

Megadados

Aula 14 – ACID

2019 – Engenharia

Fábio Ayres <fabioja@insper.edu.br>

Aula de hoje

Capítulo 13

- Transações

Mais sobre transações

- Existe um conjunto de **propriedades das transações** que tem o propósito de **garantir a validade dos dados armazenados**, mesmo que ocorram
 - falhas de sistema,
 - acessos concorrentes,
 - erros de acesso,
 - etc.

Estas propriedades são indicadas pela sigla **ACID**

ACID

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Atomicidade

Na aula passada vimos como efetuar **transações**, onde

- um conjunto de operações SQL (uma transação)...
- deve ser **concretizada** integralmente através do comando **COMMIT** ...
- ou **cancelada** inteiramente com o comando **ROLLBACK**.

Esta propriedade, de garantir que um conjunto de comandos é executado ou rejeitado como uma única unidade é chamada **atomicidade**.

Exercício

Descreva uma situação onde várias operações de banco de dados devem ser executadas atomicamente.

Consistência

É a propriedade que indica que o banco de dados passa de um estado válido para outro estado válido a cada transação.

Consistência

A consistência é estabelecida através de *constraints* e *triggers*. Por exemplo:

- *Constraints* de *foreign key* garantem que um campo de uma tabela (a chave estrangeira) aponta para uma linha válida, e única, de outra tabela.
- Os triggers 'ON DELETE' e 'ON UPDATE' indicam o que fazer quando essas operações ocorrem, para que a base se mantenha consistente.

comidas

id	PK
nome	
id-perigo	

perigo

id	PK
nome	

Comidas

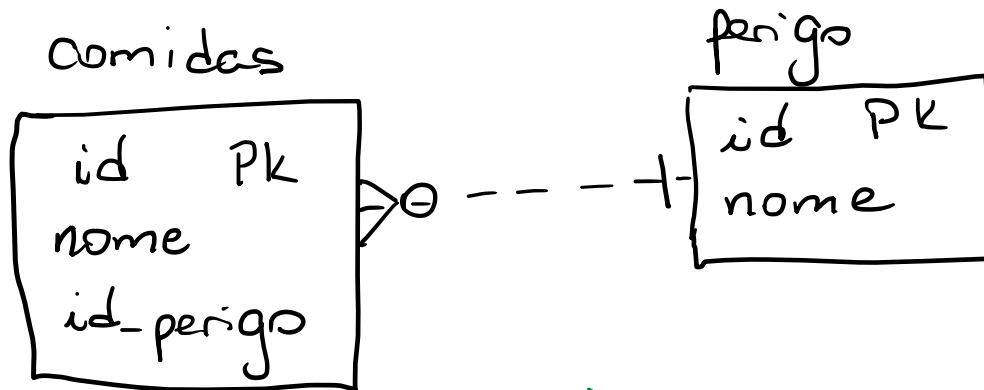
id	nome	id-perigo
1	Alface	1
2	Coxinha	1
3	peixe	3
4	macarrão	2

???

inconsistencia

perigo

id	nome
1	intestinal
2	dermatologicos
3	mental



Fk (id-perigo) REF perigo(id)

