Capítulo 15: Transacções

- Conceito de Transacção
- Estado de uma Transacção
- Execução Concorrente
- Seriabilidade
- Teste de Seriabilidade
- Recuperabilidade
- Controle de Concorrência vs. Testes de Seriabilidade
- Níveis Fracos de Consistência

Conceito de Transacção

- Uma transacção é uma unidade de execução de um programa que consulta e altera os dados.
- Uma transacção tem que encontrar a base de dados num estado consistente.
- Durante a execução de uma transacção, a base de dados pode estar temporariamente inconsistente.
- Quando uma transacção termina com sucesso (confirmada/ committed), a base de dados tem que ficar num estado consistente.
- Se uma transacção é abortada, a base de dados tem que ser restaurada para o seu estado anterior ao início da transacção.
- Depois da confirmação (commit) de uma transacção, as alterações feitas à base de dados devem perdurar, mesmo se houver falhas no sistema.
- Várias transações podem executar em paralelo.
- Dois problemas fundamentais:
  - Falhas de vários tipos, e.g. falhas de hardware e *crashes* do sistema
  - Execução concorrente de várias transações

Propriedades ACID

- Transacção que transfere €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 transacção falha (hardware ou software) após o 3ª passo e antes do 6ª passo, o sistema deve garantir que as modificações não se reflectem na base de dados, caso contrário resultaria numa inconsistência i.e. perdia-se dinheiro.
  - ✷ Tudo ou Nada no que respeita à execução da transacção.

Propriedades ACID

- Transacção que transfere €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 Consistência**: a soma de A e B permanece inalterada pela execução da transacção. Em geral, os requisitos de consistência incluem:
  - Restrições de integridade explicitas (e.g. chaves primárias e externas)
  - Restrições de integridade implícitas (e.g. Soma A+B deve permanecer inalterada)
  - A transacção deve ver uma base de dados consistente e deixar uma base de dados consistente
  - Durante a execução de uma transacção, a base de dados pode estar temporariamente inconsistente.
    - Restrições só são verificadas no fim da transacção.

Propriedades ACID

- Transacção que transfere €50 da conta A para a conta B:
  1. read(A)
  2. A := A – 50
  3. write(A)

     read(A), read(B), print(A+B)
  4. read(B)
  5. B := B + 50
  6. write(B)
- **Requisito de Isolamento**: se entre o 3ª e o 6ª passos, outra transacção pudesse aceder à base de dados parcialmente actualizada, veria uma base de dados inconsistente (a soma A+B seria menor do que deveria ser).
  - O isolamento pode ser garantido de forma trivial se apenas permitirmos a execução em série das transações i.e. Uma após a outra.
  - No entanto, executar várias transações de forma concorrente tem vantagens e.g. na performance do sistema.

Propriedades ACID

- Transacção que transfere €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 Durabilidade**: quando o utilizador for notificado que a transacção terminou (i.e. que a transferência dos €50 ocorreu), as actualizações à base de dados devem persistir apesar de falhas do sistema.

Propriedades ACID

Uma transacção é uma unidade de execução de um programa que consulta e altera os dados. Para preservar a integridade dos dados, o sistema de gestão da base de dados dever garantir:

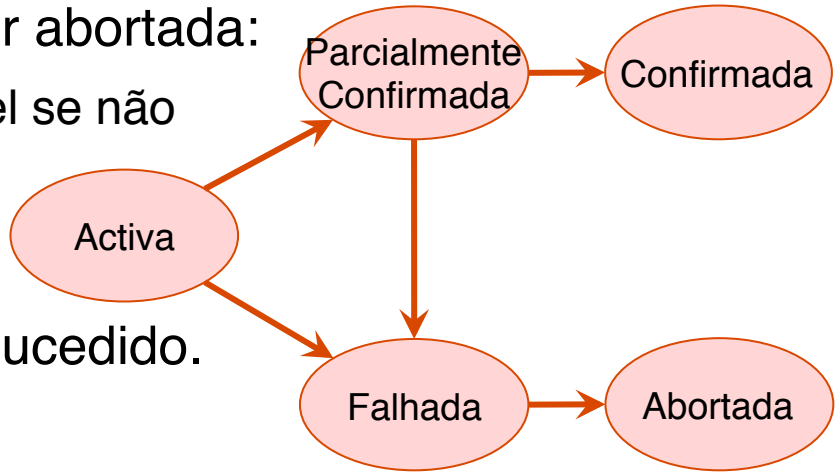
- **Atomicidade**: ou todas as operações da transacção são reflectidas na base de dados, ou nenhuma é reflectida.
- **Consistência**: a execução de uma transacção de forma isolada preserva a consistência da base de dados.
- **Isolamento**: Apesar de várias transações poderem ser executadas de forma concorrente, cada transacção não deve notar a execução concorrente das restantes. Resultados intermédios das transações devem ser escondidos das restantes transacções executadas em concorrência.
  - i.e. para cada par de transacções T_i e T_j, a T_i parece que T_j terminou a sua execução antes de T_i começar, ou T_j iniciou a sua execução após T_i terminar.
- **Durabilidade**: Depois de uma transacção ser confirmada (committed), as modificações que fez na base de dados persistem, mesmo na presença de falhas do sistema.

Transações Não-ACID

- Há domínios de aplicação onde as propriedades ACID não são necessariamente desejadas e, por vezes, impossíveis.
- É o caso das denominadas **transações de longa duração**.
  - No caso de uma duração que demora muito tempo, é pouco provável que o isolamento possa/deva ser garantido.
    - Exemplo de uma transacção de reserva de um hotel ou voo.
- Sem isolamento, a atomicidade pode ser comprometida.
- No entanto, a consistência e durabilidade devem ser preservadas.
- Uma solução habitual consiste na definição de **acções de compensação** – o que fazer se a transacção falha.
- Em bases de dados (centralizadas), transações de longa duração não são habitualmente consideradas.
- São particularmente úteis no contexto da Web.

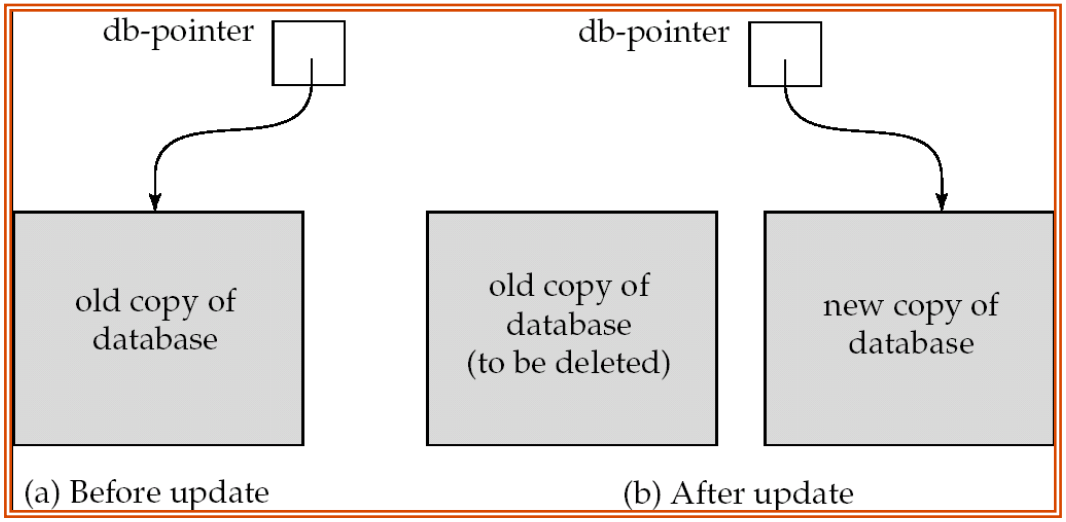
Estado de uma Transacção

- **Activa**: o estado inicial; a transacção permanece neste estado enquanto está em execução.
- **Parcialmente confirmada**: depois da última instrução ter sido executada.
- **Falhada**: depois da descoberta que a execução normal não pode continuar.
- **Abortada**: depois da transacção ter sido desfeita (rolled back) e a base de dados restaurada ao seu estado anterior ao início da transacção. Duas opções após ser abortada:
  - Recomeçar a transacção: possível se não houve um erro lógico interno.
  - Eliminar a transacção
- **Confirmada**: depois do fim bem sucedido.



