Sistemas de Gerência de Banco de Dados

Transações

Agenda

- Estruturas de armazenamento
- Transação
- ACID
- Controle de interações
- Serializabilidade
- Gerenciamento de concorrência
- Transações concorrentes
- Protocolo de bloqueio
- Proteção contra impasse (*deadlock*)
- Outros aspectos sobre controle de concorrência
- Recuperação de falhas
- Esquemas baseados em *log*
- Paginação de sombra (*shadow page*)
- Modificações adiadas
- Algoritmos de recuperação modernos
- Técnicas de recuperação modernas

Relembrando ... Estruturas de armazenamento

- Tipos de armazenamento:
    1. Volátil – exemplo: memória RAM – dados podem ser perdidos quando há falhas
    2. Não volátil – exemplo: disco rígido - dados não são perdidos quando há falha, mas podem ser corrompidos ou perdidos por quebras de disco
    3. Armazenamento estável –
        - on-line - discos rígidos espelhados (RAID), por redundância de armazenamento
        - off-line (arquivamento) - várias cópias em fita e locais fisicamente seguros

Transação – Visão geral

- Unidade lógica de execução
- Formada por conjunto de operações
- Pode
    - acessar e consultar diversos itens de dados
    - atualizar vários itens de dados

Transação

*Uma **transação** é uma **unidade de execução de programa** que **acessa** e pode **atualizar vários itens de dados**.*

- Uma transação precisa partir de um banco de dados **consistente**.
- Durante a execução da transação, o banco de dados pode estar **temporariamente inconsistente**.
- Quando a transação é completada com sucesso (confirmada), o banco de dados precisa estar **consistente**.

Continuação...

- Após a **confirmação** da transação, as mudanças que ela fez no banco de dados **persistem**, mesmo se houver falhas de sistema.
- Várias **transações** podem ser **executadas** em **paralelo**.

- Dois problemas principais para resolver:

1) **Falhas** de **vários tipos**, como falhas de *hardware* e falhas de sistema

2) **Execução simultânea** de múltiplas transações

Garantias sobre transações

- Sistema precisa
  - garantir execução adequada das transações mesmo que ocorram falhas
  - garantir que transação seja totalmente executada ou nenhuma parte dela
  - gerenciar execução simultânea de transações evitando inconsistências

Execuções simultâneas

- Proporciona:
  - melhora no *throughput*
    - número de transações em um espaço de tempo, p.ex. transações por segundo
  - redução do tempo de espera

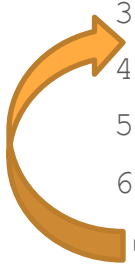
Transações e propriedades ACID

Atomicidade	▪ operações de uma transação devem refletir no BD ou nenhuma delas deve acontecer ▪ uma falha não pode deixar BD parcialmente alterado ▪ ou seja, ou todas as operações da transação são refletidas corretamente no banco de dados ou nenhuma delas é
Consistência	▪ BD consistente antes da execução da transação deve permanecer assim ao final do processamento ▪ ou seja, a execução de uma transação isolada preserva a consistência do banco de dados
Isolamento	▪ transações simultâneas devem ser isoladas umas das outras, ou seja, transação é analisada como se não houvesse concorrência ▪ isto é, embora várias transações possam ser executadas simultaneamente, cada transação precisa estar desinformada das outras transações que estão sendo executadas ao mesmo tempo.
Durabilidade	▪ se transação é confirmada, atualizações realizadas não serão perdidas, mesmo ocorrendo falha no sistema ▪ ou seja, depois que uma transação for completada com sucesso, as mudanças que ela fez ao banco de dados persistem, mesmo que existam falhas no sistema.

