TÉCNICO LISBOA

# Bases de Dados

## T26 - Transações Parte II

Prof. Daniel Faria

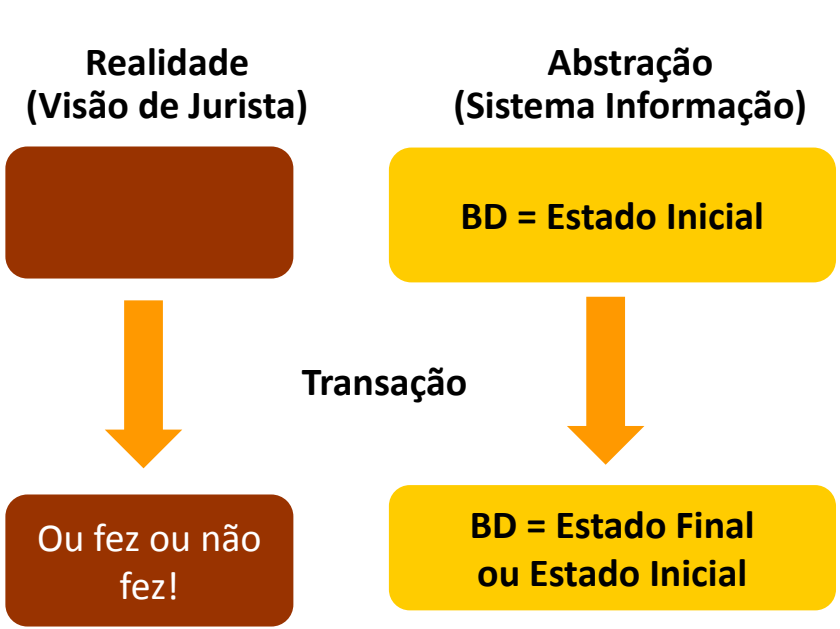
# Sumário

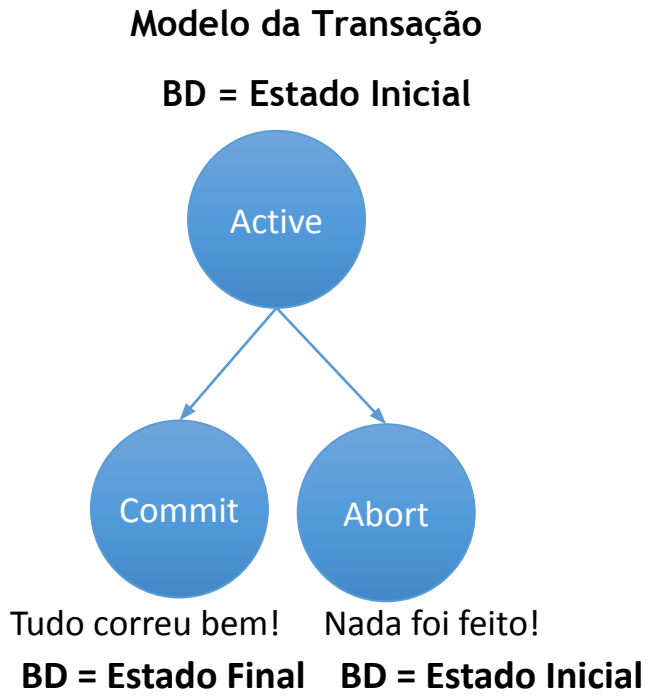
- Recapitulação Breve
- Transações e Restrições de Integridade
- Escalonamento de Transações
- Isolamento
- Locking
- Recuperação

