

SGBDs Relacionais

– Controle de Transação e Concorrência

Prof. Dr. Ives Renê V. Pola

ivesr@utfpr.edu.br

Departamento Acadêmico de Informática – DAINF
UTFPR – Pato Branco DAINF
UTFPR
Pato Branco - PR

Esta apresentação mostra como é feito o controle da concorrência em um SGBD Relacional, usando os conceitos de Transação para o controle das atualizações, e mostra os principais comandos da Linguagem SQL associados.



Roteiro

- 1 Controle de Transação
- 2 Controle de Concorrência
- 3 Sintaxe em SQL

Gerenciamento de Transações

Conceitos

Existem três grandes grupos de eventos que podem danificar o conteúdo de uma base de dados:

- ❶ **Erros de lógica** nos aplicativos;
- ❷ **Falhas** no hardware ou no software do equipamento envolvido (tanto nos clientes quanto nos servidores);
- ❸ Falta de sincronismo no **acesso concorrente** entre as várias instâncias dos aplicativos acessando simultaneamente os mesmos dados.

O objetivo desta aula é mostrar os recursos que o modelo relacional e os SGBDs nele apoiados provêm ao Administrador da Base e aos analistas e programadores de aplicativos para que eles possam tomar ações para evitar os três tipos de problemas.

Gerenciamento de Transações

Erros de lógica


A corretude dos dados armazenados em um SGBD depende em última instância da completa corretude da lógica dos aplicativos. Portanto, é necessário que os aplicativos:

- Sejam codificados segundo as **boas práticas** do desenvolvimento de software;
- Atendam aos **Princípios do Modelo Relacional**:
 - Os SGBDs disponibilizam recursos que violam esses princípios, mas eles devem usados com cautela, de maneira consciente para suprir no código dos aplicativos a proteção às violações utilizadas;
- Trabalhar com a **base normalizada**;
 - Toda operação de atualização sobre relações não normalizadas têm que ter o controle explícito da normalização no aplicativo!
- Operar sempre em modo de **Transações consistentes**.

Gerenciamento de Transações

O Conceito de Transações

Transação

- 1 Uma transação é um conjunto de um ou mais comandos que, partindo de um estado consistente da base, sempre termina deixando a base consistente; e
 - 2 Uma transação sempre é executada completamente, ou então não executa nada.
- Se todo o acesso a uma base de dados é feito apenas por transações, partindo de uma base vazia (e portanto consistente), ela sempre estará consistente.
-  Portanto, se a programação dos aplicativos for feita usando transações consistentes, circunscreve-se o problema dos erros de lógica dos aplicativos a garantir que toda transação esteja isenta deles.

Os analistas e os programadores de aplicativos sempre devem confirmar que uma transação foi executada até o fim, pois o SGBD pode a qualquer instante forçar que ela falhe, cancelando tudo o que foi feito.

Gerenciamento de Transações

Conceitos

- Um evento que pode danificar o conteúdo de uma base de dados é a **Falta de Sincronismo** no acesso concorrente entre as várias instâncias dos aplicativos acessando simultaneamente os mesmos dados.
- Esse problema também é corrigido pelo conceito de **Transações**.

Vamos agora analisá-lo em mais detalhe.

Gerenciamento de Transações

Transação = Unidade lógica de trabalho

Conecta com o Banco

 Começa uma transação

 Operações de consulta e atualização

 . . .

 Finaliza transação

 Começa Transação

 Operações de consulta e atualização

 . . .

 Finaliza transação

 . . .

 Finaliza transação

Desconecta

As Propriedades **ACID**

- **A**tomicidade

- Todas as operações da transação são refletidas corretamente ou nenhuma delas é efetivada.

- **C**onsistência

- As transações preservam a consistência da base

Estado Inicial Consistente ➡ Estado Final Consistente

- **I**solamento

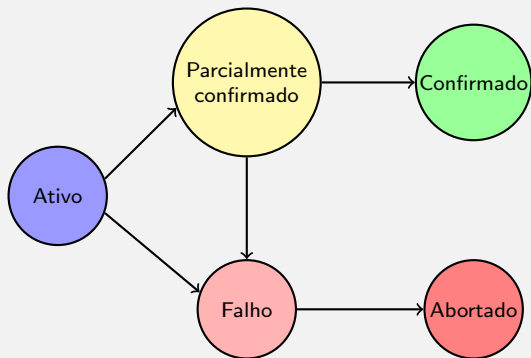
- Uma transação não vê o efeito de outra que está executando simultaneamente no sistema.

- **D**urabilidade

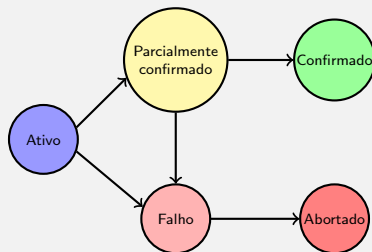
- Uma vez terminada, as alterações que a transação fez persistem na base de dados até que sejam explicitamente modificadas. O sistema de recuperação também é responsável por garantir a durabilidade.

Atomicidade e Durabilidade da Transação

- Uma transação pode ser confirmada ou ser revertida (abortada).
- Uma transação revertida precisa que as mudanças feitas por ela sejam desfeitas.
 - através de um log de operações.
- Uma transação precisa estar em um dos seguintes estados:



Atomicidade e Durabilidade da Transação



- **Ativo**: O estado inicial, enquanto estiver executando.
- **Parcialmente confirmado**: Depois que a instrução final for executada.
- **Falho**: Depois de descobrir que a execução normal não pode mais prosseguir.
- **Abortado**: Depois que a transação for revertida e o banco de dados foi restaurado ao seu estado anterior consistente.
- **Confirmado**: Após o término bem sucedido, as informações são persistentes.