Comidas		
id	nome	id-perigo
1	Alface	1
2	Coxinha	1
3	peixe	3
4	macarrão	2

perigo	
id	nome
1	intestinal
2	dermatológico
3	mental

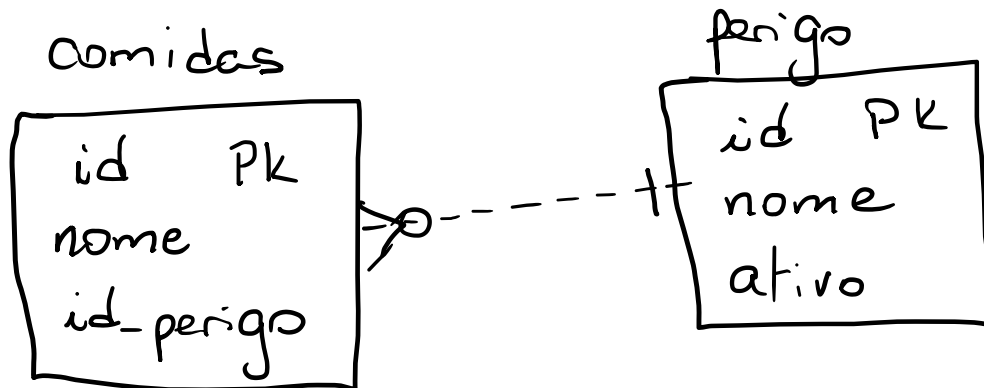
O que fazer?
ON DELETE

{
RESTRICT
CASCADE
SET NULL

MANTEM CONSISTENCIA!

Exercício

ON DELETE pode ajudar a manter consistência em *deletes* físicos. Como você poderia garantir a consistência do banco de dados na presença de *deletes* lógicos?



Fk (id_perigo) REF perigo(id)

Comidas		
id	nome	id-perigo
1	Alface	1
2	Coxinha	1
3	peixe	3 NULL
4	macarrão	2

perigo		
id	nome	ativo
1	intestinal	T
2	dermatologicos	T
3	mental	T F

ON UPDATE → chamar uma stored procedure

TRIGGERS

Isolamento

É a propriedade relativa à capacidade de executar várias transações concorrentes sem que uma transação interfira diretamente na outra, ou seja, o usuário tem a percepção de que as transações foram executadas sequencialmente.

Exemplo

Suponha que temos dois usuários efetuando as seguintes transações:

Transação 1:

```
START TRANSACTION
SELECT saldo INTO @s FROM contas WHERE id = 1
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2
COMMIT
```

Transação 2: o mesmo

```
START TRANSACTION
SELECT saldo INTO @s FROM contas WHERE id = 1
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2
COMMIT
```

Vamos pensar um pouco

Se o saldo inicial da conta 1 era 1000 e o saldo inicial da conta 2 era 500, qual será o saldo final de cada conta?

Considere os seguintes cenários:

1. A transação 1 ocorre inteiramente antes da transação 2
2. As operações das duas transações ocorrem alternadamente

Transação 1:

```
START TRANSACTION ①  
SELECT saldo INTO @s FROM contas WHERE id = 1 ②  
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1 ③  
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2 ④  
COMMIT ⑤
```

Transação 2: o mesmo

```
START TRANSACTION ⑥  
SELECT saldo INTO @s FROM contas WHERE id = 1 ⑦  
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1 ⑧  
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2 ⑨  
COMMIT ⑩
```

	saldo 1	saldo 2	@s(1)	@s(2)
	1000	500		
	1000	500		
①	1000	500	1000	
②	1000	500	1000	
③	500	500	1000	
④	500	1000	1000	
⑤	500	1000	1000	
⑥	500	1000		500
⑦	500	1000		500
⑧	250	1000		500
⑨	250	1250		500
⑩	250	1250		500

Transação 1:

```
START TRANSACTION (1)
SELECT saldo INTO @s FROM contas WHERE id = 1 (3)
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1 (5)
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2 (7)
COMMIT (9)
```

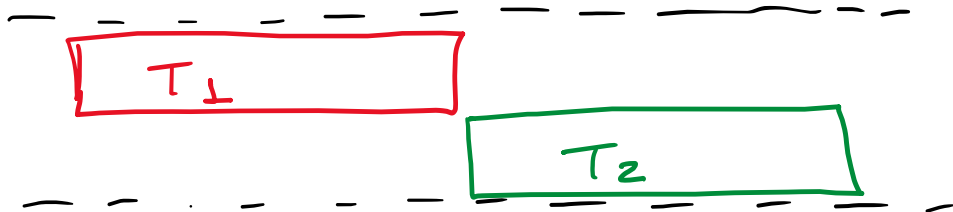
Transação 2: o mesmo

```
START TRANSACTION (2)
SELECT saldo INTO @s FROM contas WHERE id = 1 (4)
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1 (6)
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2 (8)
COMMIT (10)
```