TÉCNICO LISBOA

# Recapitulação Breve

# Abstração de um Sistema Transacional

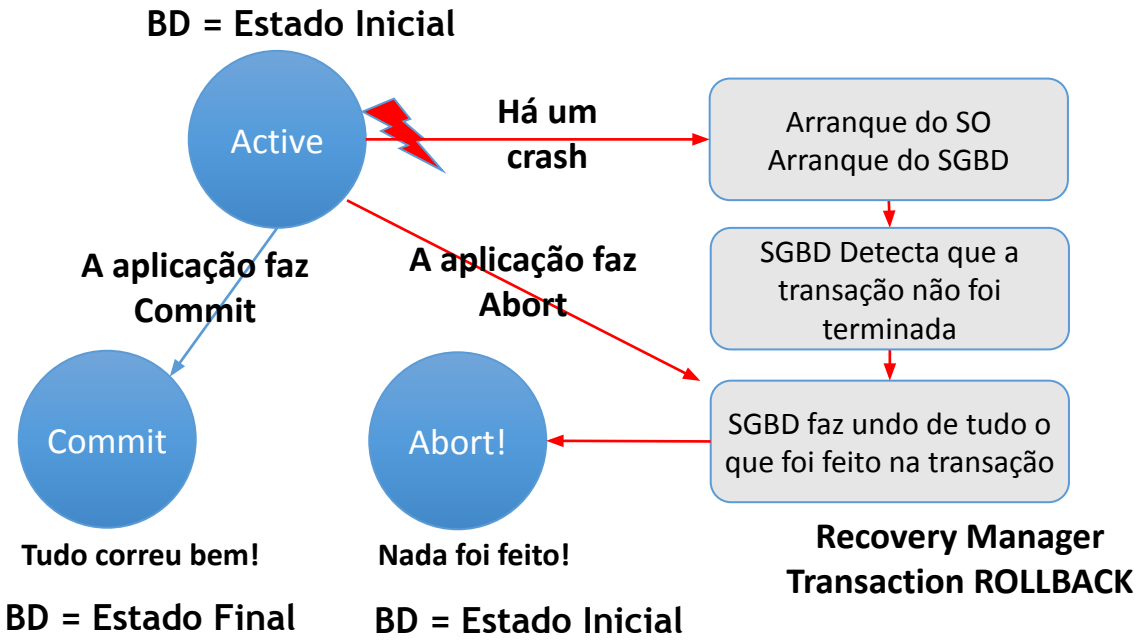



**2 Fins Possíveis**

TÉCNICO LISBOA

# Caminhos para a Conclusão




TÉCNICO LISBOA

# Propriedades ACID

- **Atomicidade:**
  - E.g. se a transação falhar entre os passos 4–6, os passos 1–3 ficam sem efeito
- **Consistência**
  - E.g. a soma A+B tem de ser igual antes e depois da transação

| | |
|---|---|
| 1 | $T_i$: **read**($A$) |
| 2 | $A := A - 50$ |
| 3 | **write**($A$) |
| 4 | **read**($B$) |
| 5 | $B := B + 50$ |
| 6 | **write**($B$) |

- **Isolamento**
  - E.g. nenhuma outra operação deve ler os valores de A e B entre os passos 3 e 6 (veria valores inconsistentes)
- **Durabilidade**
  - Se a transação termina com sucesso, as alterações são definitivas, mesmo que haja falhas de *hardware* ou *software*

# Definição de Transações em SQL

- Em SQL, qualquer consulta inicia implicitamente uma transacção, mas com uma única instrução
- Para incluir várias instruções SQL na mesma transação, deve-se preceder o grupo de instruções com
  - **`START TRANSACTION`** (ou **`BEGIN`**)
- E terminar com:
  - **`COMMIT`**: torna os resultados permanentes
  - **`ROLLBACK`**: desfaz todas as alterações feitas

# Exemplo

```
START TRANSACTION;

--Verificar saldo
SELECT balance FROM account WHERE account_number = 'A-101';

--Transferir 350€ da conta A-101 para a conta A-102:
UPDATE account SET balance = balance - 350
    WHERE account_number = 'A-101';
UPDATE account SET balance = balance + 350
    WHERE account_number = 'A-102';

COMMIT;
```

https://www.postgresql.org/docs/current/tutorial-transactions.html

TÉCNICO LISBOA

# Transações em SQL

- Se START TRANSACTION for omitido
  - Cada consulta é uma transacção
    - Se houver erros, ROLLBACK automático
    - Se não houver erros, COMMIT automático
- Erros a meio de uma transação causam *abort*
  - Deixa de ser possível continuar a transação
- Podemos usar SAVEPOINT para gravar um ponto seguro e ROLLBACK TO para regressar a esse ponto
  - É a única maneira de recuperar do estado de *abort* sem fazer ROLLBACK total