Gerenciamento de Transações

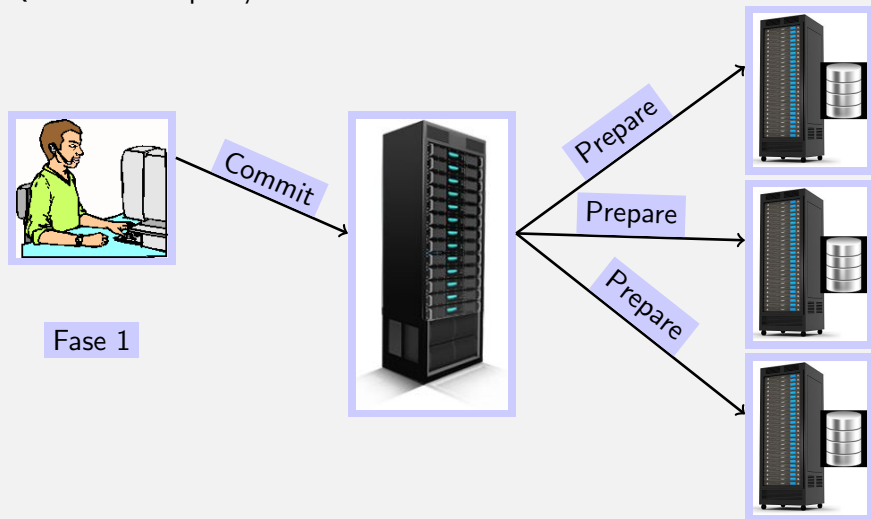
Two Phase Commit

- O controle das transações em um ambiente de concorrência é conseguido pelo protocolo conhecido como “*Two Phases Commit*”.
- Ele permite manter múltiplas partes, ou mesmo cópias dos dados em diferentes servidores, e garantir que todas as operações sejam realizadas por todos ou por nenhum.
- O protocolo de “*Two Phases Commit*” permite avaliar se todos os servidores estão ativos e consistentes para:
 - Finalizar com êxito uma operação,
 - ou permitir que algum informe inconsistência cancelando a operação em todos os servidores,
 - ou verificar que algum servidor está temporariamente sem conexão e aguardar um tempo ou cancelar a operação depois de uma espera muito longa.

Gerenciamento de Transações

Two Phase Commit

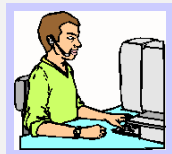
Quando uma operação é bem sucedida:



Gerenciamento de Transações

Two Phase Commit

Quando uma operação é bem sucedida:



Fase 2



OK

OK

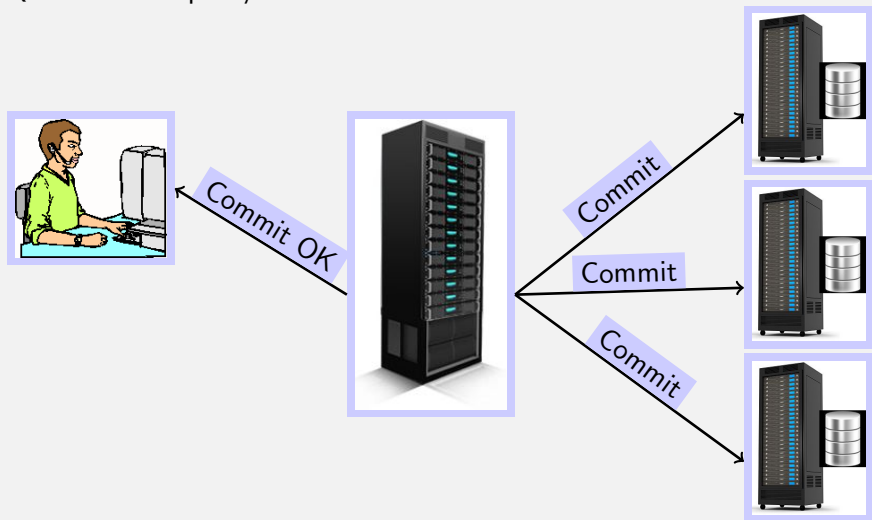
OK



Gerenciamento de Transações

Two Phase Commit

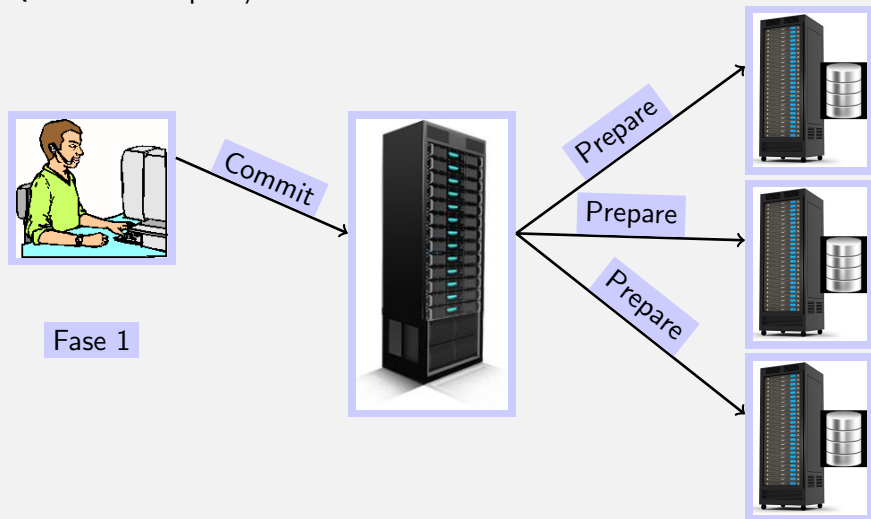
Quando uma operação é bem sucedida:



Gerenciamento de Transações

Two Phase Commit

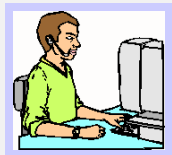
Quando uma operação **não** é bem sucedida:



Gerenciamento de Transações

Two Phase Commit

Quando uma operação **não** é bem sucedida:



Fase 2



OK

Erro

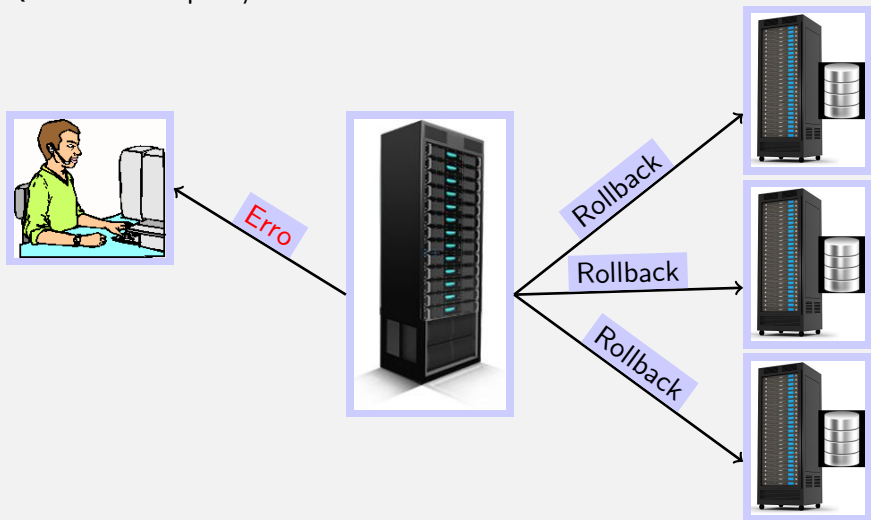
OK



Gerenciamento de Transações

Two Phase Commit

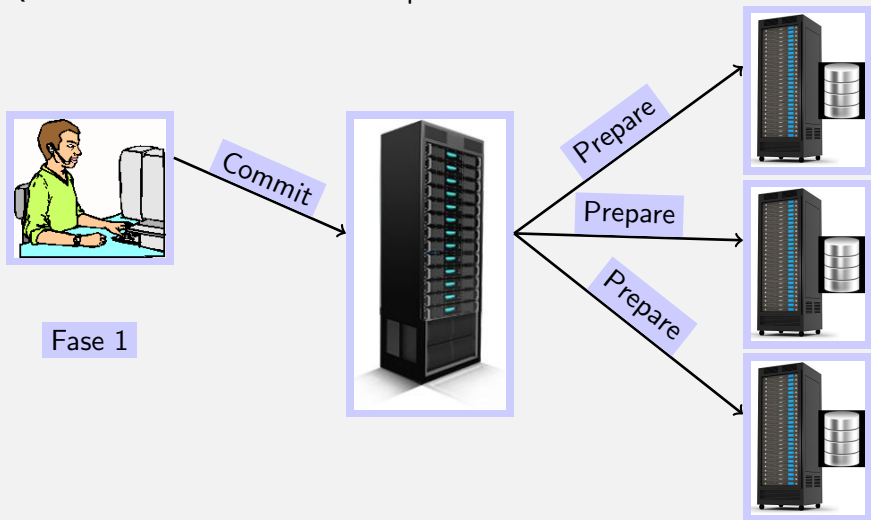
Quando uma operação **não** é bem sucedida:



Gerenciamento de Transações

Two Phase Commit

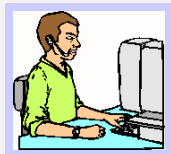
Quando um servidor não está respondendo



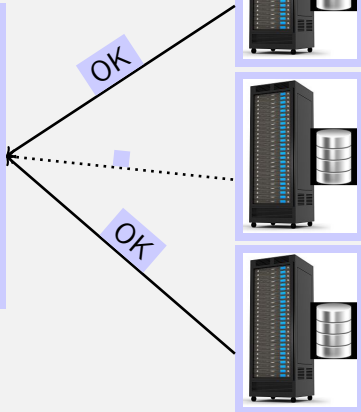
Gerenciamento de Transações

Two Phase Commit

Quando um servidor não está respondendo



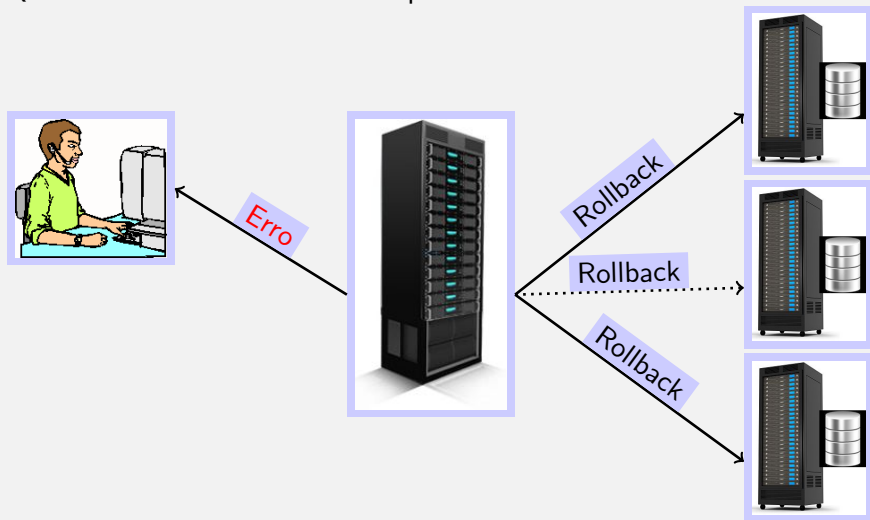
Fase 2



Gerenciamento de Transações

Two Phase Commit

Quando um servidor não está respondendo



Execução Serial X Intercalada

Uma transação é dita correta, pois partindo do estado de uma Base de Dados Correto e Consistente, a execução da transação deve terminar deixando a Base de Dados em um estado correto e consistente.

Um conjunto de transações corretas executadas em sequência deixa a base de dados em um estado correto e consistente:

Execução Serial

Cada transação é composta por uma sequência de comandos.

Um conjunto de transações pode ser executado intercalando comandos de diversas transações:

Execução Intercalada

Execução Serial X Intercalada

T1: 

T2: 

T3: 

Execução serial:



Execução intercalada:



Serializabilidade

Uma execução serial sempre leva de um estado correto e consistente para outro estado correto e consistente.

O estado final de uma base de dados, deixado por uma execução serial, pode ser diferente dependendo da ordem com que as transações são executadas, mas todos eles são estados corretos e consistentes.

Serializabilidade

O estado final de uma base de dados, deixado por uma execução intercalada, é consistente se ele for o mesmo resultado obtido pela execução de alguma execução serial.

Diz-se então que essa execução é **Serializável**.

Problemas com a Concorrência

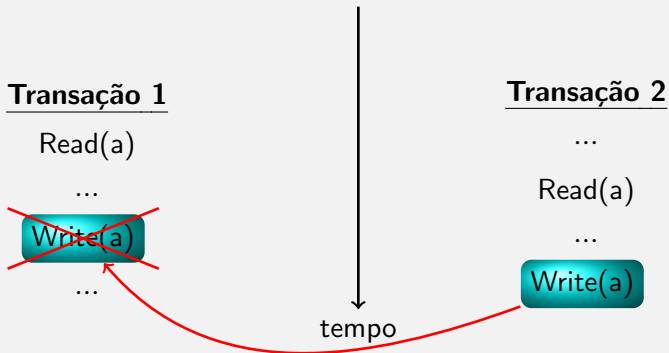
Dependendo da ordem com que os comandos de duas ou mais transações corretas são intercalados, o resultado final pode não ser correto, ou seja, não existe uma sequência de transações executadas em série que levaria a esse resultado.

Existem basicamente três problemas de concorrência originários de uma intercalação incorreta:

- ❶ Perda de Atualização
- ❷ Leitura Inválida
- ❸ Análise Inconsistente

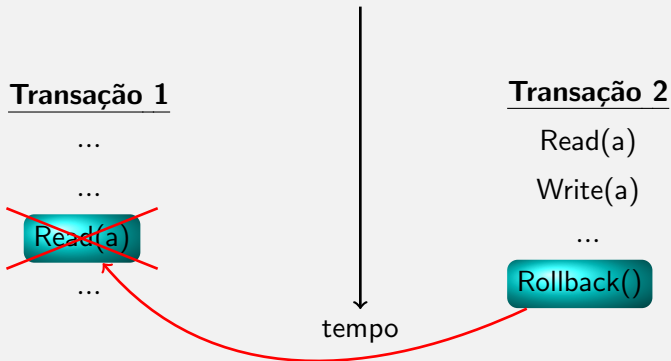
Problemas com a Concorrência

Perda de Atualização



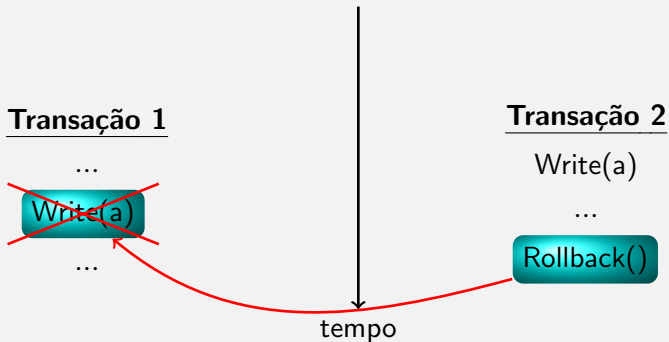
Problemas com a Concorrência

Leitura Inválida



Problemas com a Concorrência

Leitura Inválida



Problemas com a Concorrência

Análise Inconsistente

Transação 1

Read(a)  a ← 10

...

...

...

...


...

Read(b)  b ← 40

...  a ← 10, b ← 40

Este par de valores nunca existiu na base de dados!