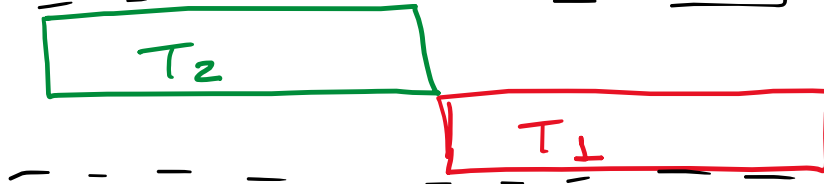
	Saldo 1	Saldo 2	@s(1)	@s(2)
(1)	1000	500	—	—
(2)	1000	500	—	—
(3)	1000	500	1000	—
(4)	1000	500	1000	1000
(5)	500	500	1000	1000
(6)	0	500	1000	1000
(7)	0	1000	1000	1000
(8)	0	1500	1000	1000
(9)	0	1500	—	1000
(10)	0	1500	—	1000

Transações serializáveis

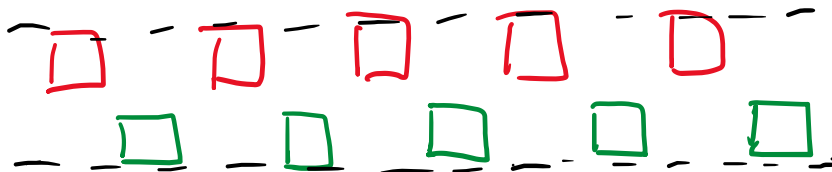
Ou isso



Ou isso



Mas não isso



Como resolver essa situação?

Parece que o problema é o acesso concorrente às linhas 1 e 2 da tabela *contas*. Você se lembra de algum mecanismo para resolver o problema de acesso concorrente à recursos compartilhados?

Implemente sua solução em pseudo-SQL (tipo, invente uns comandos que façam sentido!)

Solução

Transação 1:

START TRANSACTION

LOCK contas

SELECT saldo INTO @s FROM contas WHERE id = 1

UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1

UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2

UNLOCK contas

COMMIT

Transação 2: o mesmo

START TRANSACTION

LOCK contas

SELECT saldo INTO @s FROM contas WHERE id = 1

UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1

UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2

UNLOCK contas

COMMIT

DEADLOCK!!!



Níveis de isolamento

O padrão ANSI/ISO SQL define 4 níveis de isolamento de transações:

- SERIALIZABLE
- REPEATABLE READ
- READ COMMITTED
- READ UNCOMMITTED

SERIALIZABLE

É o nível mais restritivo de isolamento. Uma transação que tente atualizar dados não-gravados de outra transação será rejeitada.

Exercício: O que acontece se todas as transações forem SERIALIZABLE? Liste uma vantagem e um problema.

Este nível é o padrão sugerido pelo comitê ANSI/ISO, mas não é sempre adotado por causa do desempenho. MySQL **NÃO** adota este nível por default.

REPEATABLE READ

Neste nível,

- se você leu algumas linhas
- e tenta ler elas de novo,
- os mesmos valores serão retornados.

Porém **novas linhas** podem ter sido adicionadas e serão retornadas também na segunda leitura! Esse fenômeno é chamado de *leitura fantasma* (**phantom read**).

Este é o nível *default* de isolamento no engine default do MySQL (InnoDB).


Exemplo REPEATABLE READ

Transação 1

```
SET TRANSACTION ISOLATION LEVEL  
REPEATABLE READ;
```

```
SELECT * FROM contas WHERE id = 55;
```


```
SELECT * FROM contas WHERE id = 55;
```



Uma vez lida, uma linha não será alterada por outras transações até que esta transação aqui termine!

Transação 2

```
UPDATE contas SET saldo = 200 WHERE  
id = 55;  
COMMIT;
```



A transação 2 será bloqueada até que a transação 1 termine!

