



# TRANSACTIONS TRANSAÇÕES



Bruno A. N. Travençolo – FACOM

# Problemas de atomicidade

---

- ▶ Os sistemas estão sujeitos as falhas
- ▶ As aplicações devem assegurar após a detecção de uma falha os dados sejam salvos em seu último estado consistente, anterior a ela.
  - ▶ Ex: Transferir R\$ 50,00 da conta A para a conta B
    - ▶ É possível que seja feito o débito em A e que o crédito em B não se realize por causa de uma falha, criando assim um estado inconsistente
- ▶ As operações devem ser atômicas – deve ocorrer por completo ou não ocorrer
- ▶ Difícil garantir essa propriedade em um sistema convencional de processamento de arquivos



# Acesso concorrente

---

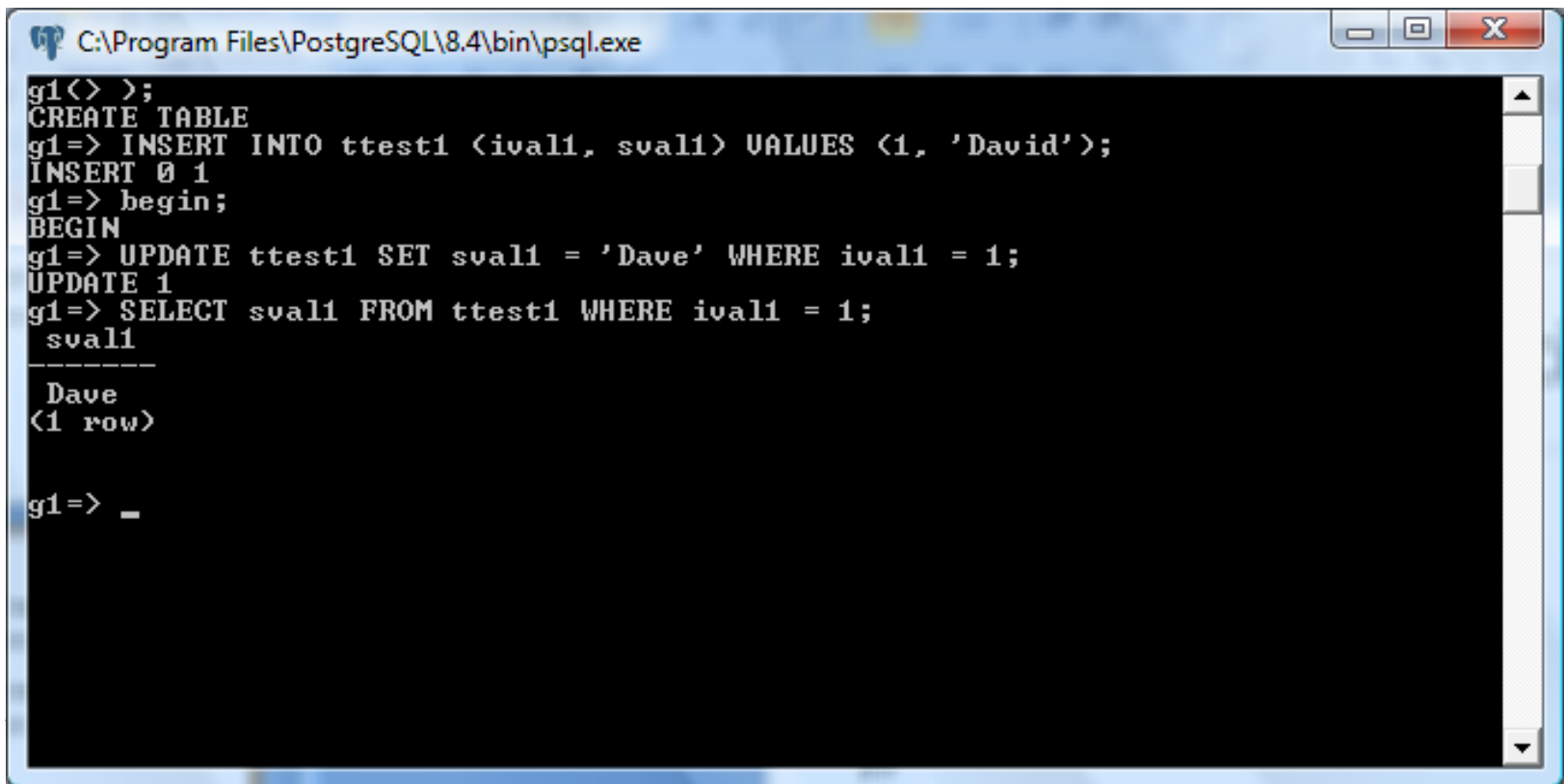
- ▶ Vários sistemas permitem a manipulação simultânea (concorrente) aos dados
- ▶ Interação entre atualizações concorrentes pode resultar em inconsistência dos dados
  - ▶ Ex: Saldo de uma conta: R\$ 500,00
  - ▶ Dois cliente retiram, ao mesmo tempo, 50 e 100 reais.
  - ▶ O sistema lê, nos dois casos, que o saldo é R\$ 500,00
  - ▶ Após as retiradas, o saldo pode fica em R\$ 450,00 ou R\$ 400,00 ao invés de R\$ 350,00
- ▶ O sistema deve supervisionar esse tipo de operação – o que é difícil caso diferentes programas acessem o mesmo dado