Transação 2

...  a ← 10, b ← 50


Read(a)  a ← 10

Write(a)  a ← 20

Read(b)  b ← 50

Write(b)  b ← 40

Commit

...  a ← 20, b ← 40

...

tempo

Protocolos baseados em bloqueio

- Uma maneira de garantir o isolamento é exigir que seja feito um bloqueio para todo dado a ser acessado.
- Um dado com bloqueio não pode ser alterado, ao menos que o bloqueio seja removido por quem o bloqueou.
- Existem dois tipos de bloqueios (*Locks*)
 - 1 **Compartilhado**: Se uma transação T tiver obtido um bloqueio no modo compartilhado (indicado por **S**) sobre o item Q, então T pode ler mas não pode escrever Q.
 - 2 **Exclusivo**: Se uma transação T tiver obtido um bloqueio no modo exclusivo (indicado por **X**) sobre o item Q, então T pode ler e escrever Q.
- As regras de concessão de bloqueio são:

	S	X
S	verdadeiro	falso
X	falso	falso

Tabela: Compatibilidade de bloqueios

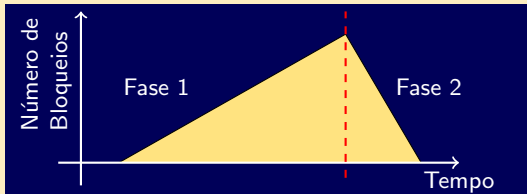
Bloqueio em duas fases

Pré-requisito:

- Antes de qualquer operação sobre qualquer objeto, deve ser requisitado um bloqueio sobre o objeto.

Protocolo de Bloqueio em Duas Fases


- **Fase 1.** Uma transação pode obter bloqueios, mas não pode liberar qualquer bloqueio.
- **Fase 2.** Uma transação pode liberar bloqueios, mas não pode obter novos bloqueios.



Isolamento entre transações

Com o bloqueio total sobre todos os objetos solicitados por uma transação, a interdependência fica muito grande e reduz muito a capacidade de concorrência da base.

A base não sabe o que uma transação pode precisar no futuro, mas quem escreveu a transação sabe. Portanto, ele poderia solicitar menos proteção para uma transação que ele sabe que não vai usar determinados recursos sobre os objetos que está utilizando.

Portanto, o usuário pode escolher um grau de **isolamento** para as suas transações antes delas iniciarem.  Propriedades ACID.

Existem muitos níveis de isolamento possíveis. A linguagem SQL define apenas **4** níveis.

Quanto mais alto o nível de isolamento, menor a interferência entre as transações, mas também menores as possibilidades de concorrência.

Grau de Isolamento

Violações previstas

Leitura Inválida

Realiza-se a leitura de um dado modificado por uma transação que ainda não terminou. Se a transação modificadora for cancelada, o dado não terá sido modificado e, portanto, a leitura foi inválida.

Leitura Não Repetível

Depois de lido um valor, a transação o lê novamente. Se nesse ínterim outra transação houver modificado o dado, a leitura do mesmo dado terá visto dois valores diferentes.

Leitura Fantasma

Suponha-se que uma transação leia todas as tuplas de uma relação que atenda a uma condição de consulta. Então uma outra transação insere (ou atualiza) uma tupla que atenderia a essa condição. Qualquer atualização da primeira transação na base que afete todas as tuplas que atendam a essa condição vai afetar uma tupla que anteriormente não existia.

Grau de Isolamento

Segundo o Padrão da Linguagem SQL

SQL prevê os seguintes níveis de isolamento para uma transação:

Grau de Isolamento	Violações Admitidas		
	Leitura Inválida	Leitura Não Repetível	Leitura Fantasma
<i>Read Uncommitted</i>	Sim	Sim	Sim
<i>Read Committed</i>	Não	Sim	Sim
<i>Repeatable Read</i>	Não	Não	Sim
<i>Serializable</i>	Não	Não	Não

Grau de Isolamento

Padronização para **Oracle**.

Oracle prevê os seguintes níveis de isolamento para uma transação:

Grau de Isolamento	Violações Admitidas		
	Leitura Inválida	Leitura Não Repetível	Leitura Fantasma
<i>Read Uncommitted</i>	Sim	Sim	Sim
<i>Read Committed</i>	Não	Sim	Sim
<i>Repeatable Read</i>	Não	Não	Sim
<i>Serializable</i>	Não	Não	Não

Grau de Isolamento

Padronização para **Postgres**.

Postgres prevê os seguintes níveis de isolamento para uma transação:

Grau de Isolamento	Violações Admitidas		
	Leitura Inválida	Leitura Não Repetível	Leitura Fantasma
<i>Read Uncommitted</i>	Sim	Sim	Sim
<i>Read Committed</i>	Não	Sim	Sim
<i>Repeatable Read</i>	Não	Não	Não
<i>Serializable</i>	Não	Não	Não

- Read Committed é o grau de isolamento **default** para uma transação em PostgreSQL

Grau de Isolamento - PostgreSQL

Todos os seguintes graus de isolamento não permitem **escritas sujas**, ou seja, escritas em um item de dados que já foi escrito por outra transação mas não foi confirmada ou abortada.

- **READ COMMITTED (Leitura confirmada)**

- Este é o grau padrão.
- Permite que dados confirmados sejam lidos durante a transação T1. Mas, se o dado for alterado e confirmado por outra transação T2 durante a transação T1, novos valores serão atribuídos e predicados refletirão essas mudanças (leitura fantasma).