Exemplo REPEATABLE READ

Transação 1

```
SET TRANSACTION ISOLATION LEVEL  
REPEATABLE READ;
```

```
SELECT * FROM contas WHERE saldo <  
1000;
```

```
SELECT * FROM contas WHERE saldo <  
1000;
```

↑

As linhas da primeira leitura
ainda estão aqui, sem
modificações! Porém...

Transação 2

```
INSERT INTO contas (saldo) VALUES  
(200);  
COMMIT;
```

↑

... as novas linhas também aparecerão no
segundo SELECT! Estas são chamadas de
linhas *fantasmas*.

Solução com REPEATABLE READ

Transação 1:

```
START TRANSACTION
```

```
LOCK contas WHERE id = 1
```

```
SELECT saldo INTO @s FROM contas WHERE id = 1
```

```
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1
```

```
LOCK contas WHERE id = 2
```

```
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2
```

```
UNLOCK contas WHERE id = 2
```

```
UNLOCK contas WHERE id = 1
```

```
COMMIT
```

Transação 2: o mesmo

```
START TRANSACTION
```

```
LOCK contas WHERE id = 1
```

```
SELECT saldo INTO @s FROM contas WHERE id = 1
```

```
UPDATE contas SET saldo = saldo - @s/2 WHERE id = 1
```

```
LOCK contas WHERE id = 2
```

```
UPDATE contas SET saldo = saldo + @s/2 WHERE id = 2
```

```
UNLOCK contas WHERE id = 2
```

```
UNLOCK contas WHERE id = 1
```

```
COMMIT
```

READ COMMITTED

Neste nível de isolamento:

- se uma transação ocorrer em paralelo com a nossa transação ...
- e fizer o COMMIT de seus dados, ...
- então nossa transação pode acabar lendo dados de linha modificados.

Este fenômeno é chamado de “**non-repeatable read**”.

Ficará mais claro com um exemplo:

Exemplo READ COMMITTED

Transação 1

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

```
SELECT * FROM contas WHERE id = 55;
```

```
SELECT * FROM contas WHERE id = 55;
```

```
SELECT * FROM contas WHERE id = 55;
```

O terceiro SELECT vai ler um valor diferente do primeiro!

Transação 2

```
UPDATE contas SET saldo = 200 WHERE  
id = 55;
```

```
COMMIT;
```

A transação 2 fez o COMMIT de uma mudança!

READ UNCOMMITTED

Neste nível de isolamento um dado modificado de uma transação não-finalizada será acessível pela nossa transação (**dirty read**). É o nível mais perigoso, e raramente utilizado.

Exemplo READ UNCOMMITTED

Transação 1

```
SET TRANSACTION ISOLATION LEVEL  
READ UNCOMMITTED;
```

```
SELECT * FROM contas WHERE id = 55;
```

```
SELECT * FROM contas WHERE id = 55;
```

↑
O segundo SELECT vai ler um
valor diferente do primeiro!

Transação 2

```
UPDATE contas SET saldo = 200 WHERE  
id = 55;
```

```
ROLLBACK;
```

Ops, ninguém deveria ter lido o valor
alterado! A transação 1 leu lixo!

↖
A transação 2 ainda
não fez o COMMIT
de uma mudança!