TÉCNICO LISBOA

# Transações e Restrições de Integridade

# Restrições de Integridade

- Restrições de integridade de tabela ou coluna, por defeito, são verificadas após cada comando
  - Mesmo durante uma transação, não é possível violá-las
  - Este comportamento pode ser controlado com os parâmetros:
    - **`DEFERRABLE`**
      - **`INITIALLY IMMEDIATE`**
      - **`INITIALLY DEFERRED`**
    - **`NOT DEFERRABLE`**

# Restrições de Integridade

- A configuração por defeito de restrições de integridade é **`NOT DEFERRABLE`**
- Se uma restrição for definida como **`DEFERRABLE INITIALLY DEFERRED`**, a sua verificação é adiada para o final da transação
  - Não faz diferença para operações atómicas (que são transações de um só statement)
- **`NOT NULL`** e **`CHECK`** não podem ser **`DEFERRABLE`** (o que faz sentido) mas os restantes tipos de restrições podem (**`PRIMARY KEY`**, **`UNIQUE`**, **`FOREIGN KEY/REFERENCES`**)

# Triggers

- Triggers que servem para definir restrições de integridade devem ser definidos com a opção **`CONSTRAINT`**
  - **`CREATE CONSTRAINT TRIGGER`**
- Constraint triggers têm de despoletar **`AFTER INSERT/UPDATE`** e é esperado que produzam uma excepção quando a restrição de integridade é violada
- Constraint triggers (e apenas constraint triggers) podem ser configurados com **`DEFERRABLE INITIALLY DEFERRED`** (após **`ON`** *`table_name`*) por forma a serem verificados apenas no final de transações

# Restrições de Participação Obrigatória

- Como já discutimos, restrições de participação obrigatória de uma tabela *A* numa segunda tabela *B* (que tem chave estrangeira para *A*) requerem:
  - Chave estrangeira circular de *A* para *B*
  - Constraint trigger para verificar a participação
- Em qualquer dos casos pelo menos uma das restrições em causa (uma ou ambas as chaves estrangeiras, ou a chave estrangeira e/ou o trigger) tem de ser configurada com `DEFERRABLE INITIALLY DEFERRED` ou não será possível inserir dados em nenhuma das tabelas
  - Ainda assim só será possível inserir dados em transações

TÉCNICO LISBOA

# Escalonamento de Transações

# Transações e Concorrência

- Execução concorrente de transações é vantajosa em termos de desempenho, permitindo
  - Maior utilização de storage, memória e CPUs, e consequente melhoria de desempenho (em transações/tempo)
    - E.g. uma transação usa o CPU enquanto outra escreve na storage
  - Redução do tempo médio de resposta das transações: as curtas não precisam de esperar pelas longas
- Pode no entanto ser problemática em termos de consistência

# Escalonamento (Scheduling)

- Sequência de instruções que especificam a ordem cronológica na qual as instruções de transações concorrentes são (potencialmente) executadas
- Um escalonamento de um conjunto de transações inclui todas as instruções de todas as transações
  - Tem de preservar a ordem na qual as instruções aparecem em cada transação individual
- Assumimos que as transações só partilham informação pelo read/write que fazem sobre os registos da BD

# Escalonamento (Scheduling)

**Exemplo:**

- **Transação T1**:
  - Transferir €50 da conta A para a conta B
- **Transação T2:**
  - Transferir 10% do saldo da conta A para a conta B

# Escalonamento em Série

**1**

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

**2**

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

- Escalonamentos em série **não equivalentes**, pois ordem afecta o resultado
  - T2 transfere menos €5 em 1 do que em 2

# Escalonamento Serializável

**1**

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

**3**

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

- Escalonamentos **equivalentes**
  - Embora 3 não seja em série diz-se que é **serializável**

# Escalonamento Não Serializável

**3**

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

**4**

| $T_1$ | $T_2$ |
|---|---|
| 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 ($B$) | |
| commit | |
| | $B := B + temp$ |
| | write ($B$) |
| | commit |

- O escalonamento 4 não preserva A+B, portanto não é serializável
  - T1 modifica B entre a leitura e a escrita de T2