- **READ UNCOMMITTED (Leitura não confirmada)**

- Não existe implementado no PostgreSQL. Pode ser solicitado, mas se comporta como se fosse READ COMMITTED. Teoricamente permite leituras em dados alterados por transações não efetivadas.

- **REPEATABLE READ (Leitura repetitiva)**

- Permite que apenas dados confirmados sejam lidos numa transação T1, e ainda garante que entre duas leituras consecutivas de um dado item, os dados são os mesmos. Se nesse ínterim alguma transação T2 alterar os dados acessados, T1 aguarda, e se T2 abortar, T1 acessa o dado, mas se T2 confirmar, T1 aborta e reinicia.

- **SERIALIZABLE (Serializável)**

- Cada operação da transação acessa apenas os dados modificados pela própria transação e os dados já efetivados pelas demais transações antes do início da transação. Além disso, garante que o resultado é o mesmo que o serial das transações. Requer mais processamento na verificação constante de todos predicados acessados durante a transação, abortando e reiniciando sempre e qualquer mudança.

Isolamento de Instantâneo - Snapshot

- Tipo específico de esquema de controle de concorrência utilizado pelos grandes banco de dados Oracle, SQL Server e PostgreSQL, para deixar uma transação (quase) serializável.
- Consiste em dar a uma transação uma “fotografia” do banco de dados no momento que a transação inicia.
- Todas operações ocorrem nessa fotografia durante a transação, completamente isolado de outras transações.
- Ao completar, apenas as informações que foram atualizadas são substituídas na base de dados
 - Note, apenas o ponteiro para a nova página atualizada é atualizado no commit.

Isolamento de Instantâneo - Snapshot

- Um pequeno detalhe, é que esta abordagem não garante 100% a serializabilidade.
- A validação de transações de atualizações requer cuidados.
 - Quando duas transações em paralelo querem atualizar o mesmo item de dados, a seguintes políticas podem ser usadas:
 - **primeiro confirmador vence**
 - **primeiro atualizador vence**
 - Toda transação perdedora é reiniciada
 - Número limitado de vezes, senão ela vence, para evitar starvation.

Isolamento de Instantâneo - Snapshot

- Um desenvolvedor pode se proteger para garantir que sua transação não seja reiniciada muitas vezes.
- Para isso, ele pode bloquear as tuplas quando sabe-se previamente quais são as atualizações.
- Exemplo:

```
SELECT * from aluno  
WHERE RA = '1234'  
FOR UPDATE;
```
- A cláusula **for update** coloca um bloqueio exclusivo nos dados coerentes com o predicado, sendo “marcados” para atualização na sequência da transação.

Controle de Concorrência - JDBC

- A mudança do nível de isolamento precisa ser feita como a primeira instrução de uma transação.
- Funções da API do JDBC como o método `Connection.setTransactionIsolation(int level)` definem o nível de isolamento das transações.
- Além disso, a confirmação automática de instruções individuais precisa ser desativada, se estiver ativada como padrão.
- O método `Connection.setAutoCommit(false)` é usado para desativar a confirmação automática.

Controle de Concorrência

Tipos de bloqueio

- Quando uma transação solicita bloqueio de um recurso, verifica-se se ele não conflita com outro emitido anteriormente por outra transação.
- A execução somente é liberada se não existe conflito. Se existir, a transação para esperando o recurso ser liberado.
- Um bloqueio compartilhado não conflita com outros bloqueios compartilhados;
- Um bloqueio exclusivo é atendido quando nenhuma outra transação esteja usando o recurso, nem para escrita nem para leitura.
- Para atender ao protocolo de bloqueio em duas fases, a transação vai acumulando os bloqueios obtidos, e só os libera, todos de uma vez, quando a transação termina.

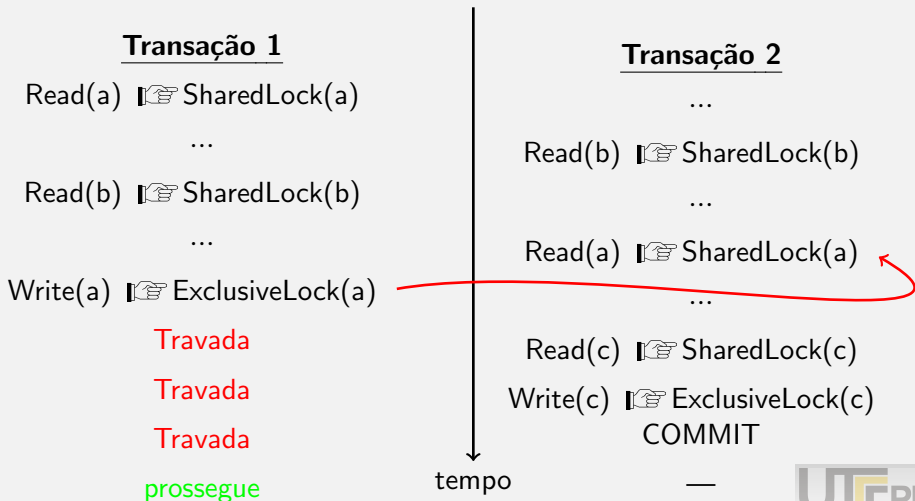
Controle de Concorrência

O problema do bloqueio perpétuo

- O protocolo de bloqueio em duas fases garante que duas transações nunca interferem uma na outra,
- mas criam um problema novo: o **bloqueio perpétuo** (*deadlock*).