Exemplo

- **Transação para transferir R$ 50 da conta A para a conta B:**

```
1. read(A)
2. A := A − 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

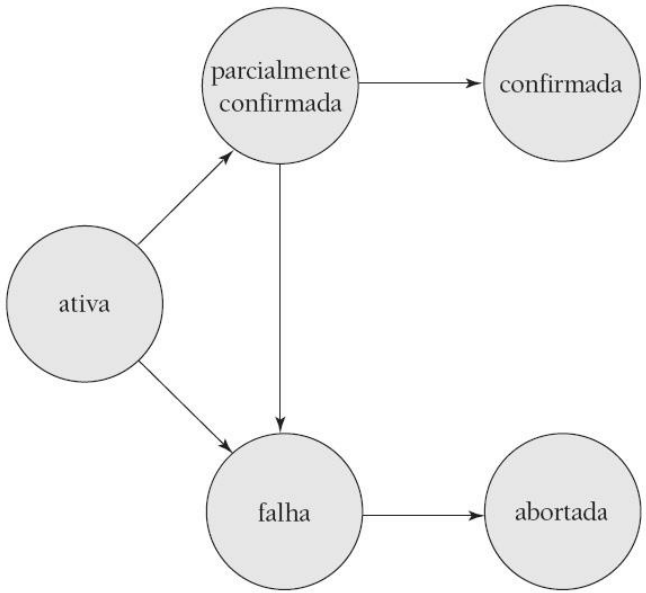


- **Requisito de Atomicidade** — Se a transação falhar após a etapa 3 e antes da etapa 6, o sistema deve garantir que suas atualizações não sejam refletidas no DB, ou uma inconsistência irá resultar.
- **Requisito de Consistência** — Soma de A + B é inalterada pela execução da transação.

Exemplo (cont.)

- **Requisito de Isolamento** — Se entre as etapas 3 e 6, outra transação receber permissão de acessar o banco de dados parcialmente atualizado, ele verá um banco de dados inconsistente (a soma *A* + *B* será menor do que deveria ser).
  - Isso pode ser trivialmente assegurado executando **transações serialmente**, ou seja, **uma após outra**.
  - Mas, executar **múltiplas transações simultaneamente** oferece vantagens significativas, como veremos mais adiante.
- **Requisito de Durabilidade** — Quando o usuário é notificado de que a transação está concluída (ou seja, a transação dos R$ 50 ocorreu), as atualizações no banco de dados pela transação precisam estar persistidas.

Estados da Transação



1. **Ativa** – É o estado inicial (`begin transaction`), onde a transação permanece enquanto está executando;
2. **Parcialmente confirmada** – Depois que a instrução final foi executada;
3. **Falha** – Depois da descoberta de que a execução normal não pode mais prosseguir;
4. **Abortada** – Depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação.
   - **Duas opções** após ter sido abortada:
     1) Reiniciar a transação - pode ser feito apenas se não houver qualquer erro lógico interno;
     2) Excluir a transação.
5. **Confirmada** – Após o término bem sucedido (`end transaction`).

Controle da interação entre as transações

- Necessidade de controlar transações simultâneas para preservar consistência
- Execução serial de transações garante consistência
- *Schedule* contém operações que afetam execução concorrente
    - Exemplos de operações internas abstraídas: *read* e *write*

Componente de gerenciamento de concorrência

- Responsável pelas técnicas de controle de concorrência
- Técnicas podem incluir:
    - bloqueio, ordenação por *timestamp* e isolamento instantâneo (*snapshot*)

- Atenção com DML (*update*, *insert* e *delete*) para
    - garantir execução simultânea correta
    - evitar fantasma (*ghost*) - consultas a dados sem que DML tenha finalizado

Gerenciamento de controle de concorrência

- Existem níveis de isolamento mais fracos que a serializabilidade
  - com melhoria do desempenho
  - através de menos restrições de concorrência
  - mas com riscos de inconsistência, embora considerados aceitáveis

Execuções Simultâneas

- **Várias transações** podem ser executadas **simultaneamente**.
- Vantagens:

1) **Melhor utilização** do **processador** e do **disco**, levando a um melhor ***throughput*** de transação (quantidade de processamentos em um determinado tempo) :

   Exemplo: uma transação pode usar a CPU enquanto outra está lendo ou escrevendo no disco

2) **Tempo médio de resposta reduzido** para transações:

   Exemplo: as transações curtas não precisam esperar atrás das longas

- **Esquemas de controle de concorrência** – mecanismos para obter isolamento; ou seja, para controlar a interação entre as transações concorrentes a fim de evitar que elas destruam a consistência do banco de dados

  Estudaremos, na próxima aula, Controle de Concorrência

Serializabilidade

- Garantia de que
  - *schedule* de concorrência = *schedule* de transações executas em série e ordenadas
- Garantida por esquemas de *controle de concorrência*
- *Schedules* precisam ter capacidade de recuperação (desfazimento)
  - se transação X vê efeitos de transação Y e,
  - se transação Y aborta,
  - então transação X também aborta

Serializabilidade (continua)

- *Schedules* não devem estar em cascata, pois
  - pode ser necessário abortar uma transação e
  - esta transação pode não conseguir abortar em cascata outras transações dependentes
- Possível evitar cascata com transações somente lendo dados confirmados

Escalonamentos *(Schedules)* - sequências de instruções

- ***Schedule*** – sequências de instruções que especificam a ordem cronológica em que as instruções das transações simultâneas são executadas:
  - Um ***schedule*** para um **conjunto de transações** precisa **consistir** em todas as **instruções** dessas **transações**;
  - Precisam **preservar** a **ordem** em que as **instruções** aparecem em cada **transação individual**.

☑ Uma transação que **completa com sucesso** sua execução terá uma instrução ***commit*** como a **última instrução** (será omitida se for óbvia);

✕ Uma transação que **não completa com sucesso** sua execução terá instrução ***abort*** (***rollback***) como a última instrução (será omitida se for óbvia).

Escalonamento (*Schedule*) 1

- Suponha que:
  - T_1 transfere R$ 50 de ***A*** para ***B*** e
  - T_2 transfere 10% do saldo de ***A*** para ***B***.
- A seguir, temos um ***schedule*** serial em que:
  - T_1 é seguido de T_2

Exemplo:

T_1 estado inicial: A=200, B=100
estado final: A=150, B=150

T_2 estado inicial: A=150, B=150
estado final: A=135, B=165

soma A + B = 300

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Escalonamento (*Schedule*) 2

- Sejam T_1 e T_2 as transações definidas anteriormente.
- O ***schedule*** a seguir não é um ***schedule*** serial, mas é **equivalente** ao *Schedule* 1.

Observação: não é um schedule serial em função de terem sido separados os blocos originais, em relação ao schedule 1 (anterior), mas são equivalentes pois o resultado final será o mesmo

Exemplo:
Inicial: A=200, B=100
Final: A=135, B=165

Soma A + B = 300

T_1	T_2
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)	
	read(B)
	$B := B + temp$
	write(B)

Escalonamento (*Schedule*) 3

- O seguinte ***schedule*** concorrente **não preserva** o valor da soma A + B.

Nos exemplos anteriores (Schedule 1 e 2):
Inicial: A=200, B=100
Final: A=135, B=165
Soma: A + B = 300

Exemplo:
Inicial: A=200, B=100
Final: A=150, B=120

Soma: A + B = 270

haverá conflito em
write(A) e write(B)

T_1	T_2
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)	
	$B := B + temp$
	write(B)

Serialização

- **Suposição básica** – Cada **transação preserva** a **consistência** do DB;
- Portanto, ***a execução serial de um conjunto de transações preserva a consistência do banco de dados;***
- Um ***schedule*** (possivelmente simultâneo, concorrente) é **serializável (*serializable*)** se for equivalente a um ***schedule* serial**.
    - Diferentes formas de **equivalência** de *schedule* ensejam as noções de:
        1) **Serialização** de **conflito**
        2) **Serialização** de ***view* (visão)**

- Ignoramos as operações, exceto ***read*** e ***write***, e consideramos que as transações podem realizar **cálculos arbitrários** sobre dados em ***buffers*** locais entre as operações de ***read*** e ***write***.