Implementação de Atomicidade e Durabilidade

- O gestor de recuperações do sistema de gestão de bases de dados implementa o suporte para atomicidade e durabilidade.
- Por exemplo, o esquema da base de dados sombra (*shadow-database*):
  - Todas as actualizações são feitas numa cópia sombra da base de dados
  - O db_pointer passa a apontar para a cópia sombra depois de:
    - A transacção ter atingido o estado de parcialmente confirmada e
    - Todas as páginas actualizadas terem sido reflectidas no disco



Implementação de Atomicidade e Durabilidade

- O db_pointer aponta sempre para a versão corrente, consistente, da base de dados.
  - Se uma transacção falha, a cópia antiga, consistente, apontada pelo db_pointer pode ser usada, e a cópia sombra eliminada.
- O esquema da base de dados sombra:
  - Assume que apenas uma transacção está activa em cada momento.
  - Assume que os discos não falham
  - É útil para editores de texto mas extremamente ineficiente para bases de dados grandes.
    - Variante denominada paginação sombra reduz a cópia de dados, mas continua a não ser prática para bases de dados grandes.
  - Não lida com transacções concorrentes.
- Outras implementações de atomicidade e durabilidade são possíveis, e.g. através da utilização de logs.

Execuções Concorrentes

- Várias transações podem ser executadas de forma concorrente.
- As vantagens são:
  - Aumento do uso do processador e do disco, levando ao aumento do rendimento das transações: uma transacção pode estar a usar o CPU enquanto outra está a ler o disco.
  - Redução do tempo de resposta médio para as transações: transações curtas não têm que esperar muito atrás das longas.
- Esquemas de controlo de concorrência: mecanismos para atingir isolamento
  - i.e. para controlar a interacção entre transações concorrentes para prevenir que destruam a consistência da base de dados:
    - Protocolos baseados em locks
    - Protocolos baseados em timestamps
    - Protocolos multi-versão
  - Estudados em Sistemas Distribuídos e Sistemas de Operação

Escalonamentos

- **Escalonamento**: sequência de instruções que especifica a ordem cronológica segundo a qual as instruções de transações concorrentes são executadas
  - Um escalonamento para um conjunto de transações tem que incluir todas as instruções de todas as transacções.
  - Tem que preservar a ordem segundo a qual as instruções instruções aparecem em cada transacção individual.
- Uma transacção que completa de forma bem sucedida terá um *commit* como última instrução.
  - Por omissão, assume-se que todas as instruções apresentadas nos slides têm um *commit* como última instrução.
- Uma transacção que falha a sua execução tem um *abort* (rollback) como última instrução.
- O objectivo é encontrar escalonamentos que preservem a consistência.

Escalonamento 1

- Seja
  - T_1 a transferência de €50 de A para B, e
  - T_2 a transferência de 10% do saldo de A para B.
- Escalonamento serializado onde T_1 é sucedida por T_2.

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 2

- Seja
  - T_1 a transferência de €50 de A para B, e
  - T_2 a transferência de 10% do saldo de A para B.
- Escalonamento serializado onde T_2 é sucedida por T_1.