# Serializabilidade

- Pressuposto fundamental: cada transação preserva a consistência da base de dados
  - Portanto, a execução em série de um conjunto de transações preserva a consistência da base de dados
- Um escalonamento (potencialmente concorrente) é **serializável** se é equivalente a um escalonamento em série

# Visão Simplificada das Transações

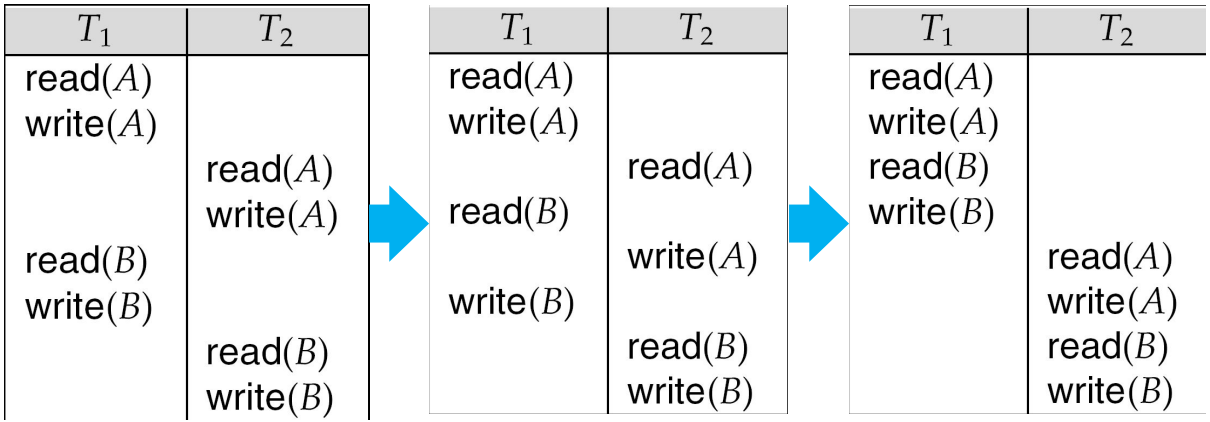
- Só consideramos as instruções de leitura e escrita na análise: escalonamentos só contêm instruções *read* e *write*

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

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

# Serialização de Conflitos

- Existe conflito se as instruções de T1 e T2 tiverem de ser obrigatoriamente executadas numa certa ordem
  - Se não houver conflito, podem ser trocadas



# Conflitos na Ordem das Instruções

Num escalonamento onde duas transações $T_1$ e $T_2$ acedem ao mesmo objeto $Q$

- $T_1$: ***read***(Q) e $T_2$: ***read***(Q)
  - Não há conflito pois a ordem é indiferente
- $T_1$: ***read***(Q) e $T_2$: ***write***(Q)
  - Há conflito: a ordem é relevante
- $T_1$: ***write***(Q) e $T_2$: ***read***(Q)
  - Há conflito: a ordem é relevante
- $T_1$: ***write***(Q) e $T_2$: ***write***(Q)
  - Há conflito: a ordem não afecta nem $T_1$ nem $T_2$ mas afeta a próxima instrução de ***read***(Q)

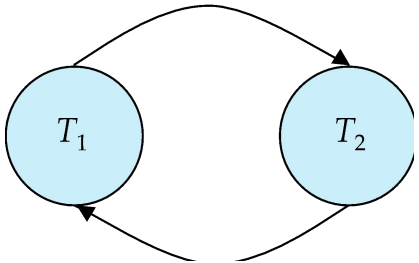
# Serializabilidade de Conflitos

- Se um escalonamento S pode ser transformado num escalonamento S’ mediante uma série de trocas de ordem de execução de instruções, dizemos que S e S’ são **equivalentes sem conflitos**
- Um escalonamento S tem **conflitos serializáveis** se é equivalente a um escalonamento em série sem conflitos