# Abrir o PSQL console

---

- ▶ No pgadminIII tem o menu plugins/PSQL console



```
C:\Program Files\PostgreSQL\8.4\bin\psql.exe
g1<> >;
CREATE TABLE
g1=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
INSERT 0 1
g1=> begin;
BEGIN
g1=> UPDATE ttest1 SET sval1 = 'Dave' WHERE ival1 = 1;
UPDATE 1
g1=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;
 sval1
-----
 Dave
(1 row)

g1=> _
```

# Propriedades ACID

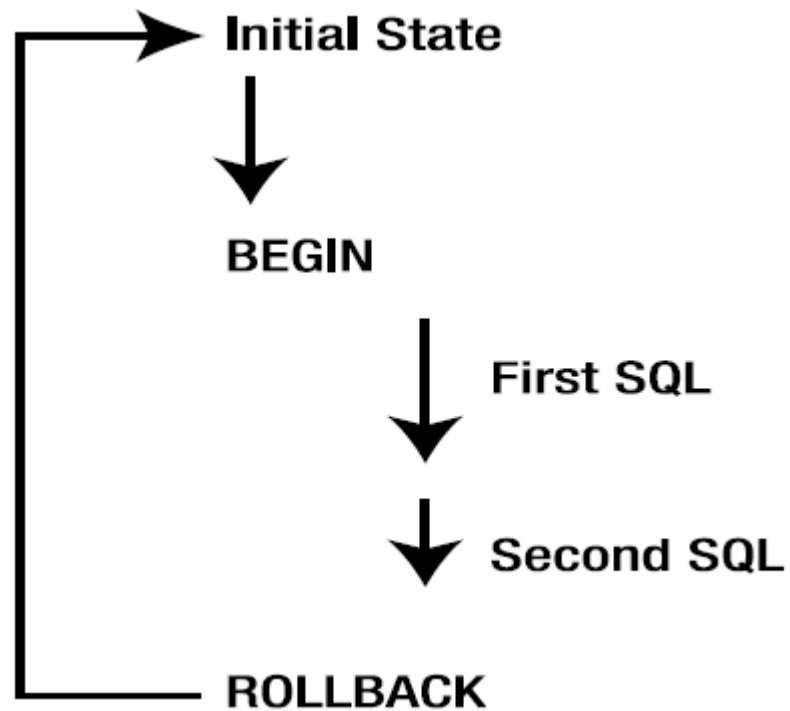
---

- ▶ **Atomicidade:** todas estas operações devem ser efetivadas ou, na ocorrência de uma falha, nada deve ser efetivado: “tudo ou nada”
- ▶ **Consistência:** as transações preservam a consistência da base
  - ▶ Estado inicial consistente -> Estado final consistente
- ▶ **Isolamento:** uma transação não vê o efeito de outra, até essa outra terminar
- ▶ **Durabilidade:** uma vez terminada, as alterações que ela fez permanecem no banco até que sejam explicitamente modificadas