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

Escalonamento 3

- Seja
  - T_1 a transferência de €50 de A para B, e
  - T_2 a transferência de 10% do saldo de A para B.
- Escalonamento não serializado, mas equivalente ao Escalonamento 1.
- Nos escalonamentos 1, 2 e 3, a soma de A+B é preservada

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 2

- Seja
  - T_1 a transferência de €50 de A para B, e
  - T_2 a transferência de 10% do saldo de A para B.
- O seguinte escalonamento não preserva o valor de A+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)

Seriabilidade

- **Assunção básica**: Cada transacção preserva a consistência da base de dados.
  - Logo, a execução serializada (sequencial) de um conjunto de transacções preserva a consistência.
- Um escalonamento (possivelmente concorrente) é serializável se é equivalente a um escalonamento serializado. Noções diferentes de equivalência de escalonamentos dão origem às noções de:
  - seriabilidade de conflictos
  - seriabilidade de vistas
- Vista simplificada de transacções:
  - Ignoraremos operações distintas das instruções **read** e **write**.
  - Assumiremos que as transacções podem efectuar computações arbitrárias sobre os dados, em buffers locais, entre leituras e escritas.
  - Os escalonamentos simplificados consistem apenas de instruções **read** e **write**.

Instruções em Conflito

- Duas instruções I_i e I_j das transações T_i e T_j, respectivamente, estão em conflito se e só se existe um item Q acedido por I_i e I_j, e pelo menos uma dessas instruções escreveu Q.
  - I_i = **read**(Q), I_j = **read**(Q). Não existe conflito.
  - I_i = **read**(Q), I_j = **write**(Q). Existe conflito.
  - I_i = **write**(Q), I_j = **read**(Q). Existe conflito.
  - I_i = **write**(Q), I_j = **write**(Q). Existe conflito.
- Intuitivamente, um conflito entre I_i e I_j força uma ordem temporal entre ambas.
  - Se Ii e Ij são instruções consecutivas num escalonamento e não estão em conflito, o resultado seria o mesmo se a sua ordem no escalonamento fosse trocada.

Seriabilidade de Conflito

- Se um escalonamento S pode ser transformado num escalonamento S' por meio de uma série de trocas de operações sem conflito, diz-se que S e S' são **equivalentes de conflito**.
- Diz-se que um escalonamento S é **serializável de conflito** se é equivalente de conflito a um escalonamento serializado.
- O escalonamento 3 pode ser transformado no escalonamento 6 – um escalonamento serializado onde T_2 sucede a T_1, através de uma série de trocas de instruções sem conflito.
  - Logo, o escalonamento 1 é serializável de conflito.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Escalonamento 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Escalonamento 6

Seriabilidade de Conflito

- Exemplo de um escalonamento que não é serializável de conflito:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- Não é possível trocar instruções deste escalonamento para obter o escalonamento serializado <T_3,T_4> ou o escalonamento serializado <T_4,T_3>

Seriabilidade de Vista

- Sejam S e S' dois escalonamentos com o mesmo conjunto de transacções. S e S' são **equivalentes de vista** se as seguintes condições forem obedecidas:
  1. Para cada item de dados Q, se a transacção T_i lê o valor inicial de Q no escalonamento S, então a transacção T_i também lê o valor inicial de Q no escalonamento S'.
  2. Para cada item de dados Q, se a transacção T_i executa **read**(Q) no escalonamento S, e esse valor foi produzido pela transacção T_j, então a transacção T_i também lê o valor de Q produzido pela transacção T_j no escalonamento S'.
  3. Para cada item de dados Q, a transacção que efectua o **write**(Q) final no escalonamento também efectua o **write**(Q) final no escalonamento S'.
- O conceito de seriabilidade de vista apenas se baseia nas operações de **read** e **write**.

Seriabilidade de Vista

- Um escalonamento S é **serializável de vista** se é equivalente de vista a um escalonamento serializado.
- Todo o escalonamento serializável de conflito é serializável de vista.
- Exemplo de um escalonamento serializável de vista mas não serializável de conflito.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- A que escalonamento serializado é equivalente de vista?
- Todo o escalonamento que é serializável de vista mas não serializável de conflito tem **escritas cegas** (blind writes).