# Teste de Serializabilidade

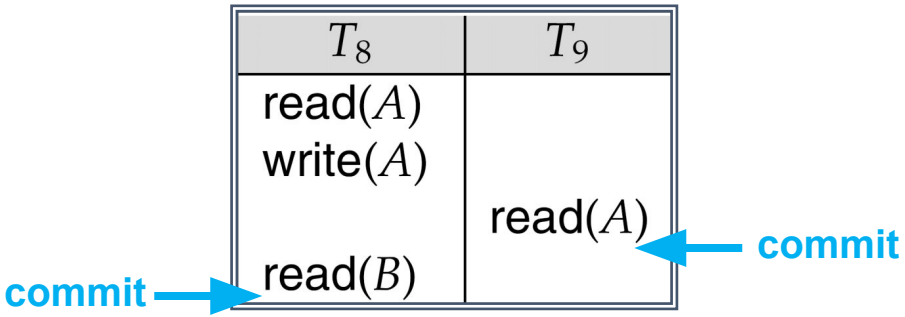
Forma simples de determinar se um escalonamento é serializável:

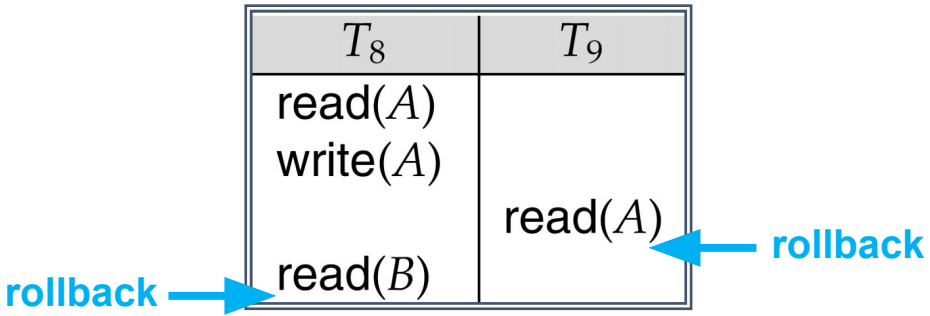
- Construir um grafo de precedências
  - Nós são transações ($T_1$, $T_2$, ..., $T_n$)
  - Arcos são conflitos ($T_i \rightarrow T_k$ )
- Desenha-se um arco de $T_i$ para $T_k$ se
  - $T_i$: **write**($Q$) antes de $T_k$ : **read**($Q$)
  - $T_i$: **read**($Q$) antes de $T_k$: **write**(Q)
  - $T_i$: **write**($Q$) antes de $T_k$: **write**($Q$)
- **Ausência de ciclos implica serializabilidade**



# Recuperação em Escalonamentos

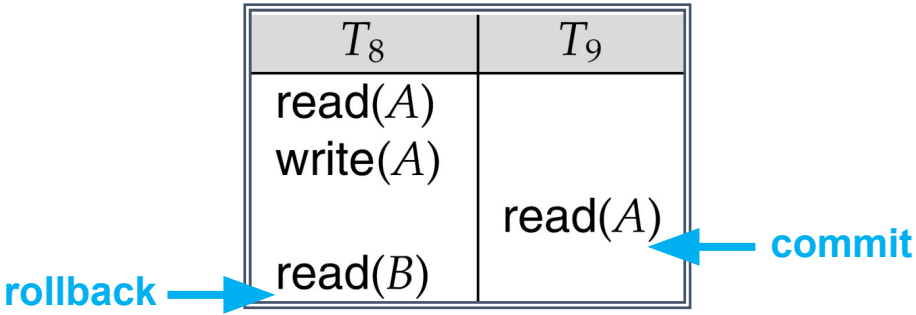
- Uma transacção que se completa com sucesso tem uma instrução de *commit* como última instrução
- Caso contrário tem uma instrução de *rollback*



# Escalonamento Não Recuperável

- Um escalonamento onde uma transação faz commit de dados modificados por outra transação é não recuperável



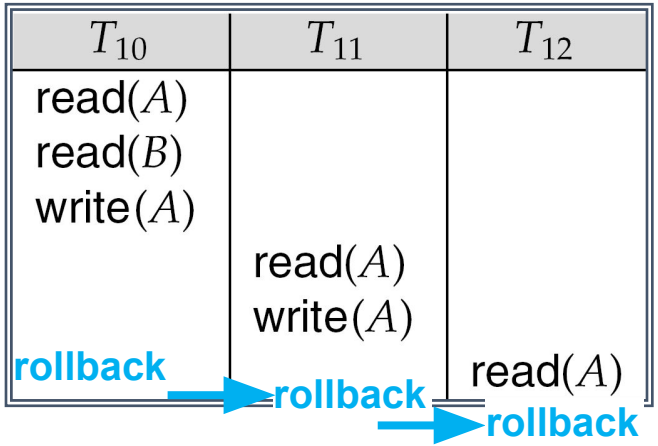

- Problema: $T_9$ já fez commit e não pode ser rolled back (durabilidade), portanto $T_8$ não pode ser rolled back (ou $T_9$ fica invalidada)