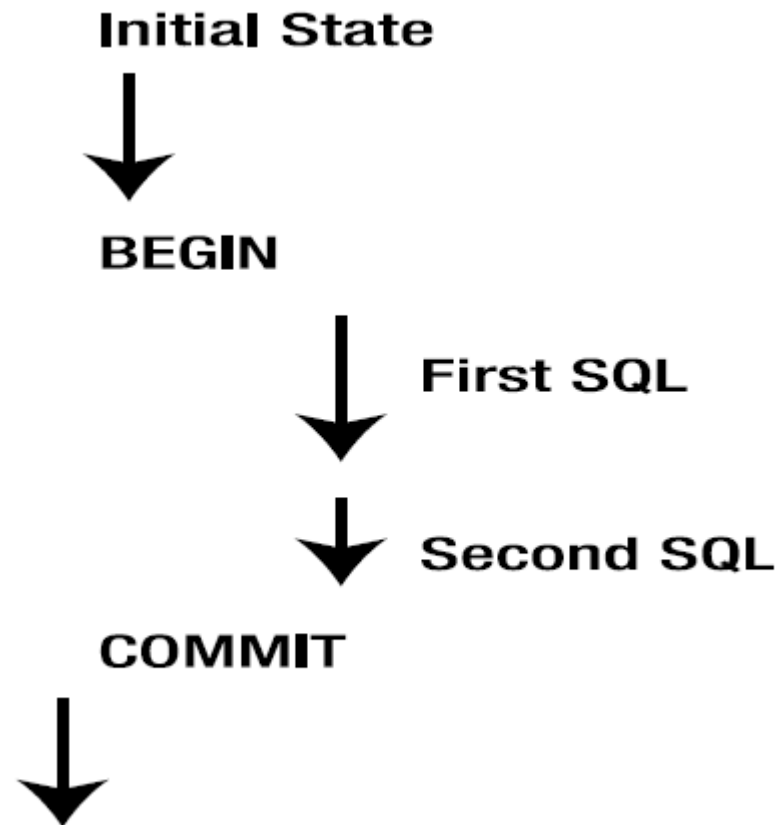
# ROLLBACK

---



# COMMIT

---



# EXEMPLO

---

```
test=> CREATE TABLE ttest1 (  
test(> ival1 integer,  
test(> sval1 varchar(64)  
test(> );  
CREATE TABLE  
test=> CREATE TABLE ttest2 (  
test(> ival2 integer,  
test(> sval2 varchar(64)  
test(> );  
CREATE TABLE  
test=>
```





# EXEMPLO

---

```
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');  
INSERT 17784 1  
test=> BEGIN;  
BEGIN  
test=> UPDATE ttest1 SET sval1 = 'Dave' WHERE ival1 = 1;  
UPDATE 1  
test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;
```

► Qual a saída?



# EXEMPLO

---

test=> ROLLBACK;

ROLLBACK

test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;

- ▶ Qual a nova saída?



## Exemplo 2 – Múltiplas tabelas

---

```
test=> DELETE FROM ttest1;
```

```
DELETE 1
```

```
test=> DELETE FROM ttest2;
```

```
DELETE 0
```

```
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
```

```
INSERT 17793 1
```

```
test=> BEGIN;
```

```
BEGIN
```

```
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');
```

```
INSERT 17794 1
```

```
test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;
```

```
UPDATE 1
```

```
test=> SELECT * FROM ttest1;
```