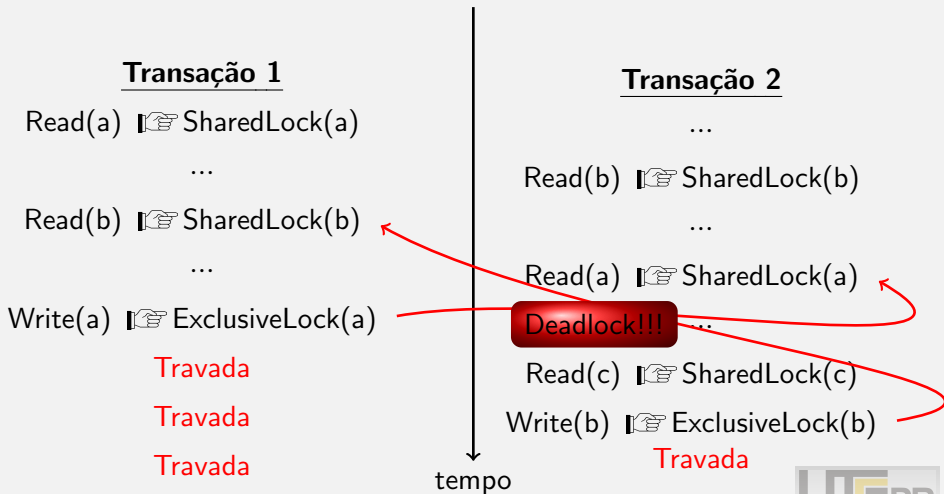
Controle de Concorrência

O problema do bloqueio perpétuo



Controle de Concorrência

O problema do bloqueio perpétuo



Controle de Concorrência

O problema do bloqueio perpétuo

- Para se proteger do *deadlock*, o servidor do SGBD mantém um **Dígrafo de Dependências** entre as transações.
- Cada transação é representada como um nó no dígrafo, e sempre que um nó para esperando por outro (ou outros), cria-se uma aresta dirigida do nó que pede para o nó que detém o bloqueio.
- ★ (A solicitação de um bloqueio exclusivo pode produzir várias arestas se aquele dado já tem bloqueio compartilhado em várias transações.)
- Sempre que alguma aresta é criada, o SGBD verifica se o dígrafo passa a ter ciclos:
- Se surgiu um ciclo, ocorreu um *deadlock*!

Controle de Concorrência

O problema do bloqueio perpétuo

- É impossível remover uma aresta 🖱️ não se pode dizer para uma transação que ela não depende mais de um dado!
- A única ação possível é remover um nó 🖱️ Aborta-se a transação correspondente, à revelia do aplicativo.
- Existem diversas heurísticas para se escolher a “vítima”. A mais comum é escolher o nó que tem o maior número de arestas incidentes.
- Essa heurística pode potencialmente liberar o maior número de transações para prosseguir. Mas pode coincidir dessa transação ser também a que está rodando a mais tempo...
- Antes de avisar o aplicativo que a transação foi abortada, o SBGD pode tentar recuperá-la, aguardando um tempo aleatório e recomeçando a transação a partir do *Log*.
- Se depois de algumas tentativas a transação sempre entrar em *deadlock*, ela é abortada e o aplicativo é avisado.

Comando SET TRANSACTION

- O comando `SET TRANSACTION` é usado para indicar que a próxima transação a ser iniciada será *read-only* ou *read/write*.
- Uma transação inicia se não houver transação ativa e for emitido um comando que gera uma trava de escrita:
 - Um comando da DDL;
 - Um comando `INSERT | UPDATE | DELETE`;
 - Um comando `SELECT ... FOR UPDATE`;
 - ou for emitido um comando `SET TRANSACTION`.
- A transação é finalizada com um comando `COMMIT` ou `ROLLBACK`.

Comando SET TRANSACTION

Define os parâmetros da próxima transação – Padrão ISO.

SET TRANSACTION – Postgres

```
SET TRANSACTION
{{READ {ONLY | WRITE} |
  ISOLATION LEVEL {SERIALIZABLE | REPEATABLE READ |
    READ COMMITTED | READ UNCOMMITTED}}
```

Exemplo:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;
```

- Transações **READ ONLY** podem executar apenas comandos de leitura.

Comando SET TRANSACTION

Define os parâmetros da próxima transação – **Oracle**

Comando SET TRANSACTION

```
SET TRANSACTION  
  {READ {ONLY | WRITE} |  
  ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}  
  [USE ROLLBACK SEGMENT <Nome_segmento>]  
  [Name <Nome_transacao>];
```

- Transações **READ ONLY** podem executar apenas comandos de leitura.

Comando SET TRANSACTION

Define os parâmetros da próxima transação – **Oracle**

- Outro comando útil para reservar recursos é:

Comando LOCK TABLE