# Escalonamento Recuperável

- Um escalonamento não recuperável nunca pode ser permitido: todos os escalonamentos têm de ser recuperáveis
- Um escalonamento é **recuperável** sse, para cada par de transações $T_x$ e $T_y$:
  - Sempre que $T_x$ executa *write*(O) antes de $T_y$ executar *read*(O), $T_x$ faz *commit* antes de $T_y$ fazer *commit*

# Rollbacks em Cadeia

- Um erro ou rollback de uma transação pode levar a abort/rollback de várias outras transações (mesmo num escalonamento recuperável)




- Este comportamento é indesejado, pois tem o potencial de custar muito mais tempo do que é poupado com a concorrência

# Escalonamento Recuperável

- Um SGBD não deve permitir escalonamentos passíveis de causar rollbacks em cadeia
- Um escalonamento é **livre de rollbacks em cadeia** sse, para cada par de transações $T_x$ e $T_y$:
  - Sempre que $T_x$ executa *write*(O) antes de $T_y$ executar *read*(O), também $T_x$ faz *commit* antes de $T_y$ executar *read*(O)
    - Isto implica que também é um escalonamento recuperável

TÉCNICO LISBOA

# Isolamento

# Níveis de Isolamento

- Algumas operações não exigem 100% de consistência
  - Uma consulta que determina (de forma aproximada) o saldo médio de todas as contas
  - O cálculo de dados estatísticos para otimização de operações
- Estas transações não precisam de ser serializadas com outras
  - Poupam-se as verificações e deixa-se a transação correr livremente em paralelo
  - Compromisso entre exatidão dos resultados e desempenho do sistema

# Terminologia

- **Dirty Read / Read Uncommitted:** uma transação lê um valor escrito por outra transação antes da segunda transação fazer commit
- **Non-Repeatable Read:** uma transação lê um valor e uma segunda transação modifica o mesmo valor e faz commit antes da primeira fazer commit (se a primeira voltar a ler, o valor mudou)
- **Phantom Read:** uma transação lê um conjunto de linhas que satisfazem uma condição e uma segunda transação insere uma nova linha que satisfaz essa condição e faz commit

# Níveis de Isolamento

- Parte do standard SQL, implementadas mais estritamente no PostgreSQL

| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read | Serialization Anomaly |
|---|---|---|---|---|
| Read uncommitted | Allowed, but not in PG | Possible | Possible | Possible |
| Read committed (Default) | Not possible | Possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Allowed, but not in PG | Possible |
| Serializable | Not possible | Not possible | Not possible | Not possible |

- Podem ser configuradas numa de duas maneiras:
  - `START TRANSACTION ISOLATION LEVEL level`
  - `SET TRANSACTION ISOLATION LEVEL level`

https://www.postgresql.org/docs/current/transaction-iso.html

# Níveis de Isolamento

- **Mecanismos de Concorrência:**
  - Para garantir o nível de isolamento desejado, o SGBD tranca (lock) automaticamente linhas ou tabelas lidas/escritas para leitura e/ou escrita por outras transações
  - O PostgreSQL usa *predicate locking*, determinando quando operações de escrita teriam um impacto em operações de leitura prévias, caso tivessem sido executadas antes (em vez de trancar toda a tabela para escrita)

TÉCNICO LISBOA

# Locking

# Locking

- Os comandos **LOCK** e **UNLOCK**, declarados dentro de transações, permitem controlar explicitamente o acesso concorrente aos dados
  - No PostgreSQL não existe **UNLOCK**, os locks são automaticamente terminados no fim da transação (o que é a prática desejada)
- Os locks podem ser a nível da **tabela** ou da **linha** e **exclusive** (impedem leitura e escrita) ou **shared** (impedem apenas escrita)
  - O PostgreSQL oferece bastantes variantes de locking dentro destas categorias: https://www.postgresql.org/docs/current/explicit-locking.html

