Transações

- Tópicos:
  - Conceito de Transação
  - Estado de uma Transação
  - Execução Concorrente
  - Seriabilidade
  - Controle de Concorrência
  - Níveis Fracos de Consistência em SQL
- Bibliografia:
  - Capítulo 14 do livro recomendado

Manipulação concorrente de dados

- Com a DML do SQL é possível consultar e fazer alterações nos dados de uma Base de Dados
  - A linguagem é declarativa e permite fazer programas que manipulam a Base de Dados abstraindo da forma concreta de armazenamento dos dados
- É possível ter vários programas (e utilizadores) a aceder simultaneamente aos dados
- A manipulação dos dados deve ser "transparente" para cada um:
  - Cada programa deve poder aceder aos dados independentemente de haver, ou não, vários outros a aceder ao mesmo tempo
  - O facto de haver vários programas a aceder em simultâneo aos dados, não deve por em causa a consistência dos dados
    - A verificação das restrições de integridade deve ser independente dos acessos simultâneos
- Será que o que vimos até agora chega para garantir isto?
  - Como é que o sistema se deve comportar quando há acessos simultâneos?

Conceito de Transação

- Considere um programa que consulta o saldo de uma conta, e depois faz um levantamento
  - Faz uma operação de **select** seguida de um **update** do saldo
  - O que acontece se pelo meio outro programa fizer também um levantamento e o saldo da conta deixe de ser suficiente para fazer o primeiro levantamento?
- Considere que um programa quer fazer uma transferência de X€ da conta A para a conta B
  - Começa por fazer um **update** para somar X ao saldo da conta B, e depois faz um **update** para subtrair X ao saldo da A
  - O que acontece se, pelo meio, outro programa fizer um levantamento da conta A, e esta deixe de ter saldo suficiente?
  - E o que acontece se, pelo meio, a base de dados for desligada?!
- O ideal era que as duas operações, em ambos os exemplos, fossem consideradas como um "bloco" que ou é todo executado, ou não o é de todo, e que pelo meio não fossem "visíveis" operações de outros programas

Conceito de Transacção

- Uma transação é uma unidade de execução de um programa que consulta e altera os dados
- Uma transação tem que encontrar a base de dados num estado consistente
- Quando uma transação termina com sucesso (confirmada/committed), a base de dados tem que ficar num estado consistente
- Se uma transação é abortada, a base de dados tem que ser restaurada para o seu estado anterior ao início da transação
- Depois da confirmação (commit) de uma transação, as alterações feitas à base de dados devem perdurar, mesmo se houver falhas no sistema
- O sistema de gestão de Bases de Dados deve suportar a execução concorrente de transações

Transações em SQL

- Em SQL é possível dizer que um conjunto de ações (operações da DML) é executado como uma transação
  - A instrução **commit work** (ou apenas **commit**) termina uma transação e inicia uma nova transação
  - A instrução **rollback work** (ou apenas **rollback**) aborta uma transação (desfazendo o efeito de todas as ações desde o início da transação) e inicia uma nova transação
  - Também é possível definir uma transação colocando todas as ações entre **begin atomic** e **end atomic**
- Em vários SGBD, por *default* cada ação é executada isoladamente numa transação (i.e. como se após cada ação houvesse um **commit**)
  - É possível desligar este comportamento com

    set autocommit = off

Propriedades de uma Transação

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

Atomicidade

- Transação que transfere €50 da conta A para a conta B:
  1. **update** account

     set balance = balance – 50

     where account_number = A
  2. **update** account

     set balance = balance + 50

     where account_number = B
- **Requisito de Atomicidade**: se a transação falha (hardware ou software) após o 1º passo e antes do 2º passo, o sistema deve garantir que as modificações não se refletem na base de dados, caso contrário resultaria numa inconsistência i.e. perdia-se dinheiro.
  - ✷ Tudo ou Nada no que respeita à execução da transação.

Propriedades ACID

- Transacção que transfere €50 da conta A para a conta B:
  1. **update** account
     set balance = balance – 50
     where account_number = A
  2. **update** account
     set balance = balance + 50
     where account_number = B
- **Requisito de Consistência**: a soma de A e B permanece inalterada pela execução da transação. 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 transação deve ver uma base de dados consistente e deixar uma base de dados consistente
  - Durante a execução de uma transação, a base de dados pode estar temporariamente inconsistente.
    - Restrições só são verificadas no fim da transação.

