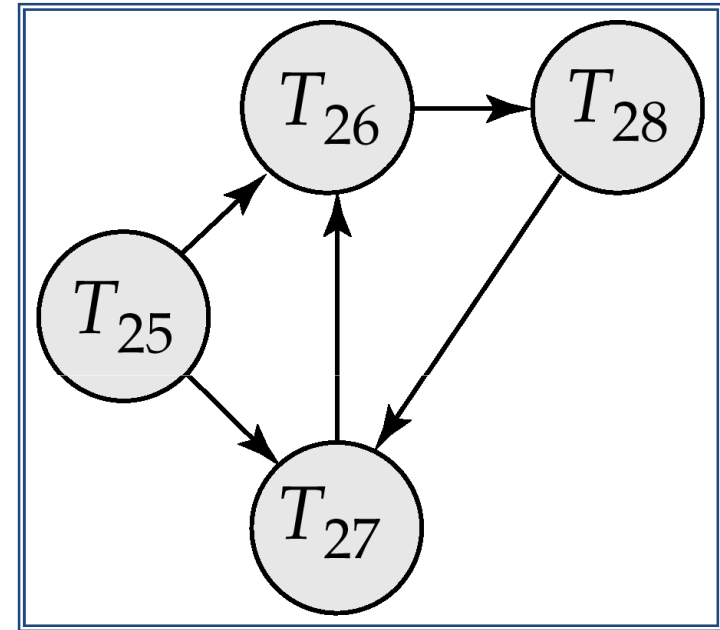
# Detecção de impasse

- Os impasses podem ser descritos como um gráfico de *espera*, que consiste em um par  $G = (V, E)$ ,
  - $V$  é um conjunto de vértices (todas as transações no sistema)
  - $E$  é um conjunto de arestas; cada elemento é um par ordenado  $T_i \rightarrow T_j$ .
- Se  $T_i \rightarrow T_j$  está em  $E$ , então existe uma aresta dedicada de  $T_i$  para  $T_j$ , implicando que  $T_i$  está esperando que  $T_j$  libere um item de dados.
- Quando  $T_i$  solicita um item de dados atualmente mantido por  $T_j$ , então a aresta  $T_i \rightarrow T_j$  é inserida no gráfico de espera. Essa aresta só é removida quando  $T_j$  não está mais mantendo um item de dados necessário por  $T_i$ .
- O sistema está em um estado de impasse se e somente se o gráfico de espera **tiver um ciclo**. Precisa invocar um algoritmo de detecção de impasse periodicamente para procurar ciclos.

## Detecção de impasse (cont.)



**Gráfico de espera sem um ciclo**



**Gráfico de espera com um ciclo**

# Recuperação de impasse

## ■ Quando o impasse for detectado:

- Alguma transação terá que ser revertida (uma vítima) para romper o impasse. Selecione essa transação como a vítima que terá o menor custo.
- Rollback - determine até onde reverter a transação
  - ▶ **Rollback total**: Aborte a transação e depois reinicie-a.
  - ▶ Mais eficiente reverter a transação somente até o ponto **necessário** para romper o impasse.
- A **inanição** acontece se a mesma transação sempre for escolhida como vítima. Inclua o número de rollbacks no fator de custo para evitar inanição

# Operações de inserção e exclusão

## ■ Se o bloqueio em duas fases for usado:

- Uma operação **delete** só pode ser realizada se a transação tiver um bloqueio exclusivo sobre a tupla a ser excluída.
- Uma transação que **insere** uma nova tupla recebe um bloqueio no modo X sobre a tupla

# Operações de inserção e exclusão

- Inserções e exclusões podem levar ao fenômeno de **fantasma**.
  - Uma transação que varre uma relação (por exemplo, encontrar todas as contas em Perryridge) e uma transação que insere uma tupla na relação (por exemplo, inserir uma nova conta em Perryridge) **podem entrar em conflito** apesar de não acessarem qualquer tupla em comum.
  - Se apenas bloqueios de tupla forem usados, schedules não seriáveis podem acontecer: a transação de varredura pode não ver a nova conta, embora possa ser seriada antes da transação de inserção.

# Bibliografia

- Sistemas de Banco de Dados (Cap. 16).  
Abraham Silberchatz, Henry F. Korth, S  
Sudarshan. 5a Ed. Elsevier, 2006.