## Exemplo 2 – Múltiplas tabelas

---

```
test=> ROLLBACK;
```

```
ROLLBACK
```

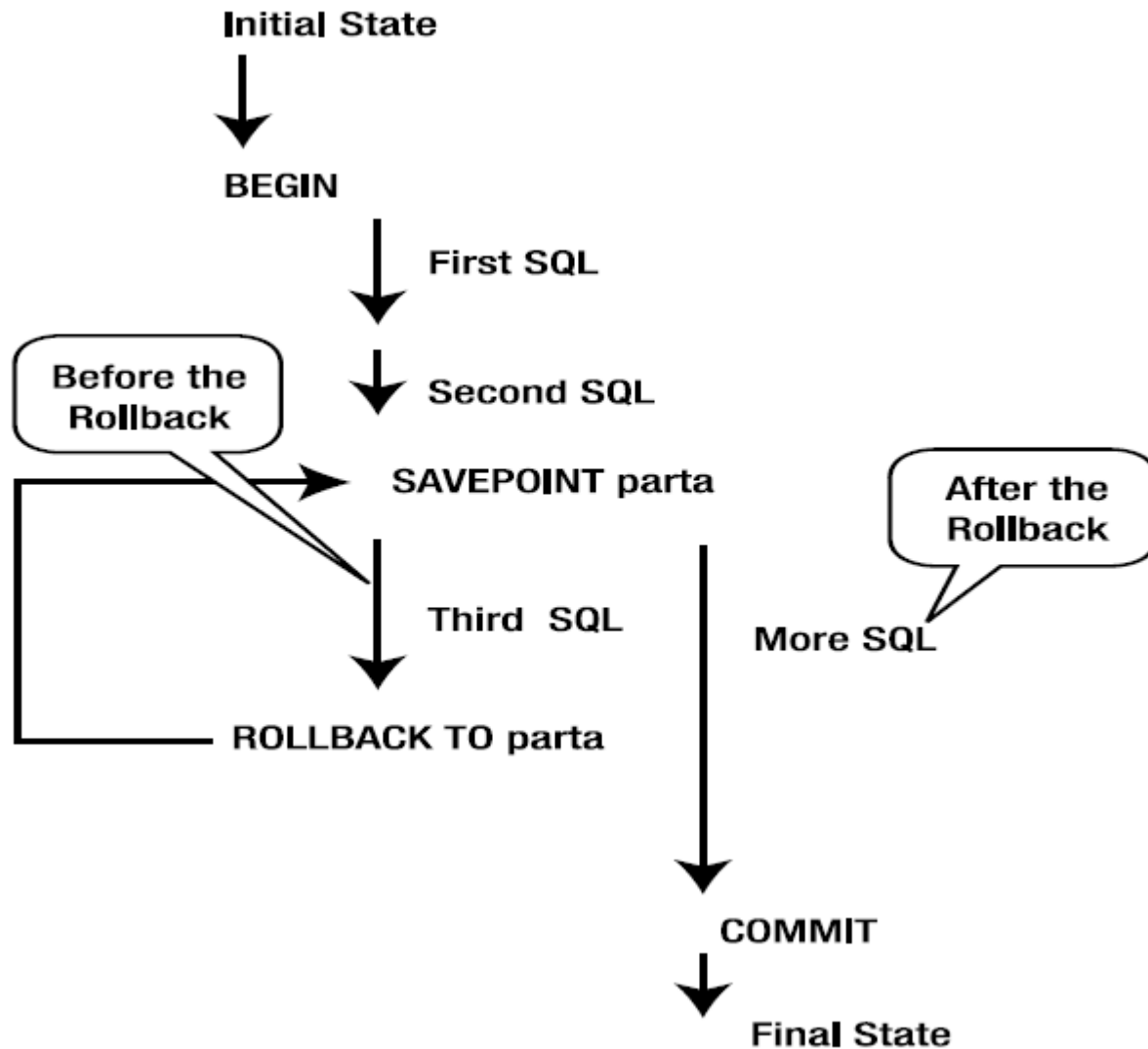
```
test=> SELECT * FROM ttest1;
```

```
test=> SELECT * FROM ttest2;
```

- ▶ Quais as saídas?



# SAVEPOINTS



# EXEMPLO

---

```
test=> DELETE FROM ttest1;
DELETE 1
test=> DELETE FROM ttest2;
DELETE 0
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
INSERT 17795 1
test=> BEGIN;
BEGIN
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');
INSERT 17796 1
test=> SAVEPOINT first;
SAVEPOINT
test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;
UPDATE 1
test=> SELECT * FROM ttest1;
```

► Saída?

---



# EXEMPLO

---

```
test=> ROLLBACK TO first;  
ROLLBACK  
test=> SELECT * FROM ttest1;  
test=> SELECT * FROM ttest2;
```

## ► Saída?

```
test=> ROLLBACK;  
ROLLBACK  
test=> SELECT * FROM ttest1;  
test=> SELECT * FROM ttest2;
```



# EXEMPLO

---

- ▶ Após o último ROLLBACK a transação está completa

```
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (99, 'Chris');
```

```
INSERT 17797 1
```

```
test=> COMMIT;
```

```
WARNING: there is no transaction in progress
```

```
COMMIT
```