- Nossos ***schedules* simplificados** consistem apenas em instruções ***read*** e ***write***.

Instruções conflitantes

- As instruções I_i e I_j das transações T_i e T_j respectivamente, estão em conflito se e somente se algum item Q acessado por I_i e por I_j e pelo menos uma destas instruções escreveram Q.

  1. I_i = read(Q), I_j = read(Q). I_i e I_j não estão em conflito
  2. I_i = read(Q), I_j = write(Q). Estão em conflito
  3. I_i = write(Q), I_j = read(Q). Estão em conflito
  4. I_i = write(Q), I_j = write(Q). Estão em conflito

- Intuitivamente, um conflito entre I_i e I_j força uma ordem temporal (lógica) entre eles.

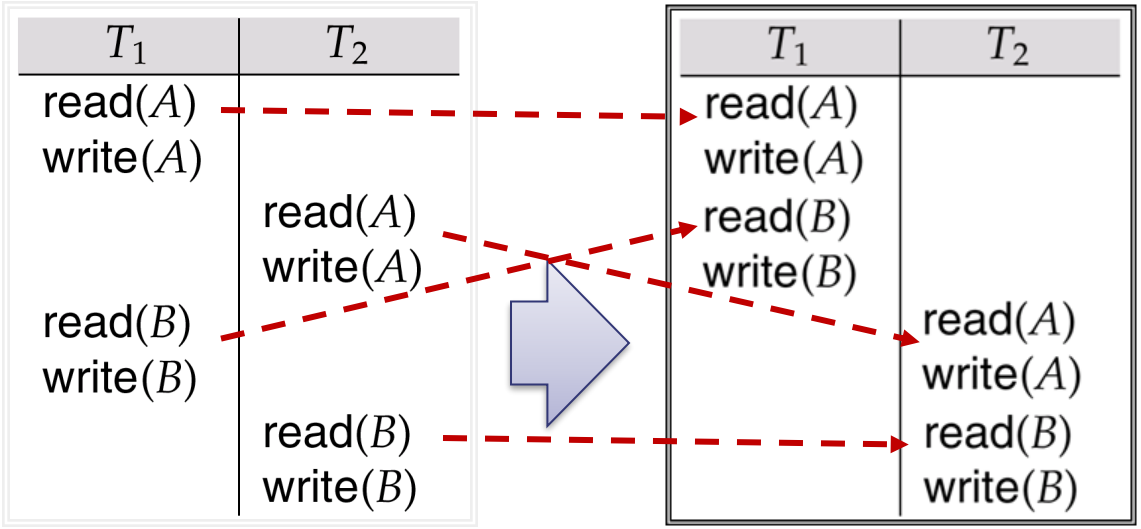
- Se I_i e I_j são consecutivos em um *schedule* e não entram em conflito, seus resultados permanecem inalterados mesmo se tiverem sido trocados no *schedule*.

Serialização de Conflito

- Se um ***schedule* S** puder ser transformado em um ***schedule* S'** por uma **série** de **trocas** de **instruções** não conflitantes, dizemos que **S** e **S'** são **equivalentes em conflito**;
- Dizemos que um ***schedule* S** é **serial de conflito** se ele for equivalente em **conflito** a um ***schedule* serial**.
- "Dado um ***schedule* S concorrente**, este será **serial de conflito** se por uma sucessão de trocas não conflitantes, produzir um ***schedule* S' serial**" (Prof. Gilberto).

Serialização de Conflito (cont.)

- O ***schedule* 1 (S)** pode ser transformado em **schedule 2 (S')** (ver próximo slide), onde T_2 segue T_1, por uma série de trocas de instruções não conflitantes.
- Portanto, o ***schedule* 2 (S')** é **serial de conflito** em relação ao ***schedule 1 (S)***.



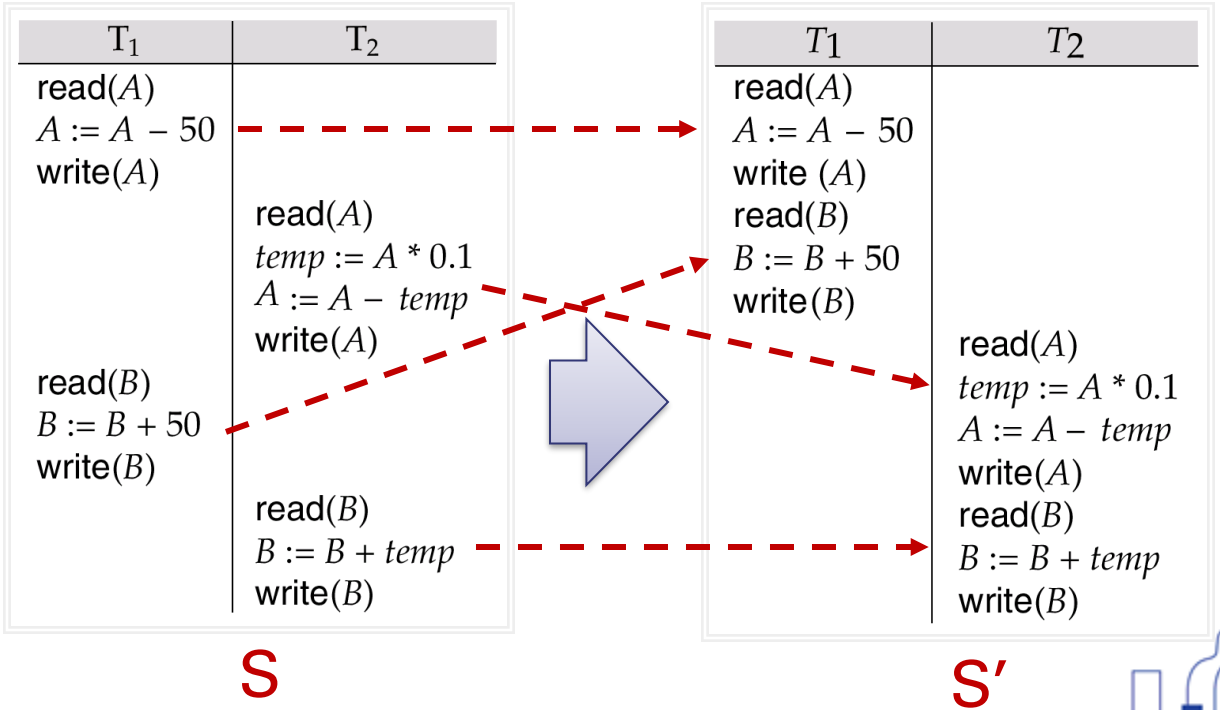
S e S' são equivalentes

Exemplo:

T_1 estado inicial: A=200, B=100
estado final: A=150, B=150

T_2 estado inicial: A=150, B=150
estado final: A=135, B=165

soma A + B = 300



Outro exemplo de serialização equivalente

- O ***schedule 1*** abaixo produz o mesmo resultado do ***schedule*** serial 2 < T_1, T_5 >

1

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Exemplo:
Inicial: A=300, B=200
Final: A=260, B=240
Soma: A + B = 500

2

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B - 10$
	write(B)
	read(A)
	$A := A + 10$
	write(A)

Exemplo:
Inicial: A=300, B=200
Final: A=260, B=240
Soma: A + B = 500

Serialização de *View* (Visão)

- Sejam **S** e **S´** dois ***schedules*** com o mesmo conjunto de transações.
- **S** e **S´** são **equivalentes** em ***view*** se três condições forem satisfeitas:
  1. Para cada item de dados **Q**, se a transação T_i **ler** o valor inicial de **Q** no ***schedule* S**, então, T_i precisa, no ***schedule* S'**, também **ler** o valor inicial de **Q**.
  2. Para cada item de dados **Q**, se a transação T_i executar **read(Q)** no ***schedule* S**, e se esse valor foi produzido pela transação T_j (se houver), então a transação T_i, no ***schedule* S'**, também precisa **ler** o valor de **Q** que foi produzido pela transação T_j.
  3. Para cada item de dados **Q**, a transação (se houver) que realiza a operação **write(Q)** final no schedule **S** precisa realizar a operação **write(Q)** final no ***schedule* S'**.

Serialização de *View* (Visão) – cont…

- Como podemos ver, a equivalência em ***view*** também é baseada unicamente em operações de **read** e **write** isolados.
- Um ***schedule* S** é **serial de *view*** se ele for equivalente em ***view*** a um ***schedule* serial**;
- Em resumo, todo ***schedule* serial de conflito** também é **serial de *view***;

Serialização de *View* (cont.)

T_1	T_2	T_3
		read(x)
	read(x)	
		write(x)
read(x)		
write(x)		

T_2	T_3	T_1
read(x)		
	read(x)	
	write(x)	
		read(x)
		write(x)

a. Leituras iniciais: OK
b. Há conflito W-R: OK
c. Escritas finais: OK

Serialização de *View* (cont.)

S1

T_1	T_2
$READ(X)$	
$X = X - 100$	
$WRITE(X)$	
$READ(Y)$	
$Y = Y + 100$	
$WRITE(Y)$	
	$READ(X)$
	$X = X + 200$
	$WRITE(X)$

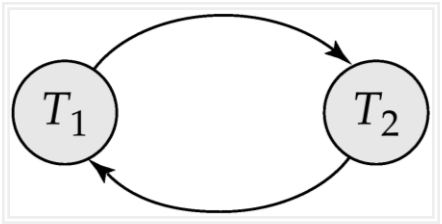
S2

T_1	T_2
$READ(X)$	
$X = X - 100$	
	$READ(X)$
	$X = X + 200$
$WRITE(X)$	
$READ(Y)$	
	$WRITE(X)$
$Y = Y + 100$	
$WRITE(Y)$	

S2 não é equivalente por visão a S1, em função das leituras de X em T2 serem diferentes, em S1 a T2 está lendo o valor produzido por T1, enquanto que em S2 a T2 está lendo o valor original.

Testando a serialização

- Considere um ***schedule*** de um conjunto de transações $T_1, T_2, ..., T_n$
- **Gráfico de precedência** —
  - Um gráfico direcionado onde os **vértices** são as **transações** (nomes)
- Desenhamos um arco de T_i até T_j se a **transação entra em conflito** e T_i acessou o item de dados no qual surgiu o conflito anteriormente
- Podemos rotular o arco pelo item que foi acessado
- Exemplo 1



Exemplo
Schedule A

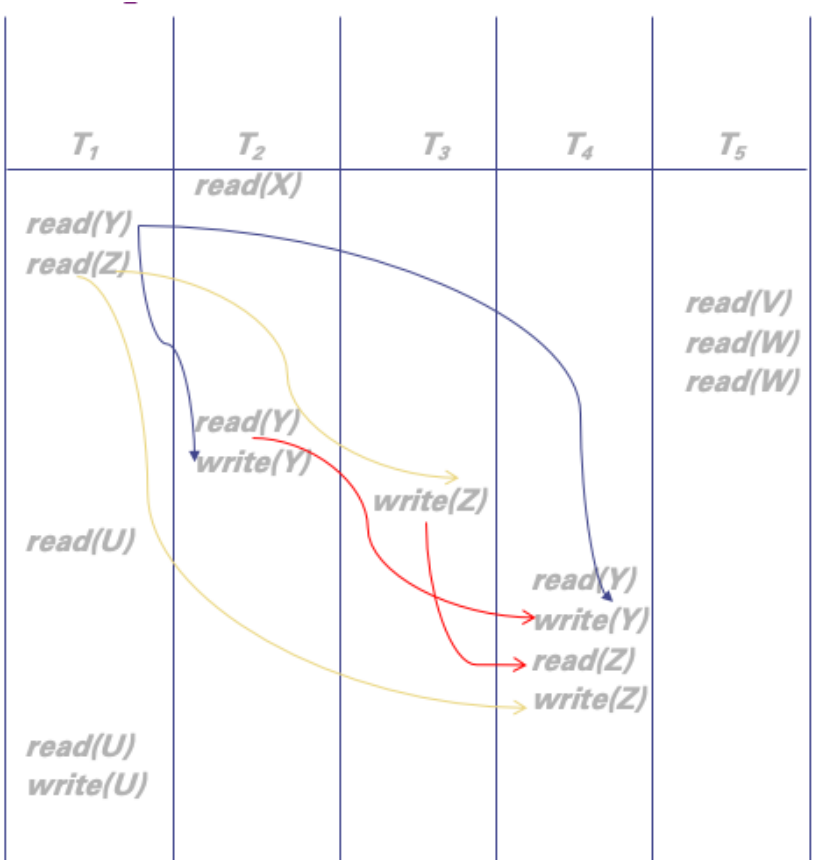
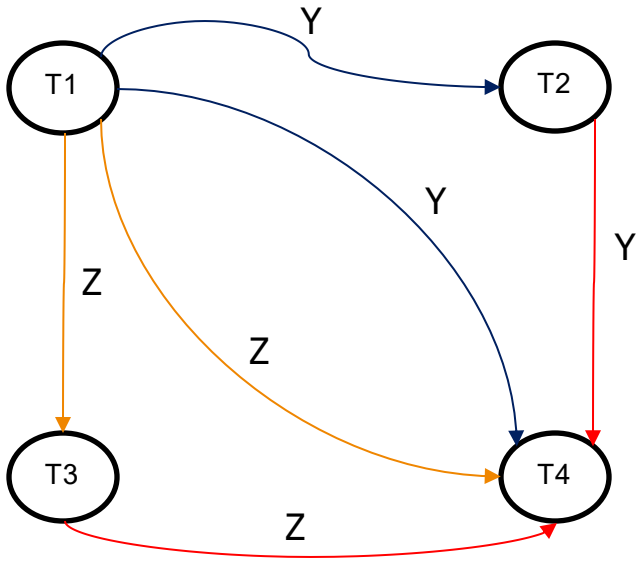


Gráfico de Precedência para o *Schedule* A



Veja os nodos que acessaram o recurso antes (read / write)
Observe que T_5 não tem recursos compartilhados com outras transações

Teste serialização de conflito

- Um ***schedule*** é **serial de conflito** se e somente se seu gráfico de precedência for **acíclico** (não circular)

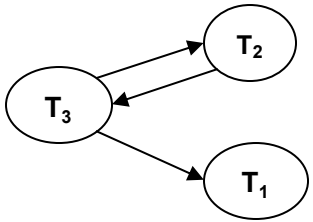
- Se o gráfico de precedência for **acíclico**, a ordem de serialização pode ser obtida por meio da classificação topológica do gráfico
    - Essa é a **ordem linear** com a **ordem parcial** do grafo.

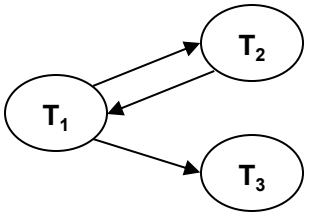
Teste serialização de conflito

T_1	T_2	T_3
		read(x)
	read(x)	
		write(x)
read(x)		
write(x)		

T_1	T_2	T_3
read(x)		
	write(x)	
write(x)		
		write(x)

Como ficaria o grafo com base nos schedules acima?





Facilidade de Recuperação

É necessário tratar as **falhas de transação** nas transações **executadas simultaneamente**.

- ***Schedule* recuperável** — Se uma transação T_j lê um item de dados anteriormente escrito por uma transação T_i, então a operação ***commit*** de T_i aparece antes da operação ***commit*** de T_j.

- O ***schedule*** abaixo **não é recuperável** se T_9 for confirmada imediatamente após o read; no caso de T_8 abortar (e feito ***rollback***, portanto), T_9 teria lido (e possivelmente mostrado ao usuário) um estado inconsistente.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- Portanto, o banco de dados precisa garantir que ***schedules*** sejam **recuperáveis**.

Facilidade de Recuperação

- ***Rollback*** em cascata – Uma única falha de transação leva a uma série de ***rollbacks*** de transação.
- Considere o seguinte ***schedule*** onde nenhuma das transações ainda foi confirmada (portanto, o ***schedule*** é recuperável):

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- Se **T_{10}** falhar, **T_{11}** e **T_{12}** também precisam ser revertidos
- Pode chegar a desfazer uma quantidade de trabalho significativa

Facilidade de Recuperação

- ***Schedules*** não em cascata — *rollbacks* em cascata não podem ocorrer;
    - Para cada par de transações T_i e T_j tal que T_j leia um item de dados escrito anteriormente por T_i, a operação ***commit*** de T_i apareça antes da operação ***read*** de T_j.