# Locking

- A aquisição de um lock sobre uma entidade garante que esta não muda de estado por ação de uma outra transação
- Uma transação que não consiga adquirir um lock ficará bloqueada (em wait-state) até conseguir
- Num bom protocolo de acesso aos dados, uma transação:
  - Adquire um lock partilhado (de leitura) antes de tentar ler o valor de um tuplo ou um lock exclusivo (de escrita) antes de tentar de alterar o valor de um tuplo
  - Liberta todos os locks com um unlock correspondente no COMMIT

# Locking


**T1:** LOCK-X (balance)

**T1: READ(balance)**

**T1:** *balance := balance+100*

**T1: WRITE(balance)**

**T1:** UNLOCK(balance)

**T2:** LOCK-X(balance)

*wait…*

*wait…*

*wait…*

**T2: READ(balance)**

**T2:** *balance := balance+200*

**T2: WRITE(balance)**

**T2:** UNLOCK(balance)

# Compatibilidade de Locks

## Locks de Tabela

| Requested Lock Mode | Existing Lock Mode | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ACCESS SHARE | ROW SHARE | ROW EXCL. | SHARE UPDATE EXCL. | SHARE | SHARE ROW EXCL. | EXCL. | ACCESS EXCL. |
| ACCESS SHARE | | | | | | | | X |
| ROW SHARE | | | | | | | X | X |
| ROW EXCL. | | | | | X | X | X | X |
| SHARE UPDATE EXCL. | | | | X | X | X | X | X |
| SHARE | | | X | X | | X | X | X |
| SHARE ROW EXCL. | | | X | X | X | X | X | X |
| EXCL. | | X | X | X | X | X | X | X |
| ACCESS EXCL. | X | X | X | X | X | X | X | X |

# Compatibilidade de Locks

## Locks de Linha

| Requested Lock Mode | Current Lock Mode | | | |
|---|---|---|---|---|
| | FOR KEY SHARE | FOR SHARE | FOR NO KEY UPDATE | FOR UPDATE |
| FOR KEY SHARE | | | | X |
| FOR SHARE | | | X | X |
| FOR NO KEY UPDATE | | X | X | X |
| FOR UPDATE | X | X | X | X |

# Locking

- Locks por si sós **não são suficientes** para garantir a serializabilidade

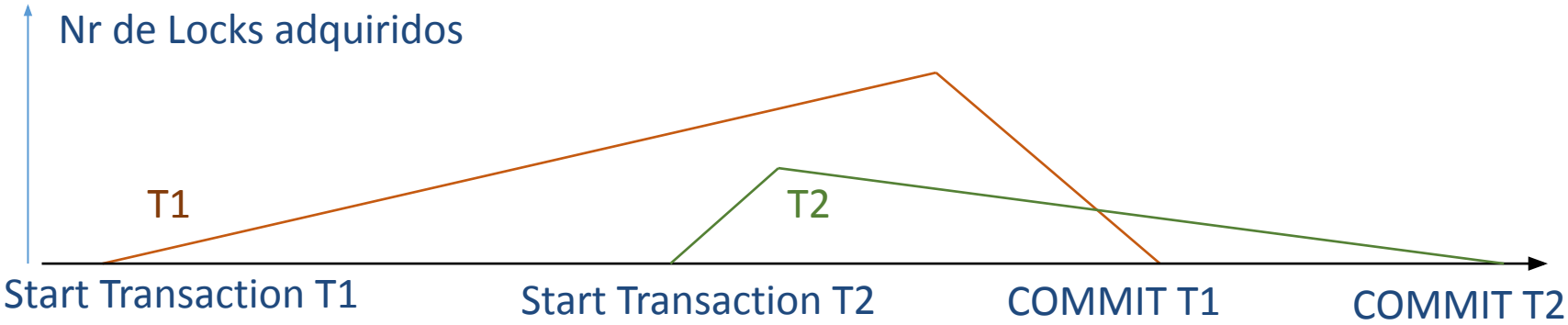
| $T_1$: | **read**($B$) |
|---|---|
| | $B := B - 50$ |
| | **write**($B$) |
| | **read**($A$) |
| | $A := A + 50$ |
| | **write**($A$) |

| $T_2$: | **read**($A$) |
|---|---|
| | **read**($B$) |
| | **display**($A$+$B$) |

- $T_2$ pode dar um resultado incoerente se executada em paralelo com $T_1$
  - Se $A$ e $B$ são atualizados por $T_1$ entre a leitura de $A$ e $B$ por $T_2$, então a soma mostrada está errada!