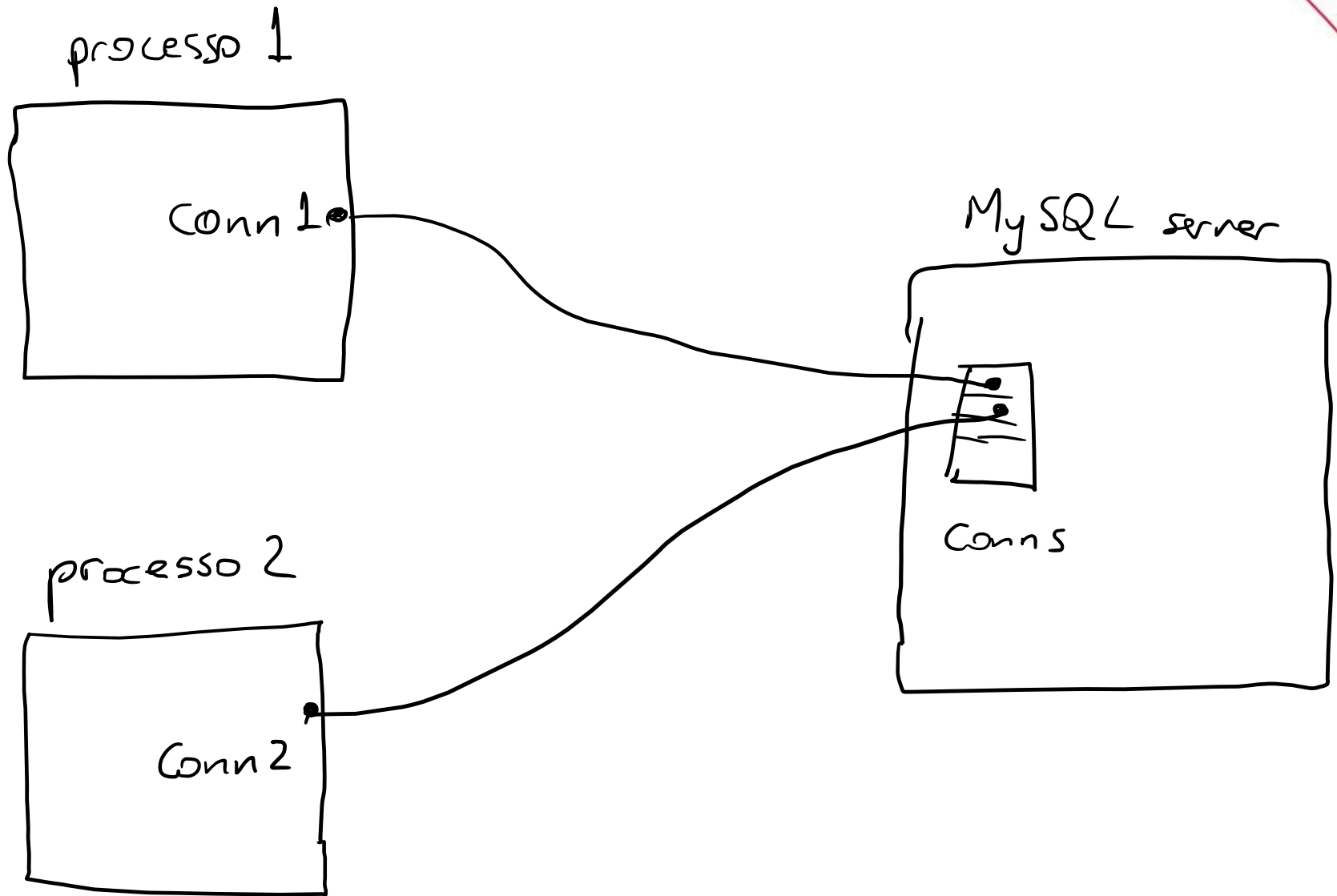
Comparação

Nível	Dirty reads?	Non-repeatable reads?	Phantom reads?
SERIALIZABLE	Não	Não	Não
REPEATABLE READ	Não	Não	Pode ocorrer
READ COMMITTED	Não	Pode ocorrer	Pode ocorrer
READ UNCOMMITTED	Pode ocorrer	Pode ocorrer	Pode ocorrer

Durabilidade

É a propriedade que diz que quando uma transação é confirmada (COMMIT), ela permanecerá gravada mesmo que a energia acabe ou o sistema trave.

Ou seja
COMMIT => vai p/ storage



Próxima aula

Capítulo 15

- Stored procedures
- Constraints e triggers

Insper

www.insper.edu.br