- Todo ***schedule*** não em cascata também é recuperável
- É desejável restringir os **schedules** aos não em cascata

Aspectos de Implementação

- Um banco de dados precisa fornecer um mecanismo que garanta que todos os *schedules* possíveis sejam **seriais de conflito** ou de ***view***, e sejam **recuperáveis** e, preferivelmente, **não em cascata**
- Uma política em que apenas uma transação pode ser executada de cada vez gera ***schedules* seriais**, mas fornece um **menor grau de concorrência**
- Os esquemas de controle de concorrência conciliam entre a quantidade de concorrência permitida e a quantidade de sobrecarga a que ficam sujeitos

Testes de Controle e de Serialização

- Testar a serialização de um *schedule* após ele ter sido executado é um pouco **tarde demais**!
- Objetivo – Desenvolver os protocolos de controle de concorrência que garantirão a capacidade de serialização.
  - Eles normalmente não examinam o gráfico de precedência enquanto está sendo criado;
  - Em vez disso, um protocolo imporá uma regra que evita *schedules* não serializáveis.
  - Estudaremos em Controle de Concorrência.
- Os testes para a serialização ajudam a entender por que um protocolo de controle de concorrência está correto

Definição de Transação na SQL

- A linguagem de manipulação de dados precisa incluir uma construção para especificar o conjunto de ações que compõem uma transação
- **Em SQL, uma transação começa implicitamente**
- Uma transação na SQL termina por:
  - Commit work confirma a transação atual e inicia uma nova transação
  - Rollback work faz com que a transação atual seja abortada

Níveis de consistência na SQL

1. ***Serializable*** — Padrão;
2. ***Repeatable read*** — Apenas registros confirmados podem ser lidos;
    - *reads* repetidos do mesmo registro precisam retornar o mesmo valor.
    - entretanto, uma transação pode não ser **serializável** — ela pode encontrar alguns registros inseridos por uma transação mas não encontrar outros;
3. ***Read committed*** — Apenas registros confirmados podem ser lidos, mas *reads* sucessivos do registro podem retornar valores diferentes (mas confirmados);
4. ***Read uncommitted*** — Mesmo registros não confirmados podem ser lidos;

* graus de consistência mais baixos são úteis para coletar informações aproximadas sobre o banco de dados.

Concorrência

Visão geral sobre transações concorrentes

- Transações simultâneas → podem não preservar **coerência** dos dados
- Controle da concorrência por esquemas de bloqueio que:
  - adia uma operação ou
  - aborta transação onde houve falha de operação

Controle de transações concorrentes

- Esquemas de controle mais comuns:
    1. Ordenação por *timestamp* (marcações por data e hora)
    2. Validação
    3. Múltipla versão

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 a interação entre transações concorrentes**: esse controle é alcançado por meio de uma série de mecanismos, os quais unidos, formam o **controle de concorrência**;
- A base dos esquemas de concorrência 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 os sistemas não incorrem em falhas.

Protocolo de bloqueio

- Conjunto de regras → indica quando transação bloquear e desbloquear dado
- Protocolo de bloqueio em duas fases:
  - permite transação bloquear novo item somente se não tiver desbloqueado nenhum outro
  - assegura serializabilidade
  - não assegura ausência de impasse (*deadlock*)

Proteção contra impasse (deadlock)

- Muitos protocolos de bloqueio não protegem contra impasses (*deadlocks*)
- Se *deadlock* não puder ser evitado, sistema deverá:
    - detectar e se recuperar do impasse
    - revertendo tantas transações quanto necessárias para quebrar o impasse

Proteção contra impasse (*deadlock*)

- Possíveis soluções
  - ordenação dos itens de dados e solicitação de bloqueios em sequência consistente
  - preempção e *rollbacks* de transação – com *timestamps* para decidir esperar ou reverter
  - construção de grafo de espera
    - *deadlock* detectado quando grafo é cíclico

Controle de concorrência

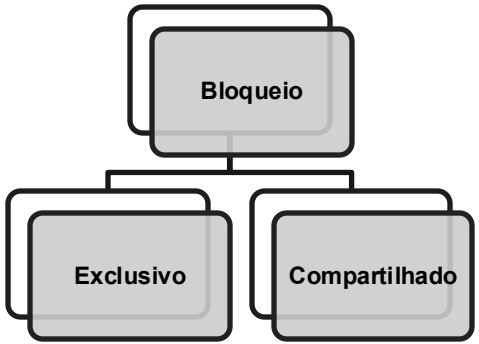
- Desafio para transações que abrangem interações com usuário
- Aplicações implementam esquema de validação de escritas usando versionamento dos registros de dados
  - não necessitam modificar banco de dados,
  - mas configuram nível fraco de serializabilidade

Controle de concorrência

- Outras técnicas especiais podem ser utilizadas para relações de dados especiais, como
  - Árvores B^+ - para permitir maior concorrência
  - Estruturas de dados sem tranca (*latch-free*) - para índices de alto desempenho

Concorrência - Definição

- **Concorrência** é a propriedade de uma transação poder ser executada em **paralelo** com outras transações
- Em sistemas com transações **simultâneas**, é importante garantir a **SERIALIZAÇÃO**
- Um modo de garantir a serialização é a utilização de **BLOQUEIOS** (***LOCK***).



Definição

- Bloqueio **Compartilhado** (**S** – ***S**hared lock*):
  - **T_1** pode ler, mas não pode escrever no item **Q**;
  - obtido sempre que não houver nenhum bloqueio exclusivo

- Bloqueio **Exclusivo** (**X** – *e**X**clusive lock*):
  - **T_1** pode ler e escrever no item **Q**;
  - obtido somente se não houver nenhum outro bloqueio

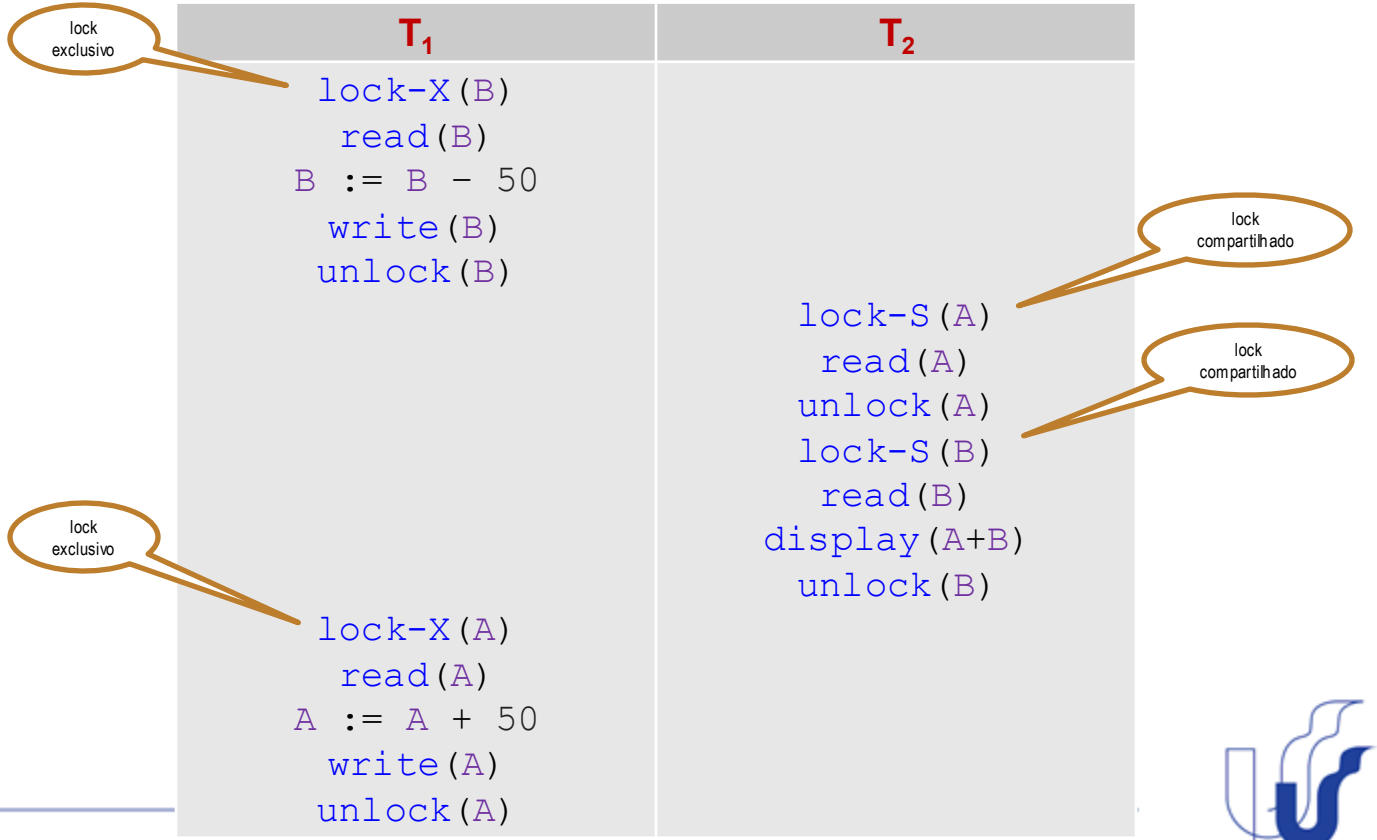
Definição

- Relação de compatibilidade entre os dois modos de Bloqueios:

Bloqueio	lock-S (Compartilhado)	lock-X (Exclusivo)
lock-S	✓ True	× False
lock-X	× False	× False

- O modo **compartilhado** é **compatível** com o modo **compartilhado**, mas **não** com o modo exclusivo
- Vários bloqueios no modo compartilhado podem ser mantidos simultaneamente

Exemplo *Lock*



Exemplo 2 - *Lock*

- *Schedule* anterior com o **desbloqueio** ao final

T_1	T_2
lock-X(B) read(B) B := B - 50 write(B) ... lock-X(A) ...	 lock-S(A) read(A) lock-S(B) ...

T_1 está mantendo um bloqueio exclusivo sobre B e T_2 está solicitando um bloqueio compartilhado, assim T_2 está esperando que T_1 desbloqueie B.

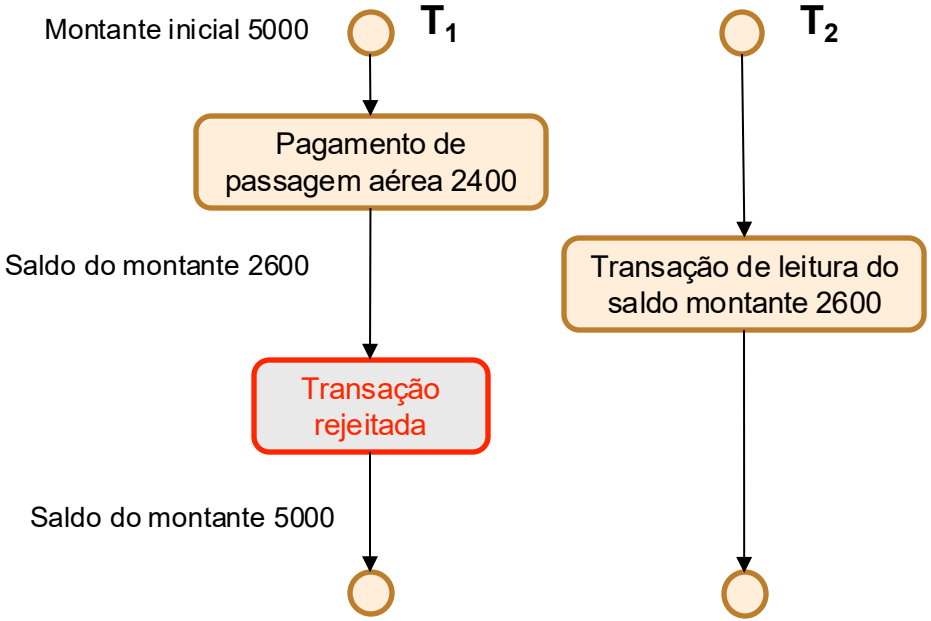
Como T_2 está mantendo um bloqueio compartilhado sobre A e T_1 está solicitando um bloqueio exclusivo sobre A, T_1 está esperando que T_2 desbloqueie A.

Nenhuma das transações pode prosseguir (**IMPASSE** ou **DEADLOCK**)

Os impasses são preferíveis que os estados inconsistentes, pois podem ser tratados pelo **ROLLBACK**.

Problemas (sem *lock*)

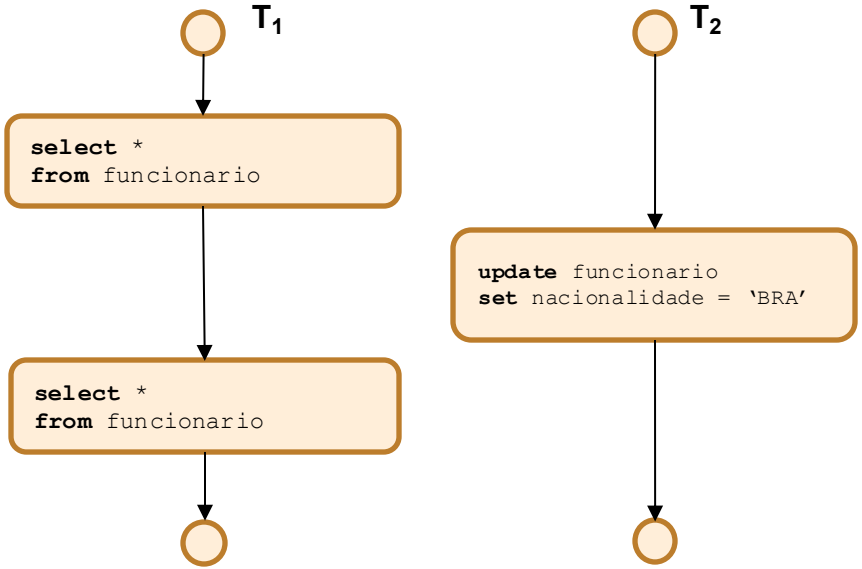
Leitura Suja (*Dirty Read*): uma transação T_2 pode ler a atualização de uma transação T_1 que **ainda não tenha sido confirmada**. Se T_1 falhar e for abortada, então T_2 terá lido um valor que não existe e é incorreto.



Problemas (sem *lock*)

Leitura Não-Repetitiva ou Não-Repetível (*Non-Repeatable Reads*):

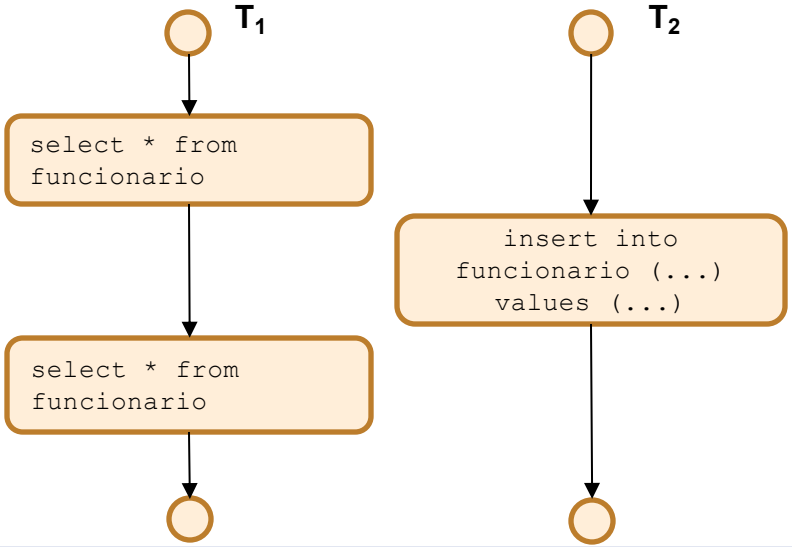
- ocorre quando uma transação lê o valor de um objeto duas vezes obtendo valores diferentes em cada leitura
- uma transação T_1 pode ler um dado valor de uma tabela. Se uma outra transação T_2 posteriormente atualizar esse valor e T_1 ler novamente esse valor, T_1 verá um novo valor.



Problemas (sem *lock*)