- ▶ test=>

**ROLLBACK**





# EXEMPLO

---

- ▶ Após o COMMIT não é possível fazer um ROLLBACK

```
SELECT * FROM ttest2;
```

```
test=> BEGIN;
```

```
BEGIN
```

```
test=> UPDATE ttest2 SET sval2 = 'Gill' WHERE ival2 = 99;
```

```
UPDATE 1
```

```
test=> COMMIT;
```

```
COMMIT
```

```
test=> ROLLBACK;
```

```
WARNING:  there is no transaction in progress
```

```
ROLLBACK
```

```
test=> SELECT * FROM ttest2;
```



# Limitações das Transações

---

- ▶ Aninhamento de transação
  - ▶ Não é possível em Postgre (e varios outros SGBDs)
- ▶ Tamanho
  - ▶ Procure manter as transações pequenas para não sobrecarregar com travamentos (“lock”)
- ▶ Tempo
  - ▶ Os dados envolvidos em uma transação ficam travados (“lock”) para outros usuários e somente são disponibilizados após um COMMIT ou ROLLBACK. Por essa razão não é recomendável transações longas



# Locking (bloqueio)

---

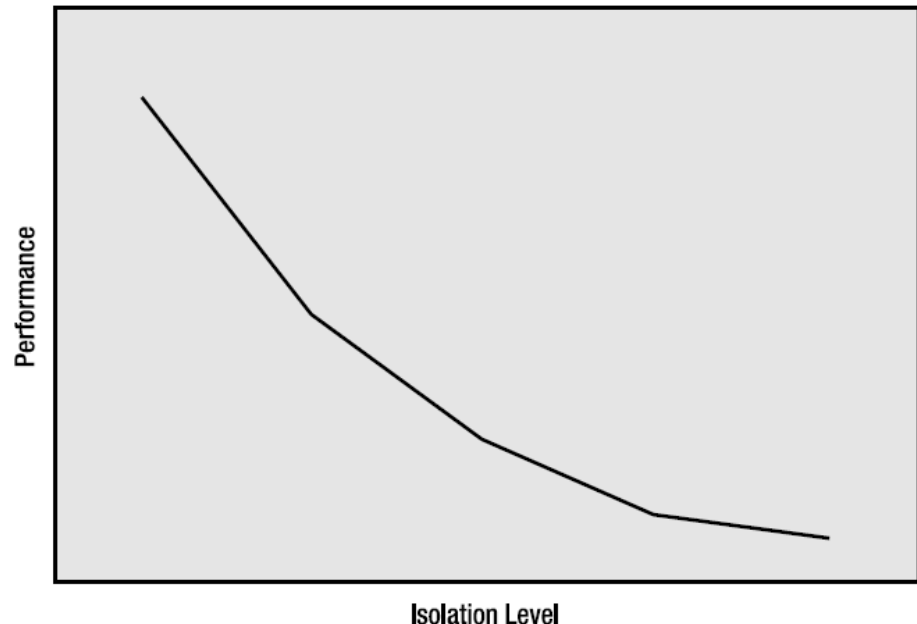
- ▶ A maioria dos BD implementam transações, particularmente para o isolamento de diferentes transações de usuários
- ▶ Um “shared lock” (bloqueio compartilhado) permite aos outros usuários lerem, mas não atualizarem os dados
- ▶ Um “exclusive lock” (bloqueio exclusivo) impede inclusive que outras transações de leiam os dados
- ▶ Exemplo: um servidor irá bloquear as linhas que estão sendo alteradas por uma transação até que a transação esteja completa – isso é feito automaticamente



# Isolamento versus performance multiusuário

---

- ▶ “Trade-off” entre
  - ▶ concorrência, performance e minimização do número de locks de um lado,
  - ▶ consistência e comportamento ideal do outro
  - ▶ Aumentar o isolamento diminui a performance multiusuário



# Locking

---

- ▶ Duas circunstâncias para a realização de um locking (bloqueio)
  - ▶ Evitar deadlocks
  - ▶ Bloqueio específico de uma aplicação



# Deadlocks

---

- ▶ O que acontece se duas aplicações diferentes tentam alterar o mesmo dado ao mesmo tempo

- ▶ Exemplo:

BEGIN;

UPDATE empregado SET  
pnome = 'Jose'  
WHERE SSN = '101';

UPDATE empregado  
SET pnome = 'Joana'  
WHERE SSN = '102';

BEGIN;

UPDATE empregado SET  
pnome = 'Márcio'  
WHERE SSN = '102';

UPDATE empregado  
SET pnome = 'Maria'  
WHERE SSN = '101';

---



# Deadlocks

---

ERRO: impasse detectado

DETALHE: Processo 4120 espera por ShareLock em transação 1502; bloqueado pelo processo 4792.