```
LOCK TABLE <Nome_Tab/View>  
  IN {ROW SHARE | ROW EXCLUSIVE | SHARE |  
      SHARE ROW EXCLUSIVE | EXCLUSIVE} MODE  
  [NOWAIT];
```

- **Row Share.** Permite acesso concorrente na tabela mas proíbe usuários de bloquear a tabela toda para acesso exclusivo.
- **Row Exclusive.** O mesmo que Row Share, mas proíbe bloqueios compartilhados.
- **Share.** Permite consultas na tabela mas proíbe updates.
- **Share Row Exclusive.** Proíbe os dois tipos de bloqueios.
- **Exclusive.** Permite consultas na tabela mas proíbe qualquer outra atividade nela.

Comando SET TRANSACTION

Resumo: transações em **Oracle**

- **READ COMMITTED** (padrão):

ESCRITA : Para executar um comando, a transação aguarda até que todos os recursos necessários sejam liberados e prossegue;

LEITURA : A transação “vê” apenas os dados consolidados (*committed*) pelas demais transações antes do início de cada operação.

- **SERIALIZABLE**:

ESCRITA : Caso outra transação altere uma tupla acessada depois do início desta transação *serializable*, se esta transação tentar alterar a tupla, ela receberá a exceção:

ORA-08177: Can't serialize access for this transaction.

LEITURA : A transação “vê” apenas os dados tal como existiam no início da transação e aqueles modificados por ela própria.

Comando SAVEPOINT

Define um ponto de salvamento para aq transação corrente.

SQL

```
SAVEPOINT <savepoint_name>;
```

- Cria um *savepoint* com o nome indicado;
- O objetivo do *savepoint* é marcar aquele estado para poder voltar a ele com uma operação de *rollback* se necessário.
- Pode haver muitos *savepoints* mas todos estão dentro de uma transação.

Comando COMMIT

Utilizado para indicar término bem sucedido de uma transação.

SQL

```
COMMIT [WORK | TRANSACTION];
```

- O comando **COMMIT** consolida todas as alterações realizadas na base de dados desde o último **COMMIT** ou **ROLLBACK**, ou desde a conexão inicial do usuário, se não houveram transações anteriores;
- Esse comando aplica-se a todos os comandos da **SQL**, incluindo comandos de definição de dados e de controle;
- Libera todos os recursos bloqueados;
- Torna permanente as ações da transação;
- Termina a transação.
- As palavras **WORK** e **TRANSACTION** são opcionais e não afetam a execução. (Apenas a opção **WORK** existe no padrão SQL).
- Todos os *savepoints* da transação desaparecem.

Comando COMMIT

Utilizado para indicar término bem sucedido de uma transação.

- Conceitualmente, um SGBD trata comandos da DDL da seguinte maneira:

```
BEGIN
    COMMIT;
do the ddl command;
    COMMIT;
EXCEPTION
    WHEN error THEN
        ROLLBACK;
        raise exception;
END;
```

- Esse padrão pode ser modificado por um comando `SET AUTOCOMMIT OFF`.

Comando ROLLBACK

Finaliza uma transação corrente restaurando a base de dados de seu estado anterior para executar a próxima transação.

SQL/Postgres

```
ROLLBACK [WORK | TRANSACTION];
```

ou

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] <savepoint_name>;
```

- Tipos de ROLLBACK:

- ROLLBACK de comando SQL;
- ROLLBACK para um *savepoint*;
- ROLLBACK de transação por solicitação do usuário;
- ROLLBACK de transação por finalização sem sucesso de processo;
- ROLLBACK de todas as transações por falha do sistema;
- ROLLBACK de transações incompletas em fases de recuperação do sistema.

Comando ROLLBACK

Finaliza uma transação corrente restaurando a base de dados de seu estado anterior para executar a próxima transação.

Exemplo 1

```
ROLLBACK;
```

- Descarta todas as alterações efetuadas na transação corrente e volta a base para o estado em que se encontrava antes do início dessa transação;
- Libera todos os recursos bloqueados;
- Termina a transação.

Comando ROLLBACK TO SAVEPOINT

Restaura o estado da base para o estado anterior à criação do `SAVEPOINT` indicado.

Exemplo 2

```
ROLLBACK TO SAVEPOINT <nome>;
```

- Desfaz as operações realizadas depois do *savepoint*;
- Os *savepoints* criados depois do ponto de *rollback* indicado são removidos;
- Libera os recursos bloqueados depois do *savepoint*;
- A transação permanece ativa, mas ainda não efetivada.

Comando ROLLBACK

Finaliza uma transação corrente restaurando a base de dados de seu estado anterior para executar a próxima transação.

Exemplo 3

```
INSERT INTO tabela1 VALUES (1);  
SAVEPOINT inicio;  
INSERT INTO tabela1 VALUES (2);  
ROLLBACK TO inicio;  
INSERT INTO tabela1 VALUES (3);  
COMMIT;
```

Nesse exemplo, o comando **INSERT** é confirmado para inserir 1 e 3, mas o insert com valor 2 foi cancelado e não foi inserido este valor.

Comando RELEASE SAVEPOINT

Remove um *savepoint* sem desfazer as operações feitas depois dele

RELEASE SAVEPOINT

```
RELEASE [SAVEPOINT] <savepoint_name>;
```

- Remove o *savepoint*;
- e remove todos os demais *savepoints* criados depois daquele especificado;
- Libera os recursos bloqueados depois do *savepoint*;
- As operações realizadas depois do *savepoint* indicado não são afetadas;
- Se diversos *savepoints* têm o mesmo nome, apenas o mais recente é removido;
- A transação permanece ativa, mas ainda não é efetivada.

Roteiro

- 1 Controle de Transação
- 2 Controle de Concorrência
- 3 Sintaxe em SQL

SGBDs Relacionais

– Controle de Transação e Concorrência

Prof. Dr. Ives Renê V. Pola

ivesr@utfpr.edu.br

Departamento Acadêmico de Informática – DAINF

UTFPR – Pato Branco DAINF

UTFPR

Pato Branco - PR

FIM