Fantasmas (*Phantoms*):

- Tuplas que podem “aparecer” ou “desaparecer” em uma seleção são chamadas **tuplas fantamas**
- Uma transação T_1 pode ler um conjunto de linhas de uma tabela com base em alguma condição especificada na cláusula WHERE da instrução SQL.
- Suponha que uma transação T_2 insira uma nova linha que também satisfaça à condição da cláusula WHERE empregada em T_1 na mesma tabela utilizada por T_1. Se T_1 for repetida verá uma linha fantasma, ou seja, uma linha que antes não existia.



Níveis de Isolamento do Padrão ANSI SQL

- O padrão ANSI SQL define quatro níveis de isolamento:
  1) READ UNCOMMITTED
  2) READ COMMITTED
  3) REPEATABLE READ
  4) SERIALIZABLE

- Cláusula SQL:
  - `SET TRANSACTION ISOLATION LEVEL "nível de isolamento"`

- Além disso, é possível informar se transação será apenas de leitura ou de leitura e escrita
  - `SET TRANSACTION { READ ONLY | READ WRITE }`

Níveis de Isolamento

- **SERIALIZABLE**: transação é **totalmente isolada**. Caso transação tenha comandos DML que tentem atualizar dados não gravados de outra transação, essa transação não será efetuada.

- **REPEATABLE READ**: dados podem ser lidos mais de uma vez, e se outra transação tiver incluído ou atualizado linhas e estas forem gravadas no DB entre uma e outra leitura, dados retornados da última busca serão diferentes de busca anterior. Efeito **leitura fantasma (*phantom*)**.

- **READ COMMITED**: caso transação necessite DML em linhas que outras transações estão utilizando, operação somente será **concluída após a liberação da linha da outra transação**.

- **READ UNCOMMITED**: lidos conteúdos não gravados ainda no DB. Há risco, visto que transação que está bloqueando a informação pode descartá-la (ROLLBACK). Efeito **leitura suja (*dirty*)**.

Níveis de Isolamento e Anomalias

- Níveis de isolamento do padrão ANSI SQL foram definidos à partir das anomalias leitura **suja**, leitura **não repetíve**l e tuplas **fantasmas**.

Nível	Leitura Suja (Dirty Read)	Leitura Não-Repetível (Non-Repeatable Reads)	Tuplas Fantasmas (Phantoms)
READ UNCOMMITTED	Sim	Sim	Sim
READ COMMITTED	Não	Sim	Sim
REPEATABLE READ	Não	Não	Sim
SERIALIZABLE	Não	Não	Não

Observação: Sim = pode ocorrer

Starvation (Inanição)

T_1	T_2	T_3	T_4	T_5
lock-S(Q)				
	lock-X(Q)			
	bloqueada...	lock-S(Q)		
	bloqueada...		lock-S(Q)	
	bloqueada...			lock-S(Q)

T_2 **nunca** recebe o direito de acesso! (pois lock-X é obtido somente se não houver nenhum outro bloqueio);
Inanição ocorre quando uma transação nunca é executada, ou seja, **"morre de fome"**;

• Pode ser **evitado** fazendo com que o **direito** de **acesso compartilhado não seja concedido** se houver uma **transação esperando por bloqueio exclusivo**.

Protocolos de Bloqueio

• Exige-se que cada **transação** do sistema siga um determinado conjunto de regras, chamado de **protocolo de bloqueio**

• Indica quando uma **transação pode ou não bloquear ou desbloquear** cada um dos **itens de dados**

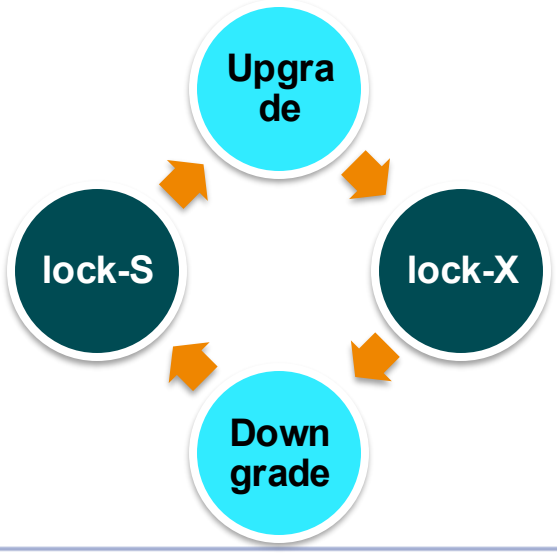
• Os protocolos de bloqueio **restringem** o número de **escalas de execução** possíveis

• O **conjunto** de **todas as escalas** permitidas pelos protocolos de bloqueio **forma** um **subconjunto** de todas as **escalas serializáveis** possíveis

Protocolos de Conversão de Bloqueios

Utilizam instruções de conversão de bloqueios para **alternar** entre modos de bloqueio diferentes

- **Upgrade(Q)**: bloqueio **compartilhado** para **exclusivo**
- **Downgrade(Q):** bloqueio **exclusivo** para **compartilhado**



Protocolo Baseado em Grafo

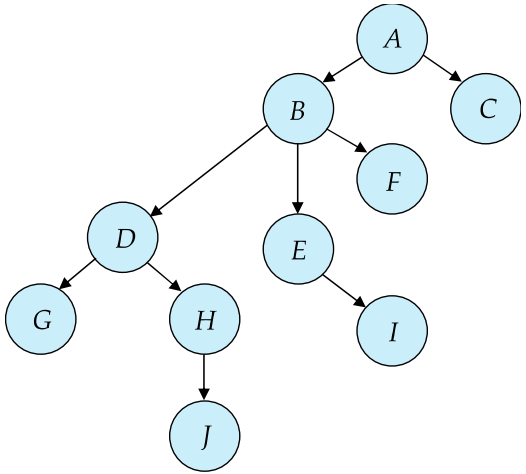
• Determinam uma **ordem parcial** no **acesso aos dados** usando **grafo de precedência**

• Supondo que,
- ✓ exista uma ordem de precedência **A -> B** no grafo,
- ✓ qualquer transação que acesse **A** e **B**,
- ✓ deve acessar **primeiro A** e **depois B**

• A ordem pode ser **física** ou **lógica** ou **aleatória**
• Existem vários tipos de protocolos baseados em grafo, o mais simples é o **protocolo de grafo em árvore**

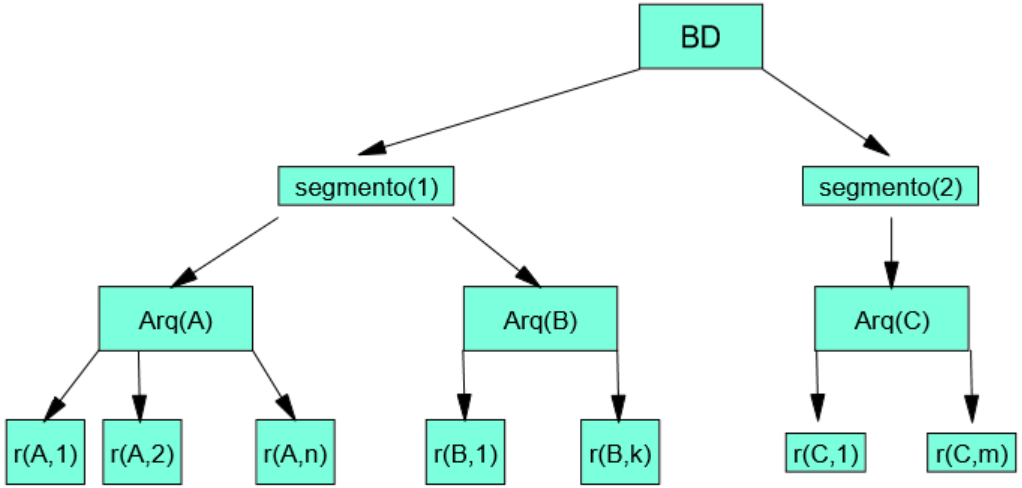
Protocolo de Grafo em Árvore

- **Bloqueios** são todos **exclusivos**;
- **Primeiro bloqueio** poderá ocorrer em **qualquer dado**;
- O **bloqueio seguinte somente ocorrerá caso** o seu **antecessor** esteja **bloqueado**;
- **Dados** podem ser **desbloqueados** a qualquer momento;
- Garante **serialização de conflito** e **evita *deadlock***;
- Pode **bloquear mais dados** do que realmente **precisa**;
- Potencial **redução** de **concorrência**;
- Aumento do **tempo de resposta** e **sobrecarga de bloqueio**.



Protocolo de Granularidade

- Até o momento, a **referência a cada item** sempre foi como uma **unidade** (A, B, C, X) à qual a **sincronização** é **aplicada**;
- O sistema pode implementar **níveis múltiplos de granularidade** conforme a seguir:



- **Quatro níveis**: banco de dados, áreas (blocos que compõem o BD), tabelas e registros.

Protocolo de Granularidade

- Cada **nó da árvore** pode ter **bloqueio individual** (**exclusivo** ou **compartilhado**);
- Quando uma **transação bloqueia** um determinado **nó**, **todos** os **nós descendentes** serão **bloqueados**;
- Se uma **transação bloqueia** de forma **explícita** o arquivo **Arq(A)**, no modo **exclusivo**, então está sendo bloqueado de forma **implícita** (no modo **exclusivo**), **todos** os **registros** daquele **arquivo**;
- **Exemplo:**

Se $\mathbf{T_i}$ bloquear o registro **r(B, 1)** do arquivo **Arq(B)**,
$\mathbf{T_i}$ precisará percorrer a árvore até o registro **r(B, 1)**.
Se **algum nó** deste caminho estiver **bloqueado** de modo incompatível,
então, $\mathbf{T_i}$ precisará **esperar**.

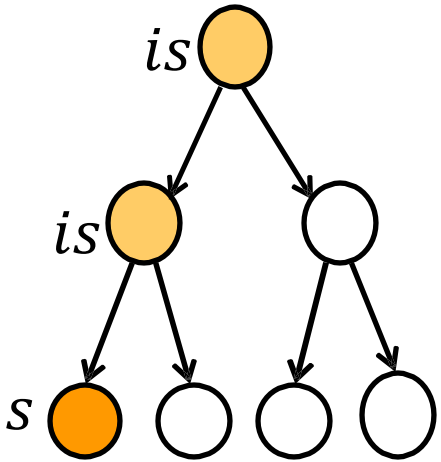
Bloqueios de Intenção

- Um bloqueio de intenção sobre v representa a "**intenção**" de realizar uma ação sobre um dos nós da subárvore enraizada por v
- Usados para navegar na árvore de objetos de múltiplas granularidades:
  - Bloqueios de intenção devem ser **obtidos sobre todos vértices** ancestrais (isto é, o caminho do vértice pai até a raiz da árvore) do vértice sobre o qual pretende-se obter o bloqueio explícito

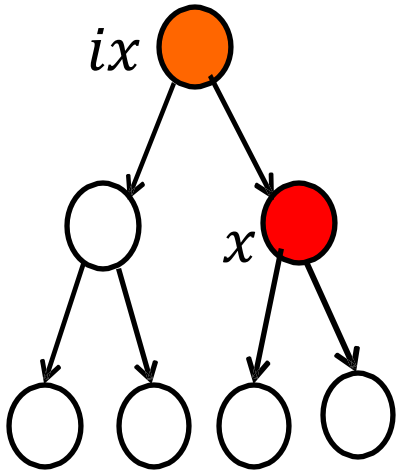
Tipos de Bloqueios de Intenção

- Bloqueio de intenção de leitura (***intention shared lock***):
  - $is(v)$: bloqueio de leitura explícito será realizado sobre algum vértice da subárvore enraizada por v
- Bloqueio de intenção de escrita (***intention exclusive lock***):
  - $ix(v)$: bloqueio de escrita explícito será realizado sobre algum vértice da subárvore enraizada por v
- Bloqueio de leitura e de intenção de escrita (***shared and intention exclusive lock***):
  - $six(v)$: bloqueio de leitura explícito sobre v e um bloqueio de escrita explícito será realizado sobre algum vértice dessa subárvore

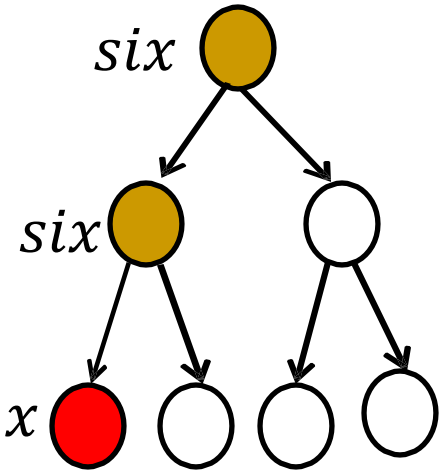
Bloqueios de Intenção: Exemplos



Sequência de bloqueios de **intenção** de **leitura** até a obtenção de um **bloqueio** (explícito) de **leitura**



Sequência de bloqueios de **intenção** de **escrita** até a obtenção de um **bloqueio** (explícito) de **escrita**