Outras noções de Seriabilidade

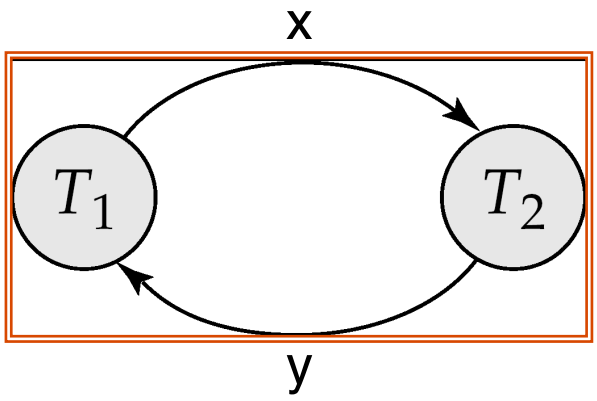
- O seguinte escalonamento produz o mesmo resultado do que o escalonamento serializado <T1,T5>, apesar de não ser nem equivalente de conflito nem equivalente de vista com ele.

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)

- Para determinar essa equivalencia seria necessário analisar operações para além das de leitura e escrita.

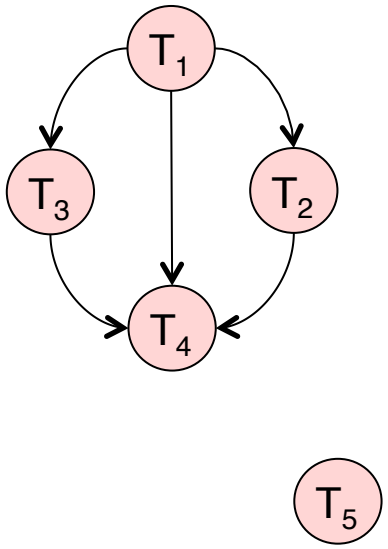
Teste de Seriabilidade

- Considere-se um escalonamento de um conjunto de transacções T_1, T_2, …, T_n.
- **Grafo de Precedências**: um grafo dirigido onde os vértices são as transacções.
- O grafo tem um arco de T_i para T_j se as duas transações têm um conflito e T_i acedeu primeiro ao item em relação ao qual o conflito existe.
- Podemos etiquetar o arco com o nome do item.



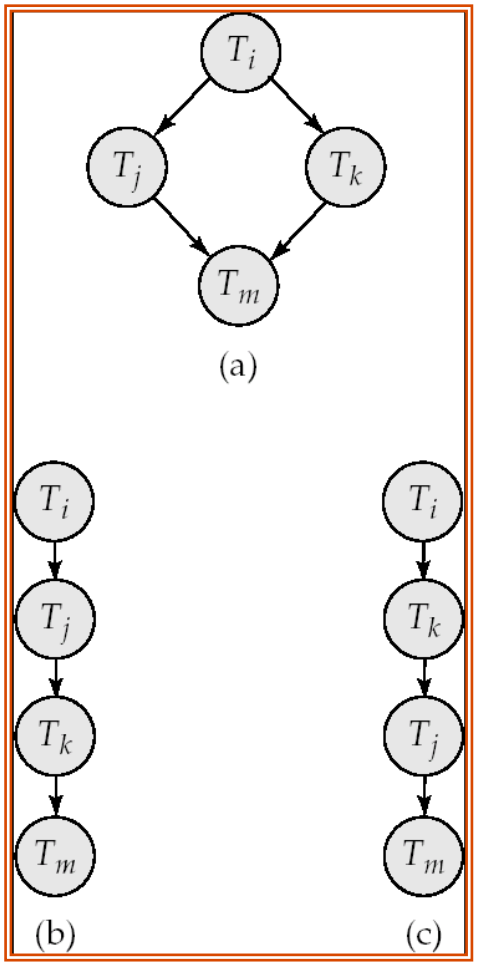
Escalonamento e Grafo de Precedência

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Teste para Seriabilidade de Conflito

- Um escalonamento é serializável de conflito se e só se o seu grafo de precedência é acíclico.
- Existem algoritmos de detecção de ciclos com complexidade temporal n^2, onde n é o número de nós no grafo.
  - Há algoritmos com complexidade temporal n+e onde e é o número de arcos.
- Se o grafo de precedência é aciclico, a ordem de serialização pode ser encontrada por uma ordenação topologica do grafo.
  - i.e. uma ordenação linear consistente com a ordem parcial do grafo.
  - Para o exemplo do slide anterior, uma serialização possível seria:
    - $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Há outras?