Propriedades ACID

- Transacção que transfere €50 da conta A para a conta B:
  1. **update** account
     set balance = balance – 50
     where account_number = A
  2. **update** account
     set balance = balance + 50
     where account_number = B

  select sum(balance)
  where account_number **in** (A,B)

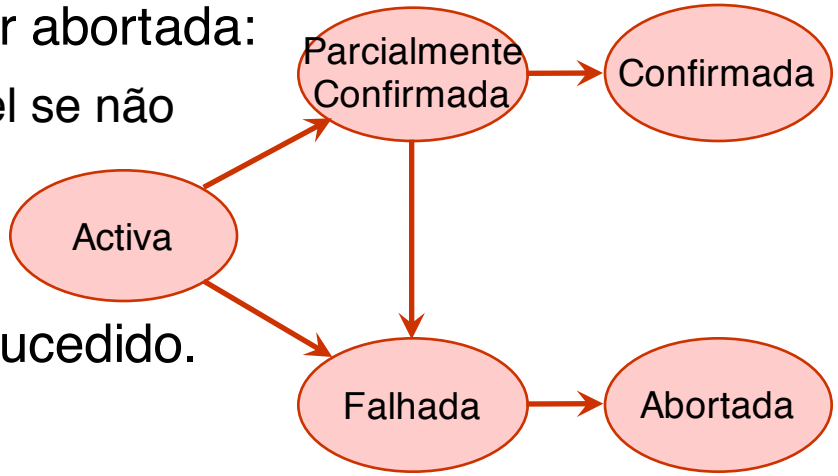
- **Requisito de Isolamento**: se entre o 1º e o 2º passos, outra transação pudesse aceder à base de dados parcialmente atualizada, veria uma base de dados inconsistente (a soma A+B seria menor do que deveria ser).
  - O isolamento pode ser garantido de forma trivial se apenas permitirmos a execução 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. **update** account
     set balance = balance – 50
     where account_number = A
  2. **update** account
     set balance = balance + 50
     where account_number = B
- **Requisito de Durabilidade**: quando o utilizador for notificado que a transação terminou (i.e. que a transferência dos €50 ocorreu), as atualizações à base de dados devem persistir mesmo que haja posteriores falhas do sistema

Estados 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 Transações

- Para garantir **atomicidade** e **durabilidade**, os SGBD implementam mecanismos de recuperação, tipicamente baseados em logs
- Duas possibilidades:
  - *Deferred modifications*:
    - Em vez de fazer as alterações diretamente na BD, à medida que a transação é executada é guardado um log com todas as alterações.
    - No fim, se a transação for confirmada o log é visitado e fazem-se todas as alterações de uma vez só
    - Se a transação for abortada, o log é apagada e a BD fica como estava
    - As queries pelo meio têm em conta a BD e os logs
  - *Immediate modifications*:
    - Todas as operações vão sendo feitas na BD e registadas no log
    - No fim, se a transação for confirmada o log é apagado e a BD fica como está
    - Se a transação for abortada o log é visitado e fazem-se as operações inversas às do log, todas de uma vez
- O Oracle usa *Deferred modifications*

Implementação de transações

- Para garantir **consistência**
  - Basta que se "desligue" a verificação das restrições de integridade pelo meio de uma transação, e que toda a verificação seja feita quando a transação está no estado de "Parcialmente confirmada"
- Este mecanismo pode ter um overhead grande na execução (em Sistemas de Bases de Dados podem perceber porquê)
- Por isso, por *default* os SGBD verificam sempre as restrições
  - Notem que isto também garante a consistência. Mas garante coisas a mais!
- Para não impor consistência pelo meio de transações
  - Adicionar **deferrable** após cada restrição (e.g. de **primary key**, **foreign key**, **check**, etc)
  - No início da transação dar o comando

set constraints all deferred

Implementação de Transações

- Uma forma muito naïve de garantir **isolamento**, seria impor que em cada momento só uma transação é que está a ser executada
  - Se outro programa quiser aceder à base de dados, tem que esperar que o programa que está a executar termine (a transação)
- Claro que isto provocaria muitas esperas!
  - E seriam desnecessárias na maior parte dos casos (bastava que os programas não estivessem a aceder aos mesmos dados!)
- SGBD permitem execução concorrente de transações
  - Esquemas de controlo de concorrência
  - Garantem que, apesar da execução ser concorrente, cada programa não vê nem é influenciado pelo que o outro está a executar
  - A ideia é encontrarem uma ordem de execução das várias ações em transações concorrentes, que aparente que cada transação está a ser executada sozinha