Sequência de **bloqueios** de **leitura** e **intenção** de **escrita** até a obtenção de um **bloqueio** (explícito) de **escrita**.

Protocolo de Granularidade – Bloqueio Intencional

- Garante **serialização**;
- Exige que os **bloqueios** sejam **feitos de cima para baixo** e as **liberações de baixo para cima**;
- Aumenta a **concorrência;**
- Reduz o **tempo de resposta**;

Protocolo de Bloqueio em Duas Fases (ou Bi-fásico ou 2PL – 2 Phase Locking)

Composto de duas fases:

1) Primeira fase: Fase de Expansão

- Pode bloquear dados
- Não pode liberar os bloqueios obtidos

2) Segunda fase: Fase de Recolhimento

- Pode liberar os dados bloqueados anteriormente
- Não pode mais bloquear dados

- Garante escala de execução **serializável em conflito**
- **Precedência** é determinada em função do instante de obtenção do **último bloqueio**
- Não evita ***rollback*** em cascata
- Variantes (**severo** e **rigoroso**) evitam o *rollback* em cascata
- Utilizado pelos SGBDs

Protocolo de Bloqueio em Duas Fases (ou Bi-fásico ou 2PL – 2 Phase Locking)

Possibilidades:

Primeira fase

- Pode obter um `lock-S` sobre o item
- Pode obter um `lock-X` sobre o item
- Pode converter `lock-S` para `lock-X` (***upgrade***)

Segunda fase

- Pode liberar um `lock-S`
- Pode liberar um `lock-X`
- Pode converter `lock-X` para `lock-S` (***downgrade***)

2PL – Exemplo 1

T1	T2	Gerenciador de Controle de Concorrência
lock-X(A)		grant lock-X(A), T1
read(A)		
write(A)		
	lock-X(A)	blocked (wait), T2
	read(A)	
	write(A)	
lock-X(B)		grant lock-X(B), T1
read(B)		
write(B)		
unlock(A,B)		revoke lock-X(A), T1 revoke lock-X(B), T1
	lock-X(B)	grant lock-X(A), T2 grant lock-X(B), T2
	read(B)	
	write(B)	
	unlock(A,B)	revoke lock-X(A), T2 revoke lock-X(B), T2

UNISINOS
DESAFIE O AMANHÃ.

2PL – Exemplo 2

T_1	T_2
lock-X(A) read(A) A := A - 50 write(A) lock-X(C) **unlock(A)** read(C)	
	lock-S(C) **Bloqueada**
C := C + 50 write(C) unlock(C)	
	read (C) lock-S (A) **unlock(C)** read(A) print(C + A) unlock(A)

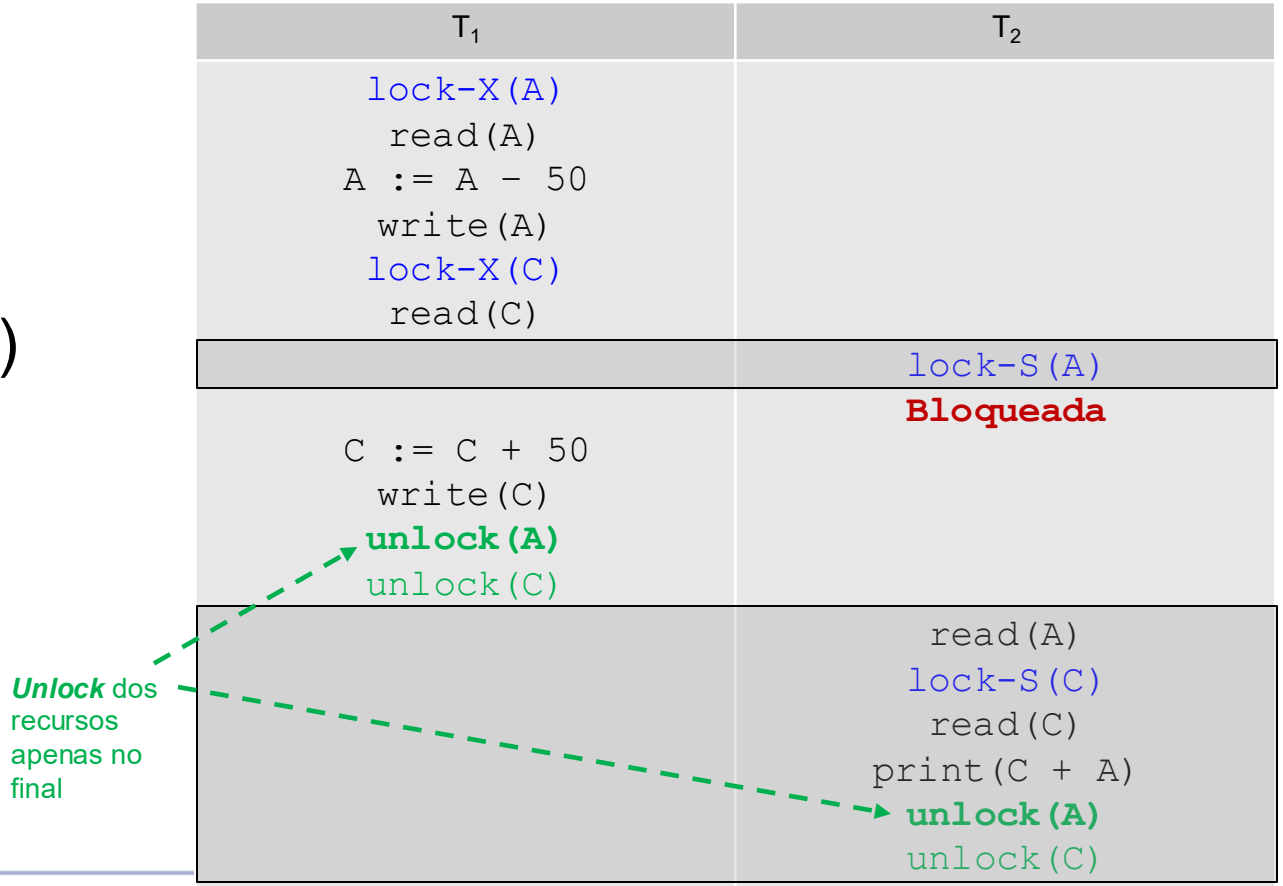
2PL – Exemplo Deadlock

T_1	T_2
lock-X(A) read(A) A := A – 50	
	lock-S(C) read (C) lock-S (A)
write(A) lock-X(C) **Bloqueada**	**Bloqueada** . . .
Não executa: unlock(A) read(C) C := C + 50 write(C) unlock(C)	**Não executa:** unlock (C) read (A) print (C + A) unlock (A)

2PL – Severo (utilizado por SGBDs)

T_1	T_2
lock-X(A) read(A) A := A - 50 write(A) lock-X(C) read(C)	
	lock-S(A)
	Bloqueada
C := C + 50 write(C) **unlock(A)** unlock(C)	
	read(A) lock-S(C) **unlock(A)** read(C) print(C + A) unlock(C)

2PL – Rigoroso (utilizado por SGBD)



UNISINOS
DESAFIE O AMANHÃ.

Protocolo com Base em *Timestamps*

Garante **serializabilidade**: envolve o uso de ***timestamp*** para **ordenar a execução** das transações de forma que o **escalonamento** formado seja **equivalente ao escalonamento serial**.

- Evita ***deadlocks*** e garante a **serialização de conflito** na ordem dos *timestamps*.
- Não impede o ***rollback*** em cascata.

Timestamp: **identificador único** criado pelo SGBD para **identificar uma transação**. São **associados** na **ordem** em que as transações são **submetidas** ao sistema.

TS(T): *Timestamp* da transação **T**.

No escalonamento as transações tomam a ordem de seus ***timestamps***

• Se $TS(T_i) < TS(T_j)$, o sistema garante que o **escalonamento** produzido é **equivalente** ao **escalonamento serial** <Ti,Tj>.

Protocolo Multiversão

- **Mantém** versões **antigas** de dados para aumentar **concorrência**:
  - Aloca a **versão correta** para uma **operação de leitura** de uma transação;
  - Operações de **leitura não** são **rejeitadas**;
  - Cada **atualização** efetuada **com sucesso** cria uma **nova versão** do dado.

Efeito colateral:

- Necessidade de mais **espaço** de **armazenamento** (RAM e disco);
- Mecanismos de "**coleta de lixo**" (versões não usadas).

Podem ser baseados em:

- Ordenação por ***timestamp***;
- ***2PL – 2 Phase Locking***.

Controle de *Deadlock*

Ocorre quando temos um **conjunto de transações** em que **todas** estão em **espera**, **uma aguardando** o **término da outra** para prosseguir.

Técnicas para **Controle de *Deadlock***:

1. **Prevenção** de *deadlock*
    - Evita os *deadlocks* antes que estes ocorram
    - Preferível se a probabilidade de ocorrerem *deadlocks* for muito alta
2. **Detecção** e **recuperação** de *deadlock*
    - Não evita os *deadlocks*, mas os detecta e impede o bloqueio indefinido das transações envolvidas
    - Mais eficiente se ocorrerem poucos *deadlocks*

Rollback pode ser necessário independentemente da técnica utilizada

Controle de *Deadlock* - **Prevenção**

Estratégias de prevenção usadas por SGBDs

1. **Esperar-morrer:**

   Se $TS(T_i) < TS(T_j)$,

   T_i só espera um dado mantido por T_j se esta for **mais nova**;
   caso contrário T_i é **abortada** (**morre**)

   Exemplo:

   Se **T1**, **T2**, **T3** tiverem *timestamps* 5, 10, 15 respectivamente
   e
   T1 solicita um item **mantido** por **T2**,
   então **T1** **esperará**.
   Se **T3** solicitar o item **mantido** por **T2**,
   T3 será **revertido**, morrerá.

Controle de *Deadlock* – **Prevenção** (cont.)

2. **Ferir-esperar**:

   Se $TS(T_i) > TS(T_j)$,

   T_i somente espera um dado mantido por T_j se esta for mais antiga;
   caso contrário ela obriga T_j a abortar e liberar o dado (T_i **fere** T_j)

   Exemplo:

   Se **T1**, **T2**, **T3** tiverem *timestamps* 5, 10, 15 respectivamente
   e
   T1 solicitar um item de dados **mantido** por **T2**, então
   o item de dados será apropriado de **T1** e
   T2 será revertido.
   Se **T3** solicitar um item de dados mantido por **T2**,
   então **T3 esperará**.

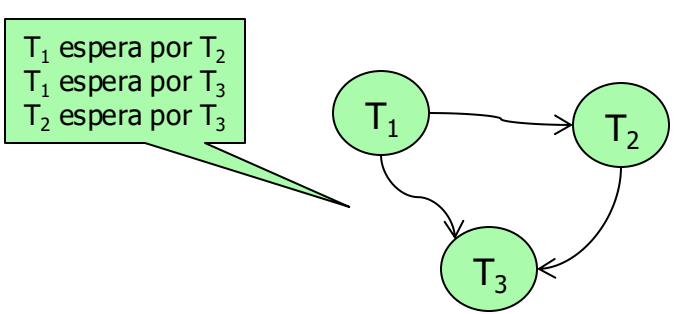
3. ***Timeout:***

   pedido de **bloqueio** possui um **tempo máximo de espera**;
   se o **dado não** for **liberado** neste tempo, a **transação** é **abortada** e **reiniciada**.

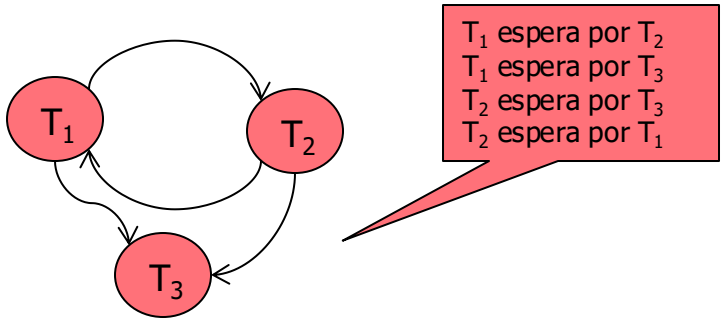
Controle de *Deadlock* - **Detecção**

Para determinar se as transações estão em ***deadlock***:

- **Manter informações** sobre a **alocação** dos dados e as solicitações pendentes
- Usando **grafos de espera**, nos quais um **ciclo indica** um ***deadlock***



Grafo sem ciclo



Grafo com ciclo

Controle de *Deadlock* - **Recuperação**

Reverter uma ou **mais transações** para **quebrar** o ***deadlock***.

Devem ser tomadas **3 (três)** ações:

1) **Selecionar uma vítima (ou vítimas) de acordo como mínimo custo:**
   - A quanto tempo a transação está em processamento e quanto tempo será ainda necessário para que a tarefa seja completada;
   - Quantos itens de dados a transação usou;
   - Quantos itens ainda a transação usará até que se complete;
   - Quantas transações serão envolvidas no *rollback*.

2) **Rollback**: Determinar até que ponto a transação deve ser revertida:
   - Reverter totalmente;
   - O suficiente para quebrar o ***deadlock*** (exige informações adicionais).

3) **Inanição**: Deve-se garantir que uma transação seja escolhida vítima somente um número finito e pequeno de vezes.

Recuperação de falhas