Teste para Seriabilidade de Vista

- O teste do grafo de precedência não pode ser usado directamente para testar a seriabilidade de vista.
  - Estensões para testar a seriabilidade de vista têm um custo exponêncial no tamanho do grafo de precedência.
- O problema de verificar se um escalonamento é serializável de vista pertence à classe de problemas NP-completo.
  - A existência de algoritmos eficientes é muito pouco provável.
- No entanto, há algoritmos práticos que verificam condições suficientes para seriabilidade de vista, que podem ser usados.

Escalonamentos Recuperáveis

- É necessário abordar o efeito de falhas em transacções que executam de forma concorrente.
- **Escalonamento recuperável**: se uma transacção T_j lê um item de dados anteriormente escrito por uma transacção T_i, então a operação de confirmação (commit) de T_i aparece antes da operação de confirmação de T_j.
- O seguinte escalonamento não é recuperável se T_9 confirma imediatamente após a leitura.

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

- Se T_8 abortar, T_9 teria lido (e possivelmente mostrado ao utilizador) um estado inconsistente da base de dados. O sistema de gestão da base de dados deve garantir que os escalonamentos são recuperáveis.

Reposições em Cascata

- **Reposições em Cascata** (cascading rollbacks) – a falha numa transacção leva a uma série de reposições (rollbacks) de transacções.
- Considerando o seguinte escalonamento onde nenhuma das transações procedeu à confirmação (logo o escalonamento é recuperável)

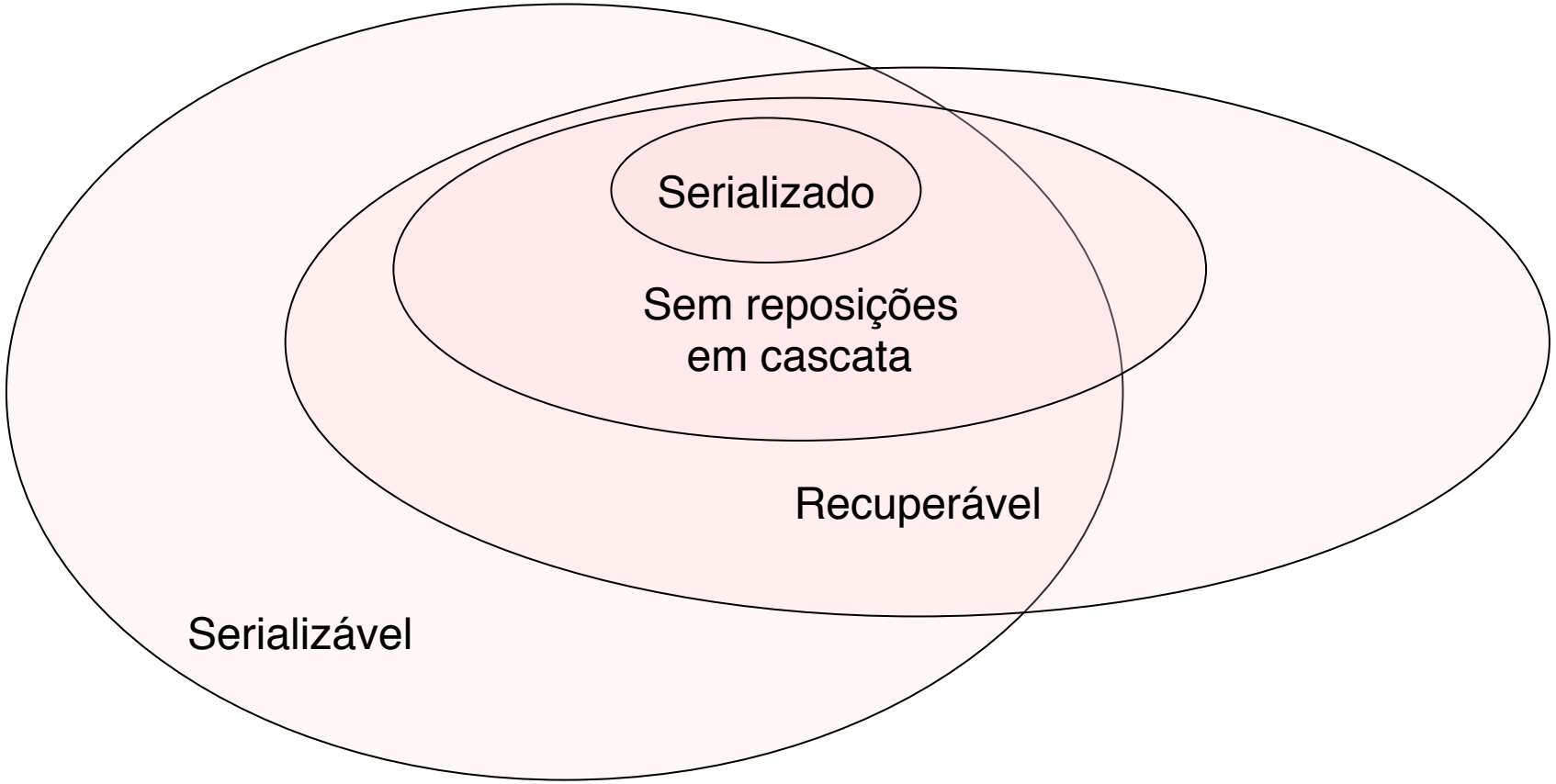
T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- Se T_{10} falha, T_{11} e T_{12} tem que ser repostas.
- Pode levar ao defazer de uma quantidade significativa de trabalho.