Escalonamentos

- **Escalonamento**: sequência de instruções que especifica a ordem cronológica pela qual as instruções de transações concorrentes são executadas
  - Um escalonamento para um conjunto de transações tem que incluir todas as instruções de todas as transações
  - Tem que preservar a ordem segundo a qual as instruções aparecem em cada transação individual
  - Para simplificar, nos slides em vez de SQL serão usadas operações **read** (em vez de **select**) e **write** (em vez de **update**, **insert** ou **delete**)
- Uma transação 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 transação que falha a sua execução tem um *abort* (rollback) como última instrução.
- **Objetivo**: encontrar escalonamentos que preservem o isolamento.
- **Escalonamento Serializado**: escalonamento *radical* onde cada transação é completamente executada antes do início da seguinte.

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(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 4

- 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 transação preserva a consistência da base de dados.
  - Logo, a execução serializada (sequencial) de um conjunto de transações preserva a consistência.
- Um escalonamento (possivelmente concorrente) é **serializável** se é equivalente a um escalonamento serializado.
- Vista simplificada de transações:
  - Ignoraremos operações distintas das instruções **read** e **write**.
  - Assumiremos que as transações podem efetuar 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 I_i e I_j 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 consecutivas 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

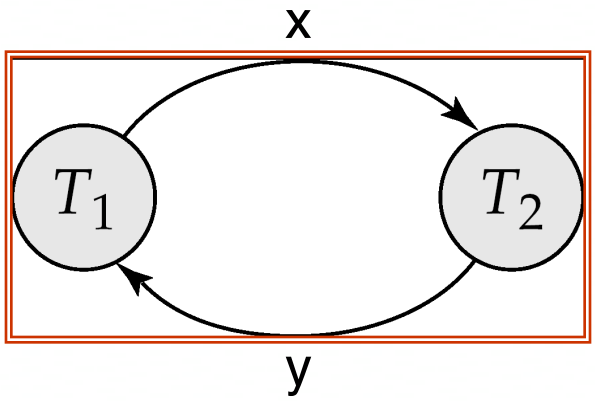
- Exemplo de um escalonamento que não é serializável de conflito:

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>
- Mas, na verdade este escalonamento não causa qualquer problema ao isolamento!
  - Por isso há outra noções de seriabilidade.
  - Mas não as estudaremos nesta Unidade Curricular

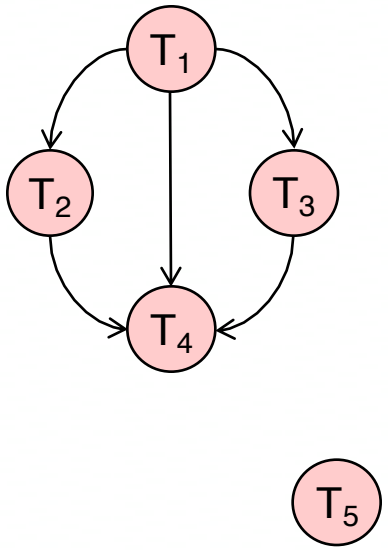
Teste de Seriabilidade

- Considere-se um escalonamento de um conjunto de transações T_1, T_2, …, T_n.
- **Grafo de Precedências**: um grafo dirigido onde os vértices são as transações.
- 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.