Agenda

- Visão geral sobre recuperação de falhas
- Sistema de recuperação de falhas
- Esquemas baseados em *log*
- Paginação de sombra (*shadow page*)
- Esquema de modificações adiadas
- Algoritmos de recuperação modernos
- Sistema de *backup* remoto
- Técnicas de recuperação modernas
- Dump
- Esquema de recuperação ARIES

Visão geral sobre recuperação de falhas

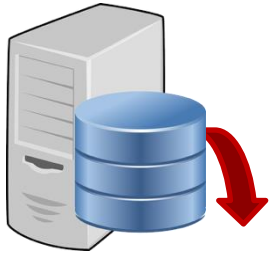
- Computador e dispositivos mecânicos ou elétricos sujeitos a falhas
- Dados podem ser perdidos
- Incluindo:
    - falha de disco
    - falta de energia
    - erros de *software*

Visão geral sobre recuperação de falhas

- Transações também podem falhar, por motivos como:
  - violação de restrições de integridade ou
  - impasses (*deadlocks*).

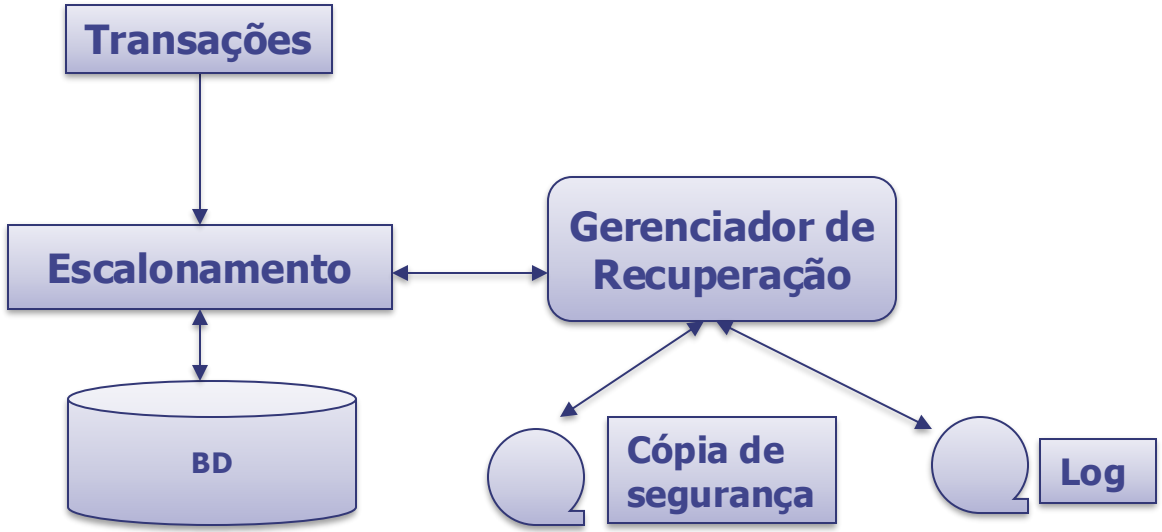
Introdução

- Um sistema de computador está sujeito a **falhas** e informações podem ser **perdidas**
- Em resumo, existe uma grande variedade de falhas:
  - Quebra de disco
  - Falha de energia
  - Erro de software
  - Fogo
  - Sabotagem



- O SGBD deve **precaver-se** para garantir que as propriedades de **atomicidade** e **durabilidade** das transações sejam preservadas quando ocorrem falhas
- A parte integrante do SGBD, responsável pelas ações de precaução, é o **ESQUEMA DE RECUPERAÇÃO**

Arquitetura do Gerenciador de Recuperação



Recuperação de **curta** duração
Exemplo: Falha de transação (uso de Log)
Recuperação de **média** e **longa** duração
Exemplo: Falha de sistema (BD + Log)
Exemplo: Falha de disco (Cópia de segurança + Log)

Classificação de Falha

- O **tipo de falha** mais simples de tratar é aquele que **não resulta na perda** de informação;
- Contudo, as falhas mais difíceis de tratar são aquelas que resultam em **perda de informação**. Iremos considerar os seguintes tipos de falha:
  - **Falha de Transação**
    - **Erro lógico**: a transação não pode mais continuar com sua execução normal devido a alguma condição interna, como uma entrada inadequada, um dado não encontrado, *overflow* ou limite de recurso excedido;
    - **Erro de sistema**: o sistema entrou em um estado inadequado (*deadlock*). A transação, entretanto, pode ser reexecutada posteriormente.
  - **Queda do sistema**
    - Há um **mal funcionamento** de **hardware** ou um **bug** no **software** de banco de dados ou no sistema operacional que causou a perda de conteúdo no armazenamento volátil. O conteúdo de armazenamento não-volátil permanece intacto e não é corrompido. Condição: **falhar-parar**.

Classificação de Falha

- **Falha de Disco**
  - Um **bloco de disco perde** seu **conteúdo** em função da **quebra** do **cabeçote** ou da **falha** durante uma **operação** de **transferência** de dados
- **Desastres**
  - Como **incêndios** ou **enchentes** (não está no escopo, pois estamos preocupados com a gerência de recuperação)

- Há necessidade de identificar os modos de falhas possíveis dos equipamentos usados para armazenar dados. Assim, aplica-se o algoritmo certo.

Falha Durante a Transferência de Dados

- A transferência de blocos entre a memória e o armazenamento de disco pode resultar em:
  1. **Conclusão bem-sucedida**: a informação transferida chegou de forma segura a seu destino;
  2. **Falha parcial**: uma falha ocorreu no meio da transferência e o bloco de destino contém informação incorreta;
  3. **Falha total**: a falha ocorreu cedo o suficiente, de modo que o bloco de destino permanece intacto.

Sistema de recuperação de falhas

- Em caso de falha, banco de dados pode:
  - não estar mais consistente
  - não refletir mais estado real
- Esquema de recuperação deve garantir propriedades ACID

Recuperação e Atomicidade

- Vamos **considerar** um **transação** T_i em que ocorre uma **transferência** de **R$ 50** de uma conta **A (R$ 1000)** para **B (R$ 2000);**
- Suponha que uma **queda de sistema** tenha ocorrido **durante a execução** de T_i, **após** o `output(`B_a`)`, mas **antes** do `output(`B_b`)`, em que B_a e B_b denotam os blocos de *buffer* em que A e B residem.
- Como os **conteúdos de memória foram perdidos**, não sabemos o destino da transação; então, poderíamos chamar um dos dois possíveis procedimentos de recuperação:
  1. **Reexecutar** T_i: este faz com que o valor de A torne-se R$ 900, em vez de R$ 950. Então, o sistema entra em um estado **inconsistente**;
  2. **Não reexecutar** T_i: no estado corrente do sistema, os valores de A e B são de R$ 950 e R$ 2000, respectivamente. Então, o sistema entra em um estado **inconsistente**.
- Em ambos os casos, o banco de dados é deixado em estado **inconsistente**.
- Logo, esse esquema **SIMPLES** de recuperação **NÃO FUNCIONA**.

Esquemas baseados em log

- Atualizações registradas em *log* → em armazenamento estável
- Transação confirmada → último *log* (*commit*) enviado para disco estável
- *Log* contêm valores antigos e novos para todos os registros atualizados
  - Valores novos - caso atualizações necessitem ser **refeitas** após uma falha
  - Valores antigos - para **reverter** atualizações se transação abortar a qualquer momento

Técnicas de Recuperação (*Recovery*)

1. **Baseadas em *Log***
   a. Modificação **Imediata** do BD
      - Técnica **UNDO/REDO**
      - Técnica **UNDO/NO-REDO**
   b. Modificação **Postergada (Adiada)** do BD
      - Técnica **NO-UNDO/REDO**
2. **Baseadas em *Shadow Pages***
   - Técnica de **NO-UNDO/NO-REDO**
3. **Baseadas em *Archive/Dump***

Tipos de *Log*

- *Log* de **UNDO**:
  mantém apenas o valor **antigo** do dado (*before image*)
- *Log* de **REDO**:
  mantém apenas o valor **atualizado** do dado (*after image*)
- *Log* de **UNDO/REDO** (mais comum):
  mantém os valores **antigo** e **atualizado** do dado

Recuperação Baseada em *Log*

- É uma estrutura de dados que possui uma **sequência de registros de log** que mantém as **atividades** no banco
- Descreve uma **única escrita no banco** e possui a seguinte estrutura:
  a. **Identificador de transação**: identificador único da transação que realiza operação de escrita;
  b. **Identificador de item de dado**: identificador único do item de dado escrito (normalmente, é a localização no disco ou o tipo de instrução);
  c. **Valor antigo**: valor do dado anterior à escrita;
  d. **Valor novo**: valor que o dado terá após a escrita.

Recuperação Baseada em *Log*

- Estrutura da transação no arquivo de Log

<T_i start> -- Transação T_i **começou**

<T_i, x_j, v_1, v_2> -- Transação T_i realizou uma **escrita** no dado x_j
x_j tinha valor v_1 antes da alteração
x_j terá valor v_2 após a escrita

<T_i commit> -- Transação T_i foi **efetivada**

<T_i abort> -- Transação T_i foi **abortada**

Recuperação Baseada em *Log*

- Seja o ***schedule* S** abaixo:

T_1	T_2
read(A) A := A – 50 write(A) read(B) B := B + 50 write(B)	read(C) C := C – 100 write(C)

Log

<T_1 **start**>
<T_1, A, 1000, 950>
<T_1, B, 2000, 2050>
<T_1 **commit**>

<T_2 **start**>
<T_2, C, 700, 600>
<T_2 **commit**>

BD

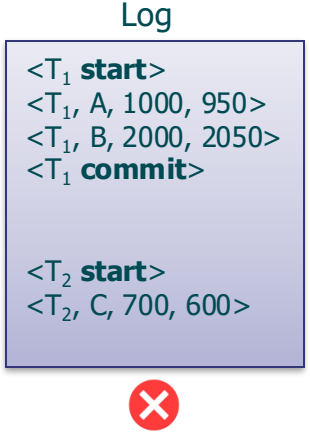
A = 950
B = 2050

C = 600

- Usando o **log**, o sistema pode **lidar com qualquer falha** que resulte em perda de informação no armazenamento volátil;