Escalonamentos sem Reposições em Cascata

- **Escalonamentos sem Reposições em Cascata** (Cascadeless Scheules): escalonamentos onde reposições em cascata não acontecem. Para cada par de transacções T_k e T_j tal que T_j lê um item de dados anteriormente escrito por T_k, a operação de confirmação de T_k tem que aparecer antes da operação de leitura de T_j.
- Todo o escalonamento sem reposições em cascata é recuperável.
- É desejável restringir aos escalonamentos sem reposições em cascata.

Classes de Escalonamentos



Controle de Concorrência vs. Testes de Seriabilidade

- Um sistema de gestão de bases de dados deve fornecer um mecanismo que garanta que todos os escalonamentos possíveis sejam:
  - Serializáveis de conflito ou de vista, e
  - Recuperáveis e, preferencialmente, sem cascata.
- Uma política segundo a qual apenas uma transacção pode executar de cada vez gera escalonamentos serializados, mas fornece um fraco grau de concorrência.
- Dado um conjunto de transacções, testar todos os possíveis escalonamentos, à priori, para encontrar um serializável é impraticável:
  - Ingénuo: combinatória enorme de escalonamentos
  - Estático: Assume conhecimento completo das transacções à partida
- Testar se um escalonamento é serializável *depois* da sua execução é um pouco tarde de mais!
- **Objectivo**: desenvolver protocolos de controle de concorrência que garantam seriabilidade.

Controle de Concorrência vs. Testes de Seriabilidade

- Protocolos de controlo de concorrência permitem escalonamentos concorrentes, mas garantem que os escalonamentos são serializáveis de conflito/vista, e são recuperáveis e sem reposições em cascada.
- Normalmente os protocolos de controlo de concorrência não examinam o grafo de precedências à medida que este é criado.
  - Em vez disso, um protocolo impõe uma disciplina que evita escalonamentos não serializáveis.
  - Estes protocolos serão estudados noutras cadeiras.
- Diferentes protocolos de controlo de concorrência oferecem diferentes compromissos entre a quantidade de concorrência que permitem e a quantidade de overheads em que incorrem.
- Os testes de seriabilidade ajudam a perceber a razão da correcção de um protocolo de controlo de concorrência.