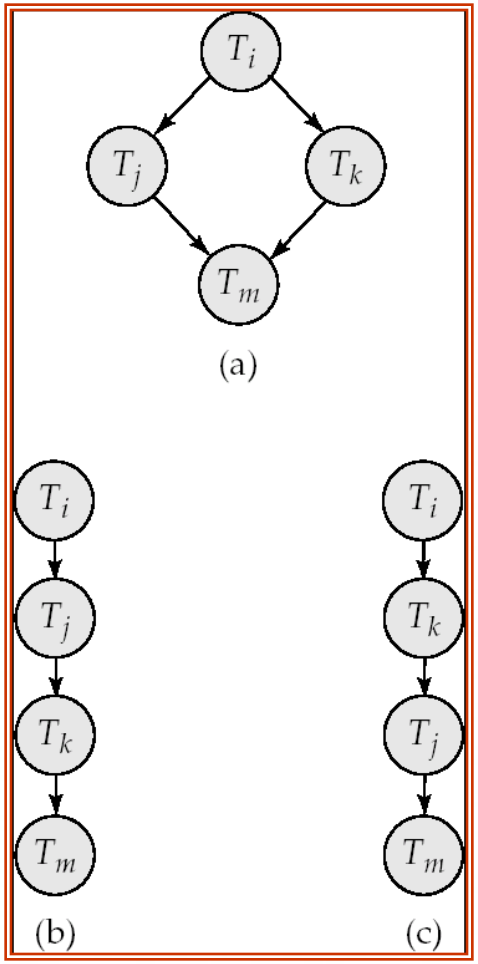
Escalonamento e Grafo de Precedência

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 acíclico, a ordem de serialização pode ser encontrada por uma ordenação topológica 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?



Controle de Concorrência

- Um sistema de gestão de bases de dados deve fornecer um mecanismo que garanta que todos os escalonamentos possíveis sejam serializáveis
  - Uma política segundo a qual apenas uma transação pode executar de cada vez gera escalonamentos serializados, mas fornece um fraco grau de concorrência.
- Dado um conjunto de transações, 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!
- **Objetivo**: desenvolver protocolos de controle de concorrência que garantam seriabilidade à medida que as transações vão executando

Controle de Concorrência

- Protocolos de controlo de concorrência permitem escalonamentos concorrentes, garantindo que os escalonamentos são serializáveis
- 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 em detalhe 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 correção de um protocolo de controlo de concorrência.

Tipos de protocolos

T1	T2
read(A)	
	write(A)
write(A)	
write(B)	
	read(A)

- Que fazer?
  - Pode-se dar o caso de não haver problema nenhum
  - Mas pode bem acontecer que haja problema
- Protocolos otimistas deixam avançar e se houver problema mais tarde, abortam as transações
- Protocolos pessimistas param antes de potenciais conflitos
  - A transação (neste caso T2) não avança até que a T1 termine
  - Nos casos em que depois não havia problema, a T2 só esteve a perder tempo!

Níveis Fracos de Isolamento

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

Níveis de Isolamento em SQL

- **serializable** – valor por omissão.
- **repeatable read** – apenas registos confirmados (committed) são lidos, e leituras repetidas do mesmo registo têm que obter o mesmo valor. No entanto, uma transação pode não ser serializável – pode encontrar registos inseridos por uma transação 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 transação lê dados escritos por uma outra transacção que ainda não foi confirmada
- **Leituras não repetíveis**: uma transação volta a ler dados anteriormente lidos e descobre que outra transação confirmada alterou os dados.
- **Leituras fantasma**: uma transação volta a executar uma consulta que devolve um conjunto de tuplos que satisfazem uma dada condição, e descobre que outra transação confirmada inseriu tuplos adicionais que satisfazem 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 Isolamento e Oracle

- 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 transações de uma sessão:
  - **alter session set isolation level = [read committed | serializable]**
- Na verdade, em vez do nível serializável, o Oracle fornece um nível de isolamento 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 transação observam um *snapshot* consistente da base de dados.
  - Normalmente o *snapshot* tem os valores confirmados no início da transação (ou aquando da primeira operação de leitura).
  - Se, no fim da transação, as operações de escrita estiverem em conflito com outras transações concorrentes desde o snapshot, a transação falha.
- Permite mais concorrência do que o nível serializável.
- Pode causar anomalias (escritas enviesadas/write skews)

Escritas Enviesadas

- Resultam da falha de deteção de conflitos 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 transações.
  - T1 (resp. T2) decrementam o valor de I1 (resp. I2) em 20.
  - Cada uma das transações, por si só, é consistente, e não há qualquer conflito nos valores atualizados, pelo que ambas são bem sucedidas!
  - No entanto, nenhuma serialização sucederia.
- Este fenómeno pode ser remediado impondo conflitos escrita-escrita
  - E.g. criando um item que armazena I1+I2, que seria atualizado por ambas as transações.