- A operação `redo(`T_i`)` deve ser **idempotente**, isto é, **executá-la várias vezes deve ser equivalente a executá-la uma vez só**.

Recuperação Baseada em *Log*

- Digamos que ocorreu uma **falha** logo após o `write(c)`. Neste caso, o log ficará conforme apresentado abaixo.



- Quando o SGBD retorna, a operação `redo(T1)` é realizada, pois o registro `<T1 commit>` aparece no log em disco e o `<T2 commit>` não encontra-se no log.
- Ao ocorrer uma **queda** novamente, o sistema de recuperação irá processar novamente. Mantendo a **consistência dos dados**.

Recuperação Baseada em *Log*

- **Dois** mecanismos podem ser pensados para permitir a recuperação:

1. **Modificação ADIADA**: apenas ao **final** a modificação é de fato confirmada no banco de dados;
2. **Modificação IMEDIATA**: **assim que** a **operação ocorre**, ela é gravada no banco de dados.

Esquema de modificações adiadas

- Operações *write* são adiadas até que transação seja confirmada
    - Sistema usa *log* da transação para executar escritas adiadas

- Com modificação adiada, *log* não precisa de valores antigos

- Sistema pode usar pontos de verificação (*checkpoints*) para reduzir *overhead* da pesquisa do *log* para refazer transações

Recuperação Baseada em *Log* - ADIADA

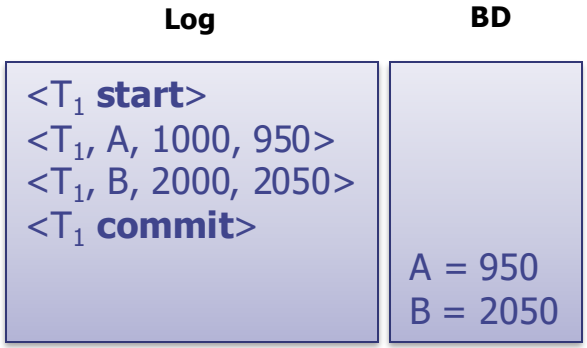
- A técnica de adiar a modificação do banco de dados assegura a **atomicidade** da transação gravando todas as modificações do banco de dados no log, **adiando a execução de todas as operações *write*** de uma transação **até que a transação seja parcialmente executada**;

- Depois de uma **falha** ter ocorrida, o subsistema de recuperação **consulta** o **log** para determinar que transações precisam ser refeitas.
- A transação T_i precisa ser refeita se, e somente se, o log contiver o registro **<T_i start>** e o registro **<T_i commit>;**

- Perceba que **não é preciso desfazer uma transação**, uma vez que os dados são de fato **efetivados no banco apenas ao final da transação**.

Recuperação Baseada em *Log* - ADIADA

- Ex.: considere que **A** e **B** tenham **R$ 1000** e **R$ 2000**, respectivamente, antes das transações serem executadas, conforme o ***schedule* S**.
- A seguir são mostrados estados do **log** e do banco de dados correspondentes a **T_1**.



Recuperação baseada em *Log* - IMEDIATA

- A técnica de atualização imediata permite modificações do banco de dados **enquanto a transação está ainda no estado "ativo"**;

- Estas modificações denominam-se **modificações não confirmadas**. Em caso de falha, o valor antigo dos registros **log** pode ser usado para restaurar os itens de dados modificados aos valores que tinham **antes do início da transação**;

- **Undo(T_i)**, **restaura** o valor de todos os itens de dados atualizados pela transação T_i aos **valores antigos**;

- **Redo(T_i)**, **ajusta** o valor de todos os itens de dados atualizados pela transação T_i aos **novos valores**.

Recuperação Baseada em *Log* - IMEDIATA

- Ex.: considere que **A** e **B** tenham **R$ 1000** e **R$ 2000**, respectivamente, antes das transações serem executadas, conforme o ***schedule* S**.
- A seguir são mostrados os estados do **log** e do banco de dados correspondentes a T_1.

Log	BD
<T_1 **start**> <T_1, A, 1000, 950> <T_1, B, 2000, 2050> <T_1 **commit**>	A = 950 B = 2050

Recuperação Baseada em *Log* - IMEDIATA

- Depois da ocorrência de uma **falha**, o esquema de recuperação consulta o **log** para **saber quais transações precisam ser refeitas** e quais **precisam ser desfeitas**.

- Esta classificação de transação é feita assim:
- A transação T_i precisa ser desfeita (**UNDO**) se o log contiver o registro $<T_i$ start$>$ e não contiver o registro $<T_i$ commit$>$
- A transação T_i precisa ser refeita (**REDO**) se o log contiver os registros $<T_i$ start$>$ e $<T_i$ commit$>$

Paginação de sombra (*shadow page*)

- Alternativa para recuperação por *log*
- Útil para transações executadas em série
- Mantém duas tabelas durante transação (inicialmente iguais)
    - Tabela de página atual
        - acessada e manipulada durante processamento normal
    - Tabela de página sombra
        - não modificada
        - em armazenamento não volátil → para recuperação caso necessário

Algoritmos de recuperação modernos

- Baseados no conceito de repetição de história
  - ações durante operação normal (desde último *checkpoint*) são reproduzidas durante a passada de ***redo*** da recuperação
  - restaura para estado do momento do último *log* do armazenamento estável antes da falha
  - ***undo*** é realizado a partir desse estado, processando *logs* incompletos na ordem reversa

Sistema de *backup* remoto

- Objetivo → alta disponibilidade
- Processamento de transação continua mesmo com site primário sinistrado (incêndio, inundação, etc).
- Dados e *logs* de site primário continuamente copiados para site *backup* remoto
- Se site primário falhar, site de *backup* remoto assume processamento

Técnicas de recuperação modernas

- Suporte à alta concorrência, como por árvores B^+
- Permitem liberação antecipada de bloqueios de nível inferior
    - por exemplo: em inserções ou exclusões
- Transações retêm bloqueios de nível mais alto
    - garantia de transações concorrentes não impedir desfazer operações

Dump

- Recuperação de falhas por perda de armazenamento não volátil
- Utilização de *dump* mais recente para restaurar dados
- Utilização de *log* para retornar ao estado consistente mais recente

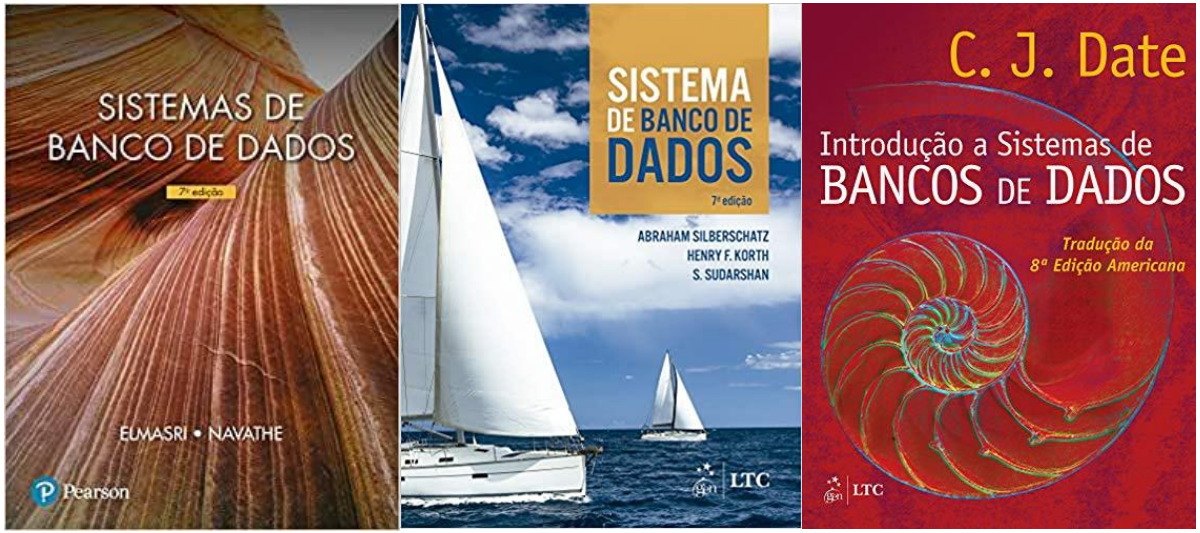
Esquema de recuperação ARIES

- *Algorithm for Recovery and Isolation Exploiting Semantics*
- Série de recursos para oferecer maior concorrência
- Reduz *overhead* de *logging*
- Minimiza tempo de recuperação
- Baseado na repetição de histórico
- Permite operações de ***undo*** lógico
- Esvazia páginas continuamente
- Não precisa esvaziar todas as páginas ao passar por *checkpoint*
- Usa números de sequência lógicos (LSNs) para otimizar recuperação

Referências

- MULLER, Gilberto I.; **Apresentação e Notas de Aula**. Unisinos, 2021.
- CONNOLLY, Thomas M.; BEGG, Carolyn E. (Adapt.). **Database systems: a practical approach to desing, implementation, and management**. 2. ed. England: Addison-Wesley, 1999. 1094 p.
- HEUSER, Carlos Alberto. **Projeto de banco de dados**. 6. ed. Porto Alegre: Bookman, 2008. 282 p.
- OZSU, M. Tamer; VALDURIEZ, Patrick. **Principles of distributed database systems**. 2. ed. New Jersey: Prentice-Hall, 1999. 665 p.
- CONNOLLY, Thomas M.; BEGG, Carolyn E. **Database systems: a practical approach to design, implementation, and management**. 5th. ed. Harlon: Addison-Wesley, 2010. xlviii, 1242, [97] p
- DATE, C. J. **Introdução a sistemas de bancos de dados**. 4. ed. Rio de Janeiro: Campus, 2000.
- ELMASRI, Ramez; NAVATHE, Sham. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson, 2011. xviii, 788 p.
- SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados**. 7. ed. Rio de Janeiro: LTC, 2020. 728 p.

Referências Bibliográficas



Sistemas de Gerência de Banco de Dados