Níveis Fracos de Consistencia

- Algumas aplicações estão dispostas a conviver com formas mais fracas de consistência, permitindo escalonamentos que não são serializáveis.
  - E.g. uma transacção apenas de leitura que pretende obter uma valor aproximado do saldo total de todas as contas.
  - E.g. valores estatísticos calculados para optimização de consultas.
  - Estas transacções não têm que ser serializadas em relação a outras transacções.
- Compromisso entre correcção e performance.

Níveis de Consistencia em SQL

- **serializable** – valor por omissão.
- **repeatable read** – apenas registo confirmados (committed) são lidos, e leituras repetidas do mesmo registo têm que obter o mesmo valor. No entanto, uma transacção pode não ser serializável – pode encontrar registos inseridos por uma transacção mas não por outras.
- **read committed** – apenas registos confirmados podem ser lidos, mas leituras sucessivas de um registo podem obter valores diferentes (mas confirmados).
- **read uncommitted** – mesmo registos não confirmados podem ser lidos.
- Níveis mais baixos de consistência são úteis para obter informação aproximada da base de dados.

Fenómenos a prevenir

- **Leituras sujas** (dirty reads): uma transacção lê dados escritos por uma outra transacção que ainda não foi confirmada
- **Leituras não repetíveis**: uma transacção volta a ler dados anteriormente lidos e descobre que outra transacção confirmada alterou os dados.
- **Leituras fantasma**: uma transacção volta a executar uma consulta que devolve um conjunto de tuplos que satizfazem uma dada condição, e descobre que outra transacção confirmada inseriu tuplos adicionais que satizfazem essa condição.

Nível de isolamento	Leituras sujas	Leituras não repetíveis	Leituras Fantasma
Read uncommitted	possível	possível	possível
Read committed	impossível	possível	possível
Repeatable read	impossível	impossível	possível
Serializable	impossível	impossível	impossível

Níveis de Consistência e Oracle

- De acordo com os manuais, o Oracle fornece três níveis de consistência:
  - Read committed (por omissão)
    - **set transaction isolation level read committed**
  - Serializable
    - **set transaction isolation level serializable**
  - Read only (não previsto no Standard SQL)
    - **set transaction read only**
- Para alterar o nível de isolamento de todas as transacções de uma sessão:
  - **alter session set isolation level = [read committed | serializable]**
- Na verdade, em vez do nível serializável, o Oracle fornece um nível de consistência mais fraco, não previsto no standard SQL, conhecido como *Snapshot Isolation*.

Isolamento *Snapshot*

- Nível de isolamento mais fraco do que a seriabilidade.
- Garante que todas as operações de leitura numa transacção observam um *snapshot* consistente da base de dados.
  - Normalmente o *snapshot* tem os valores confirmados no início da transacção (ou aquando da primeira operação de leitura).
  - Se, no fim da transacção, as operações de escrita estiverem em conflito com outras transacções concorrentes desde o snapshot, a transacção falha.
- Permite mais concorrência do que o nível serializável.
- Pode causar anomalias (escritas enviesadas/write skews)

Escritas Enviesadas

- Resultam da falha de detecção de conflictos leitura-escrita.
- Exemplo:
  - Considere-se uma base de dados com 2 itens, I1 e I2, com uma restrição impondo que I1+I2≥0.
  - Num dado momento, quando tanto I1 como I2 contêm o valor 10, são iniciadas duas transacções.
  - T1 (resp. T2) decrementam o valor de I1 (resp. I2) em 10.
  - Cada uma das transacções, por sí só, é consistente, e não há qualquer conflito nos valores actualizados, pelo que ambas são bem sucedidas!
  - No entanto, nenhuma serialização sucederia.
- Este fenómeno pode ser remediado impondo conflictos escrita-escrita
  - E.g. criando um item que armazena I1+I2, que seria actualizado por ambas as transacções.