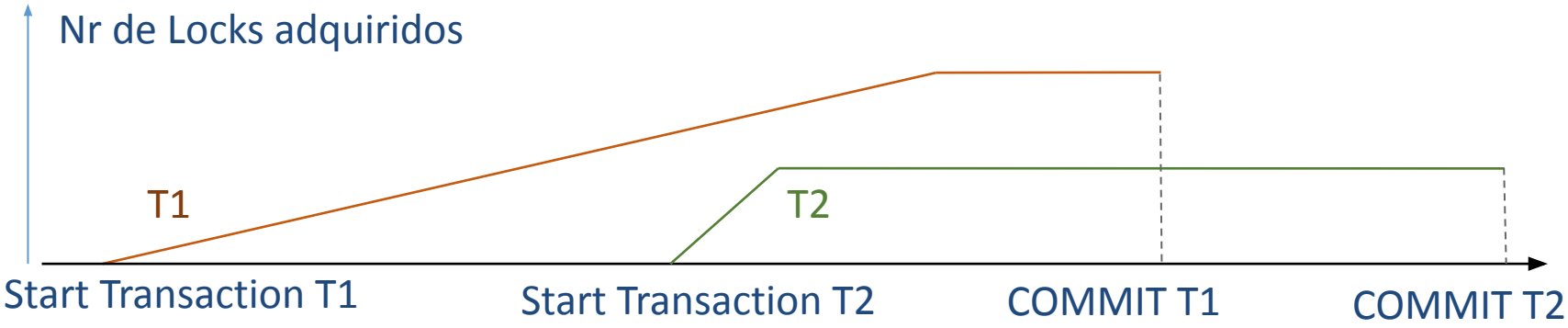
# Two-Phased Locking (2PL)

- Todas as ações de lock antecedem todas as ações de unlock, ou por outras palavras, **depois de um unlock não pode haver locks**
- Se as transações seguirem este protocolo, é garantido que o resultado da execução concorrente é sempre serializável
- A ordem de execução correspondente é definida pela ordem com que as transações adquirem o seu último lock



# Strict Two-Phased Locking (S2PL)

- Igual a 2PL + unlock de locks (exclusivos) só acontece após commit ou abort
  - A única opção possível no PostgreSQL dada ausência de unlock
- Garante recuperabilidade e liberdade de rollbacks em cadeia



# Deadlocks

- Surgem quando duas transações concorrentes precisam ambas de um lock não compatível com o lock que a outra transação já têm
  - E.g. $T_1$ cria um lock exclusivo sobre *B*, $T_2$ cria um lock exclusivo sobre *A*, $T_1$ precisa de um lock sobre *A* para prosseguir e $T_2$ precisa de um lock sobre *B* para prosseguir
- O PostgreSQL tem algoritmos para detectar e corrigir estas situações, abortando uma das transações

TÉCNICO LISBOA

# Recuperação

# Recuperação

- Para recuperar as ações de transações abortadas (atomicidade) os SGBD mantêm um ficheiro log onde todas as alterações à base de dados são gravadas
- Este mecanismo também é usado para recuperar de falhas de sistema
    - Todas as transações ativas no momento da falha são abortadas e as suas alterações revertidas quando o sistema é reiniciado

# Logging

- O ficheiro log grava as seguintes informações
  - Operações de escrita: se $T_i$ escreve num objeto, os valores antigo e novo são guardados
    - O ficheiro log tem de ser gravado antes das páginas da tabela e índices em memória serem alteradas
  - Operações de commit e abort: se $T_i$ faz commit ou abort, um registo desta informação é escrito no log
- Após commit, a informação do log sobre a transação deixa de ser relevante (o log é organizado por transação para facilitar a eliminação)

TÉCNICO LISBOA