Processo 4792 espera por ShareLock em transação 1501; bloqueado pelo processo 4120.

DICA: See server log for query details.

- ▶ A seção que mostra o erro será cancelada (ROLLBACK) e na outra seção poderá ser executado um COMMIT
- ▶ O Deadlock é detectado automaticamente pelo PostgreSQL



# Deadlocks

---

BEGIN;

UPDATE empregado SET  
pnome = 'Jose'  
WHERE SSN = '101';  
**TRAVA LINHA SSN=101**

UPDATE empregado  
SET pnome = 'Joana'  
WHERE SSN = '102';  
**AGUARDA O  
DESTRAVAMENTO DA  
LINHA SSN=102**

BEGIN;

UPDATE empregado SET  
nome = 'Márcio'  
WHERE SSN = '102';  
**TRAVA LINHA  
SSN=102**

UPDATE empregado  
SET pnome = 'Maria'  
WHERE SSN = '101';  
**AGUARDA O  
DESTRAVAMENTO DA  
LINHA SSN=101**

---





# Evitando deadlocks

---

- ▶ Sempre mantenha a transação pequena, a menor possível. Isso evita o bloqueio de tabelas e linhas e diminui a chance de um deadlock
- ▶ Outra opção é sempre usar a mesma ordem no processamento de linhas e colunas
  - ▶ Se a ordem de alteração no exemplo anterior fosse a mesma o deadlock não ocorreria



# Lock explícitos

---

- ▶ É possível realizar o bloqueio de colunas e linhas explicitamente, embora isso deva sempre ser evitado
- ▶ O bloqueio somente é liberado depois de um COMMIT ou ROLLBACK
- ▶ Bloqueando colunas
  - ▶ Acrescente um FOR UPDATE no select;  
`SELECT * FROM matricula WHERE nota < 5 FOR UPDATE;`  
(em outro shell tente:  
`update matricula set nota = 3 where nusp = 8901;`)



# Lock explícitos

---

- ▶ Bloqueando tabelas
  - ▶ LOCK TABLE *nome da tabela*
  - ▶ Esse comando deve ser sempre evitado.



# Transações com múltiplos usuários

---

- ▶ Usuários concorrentes devem ser isolados uns dos outros.
- ▶ Níveis de isolamento
  - ▶ Dirty reads
    - ▶ Para uma determinada transação ocorre a leitura de dados que foram alterados por outra transação ainda não finalizada
  - ▶ Unrepeatable reads
    - ▶ Para uma determinada transação ocorre a leitura de dados que foram alterados por outra transação finalizada
  - ▶ Phantom reads
    - ▶ Ocorre quando novos dados são inseridos no banco ao mesmo tempo que a tabela afetada pela alteração sofre um update
  - ▶ Lost Updates
    - ▶ Ocorre quando duas transações alteram o mesmo dado e a segunda alteração causa a perda da primeira



**Table 9-2.** *Dirty Reads*

Transaction 1	Data Seen by Transaction 1	Data Seen by Other Transactions with Dirty Reads Allowed	Data Seen by Other Transactions with Dirty Reads Prohibited
BEGIN			
	David	David	David
UPDATE customer SET fname='Dave' WHERE customer_id = 15;			
	Dave	Dave	David
COMMIT			
	Dave	Dave	Dave
BEGIN			
UPDATE customer SET fname = 'David' WHERE customer_id = 15;			Dave
	David	David	Dave
ROLLBACK			
	Dave	Dave	Dave

\*\*\* Postgre não permite Dirty reads

**Table 9-3. Unrepeatable Reads**

<b>Transaction 1</b>	<b>Data Seen by Transaction 1</b>	<b>Data Seen by Other Transactions with Unrepeatable Reads Allowed</b>	<b>Data Seen by Other Transactions with Unrepeatable Reads Prohibited</b>
BEGIN		BEGIN	BEGIN
	David	David	David
UPDATE customer SET fname = 'Dave' WHERE customer_id = 15;			
	Dave	David	David
COMMIT			
	Dave	Dave	David
		COMMIT	COMMIT
		BEGIN	BEGIN
SELECT fname FROM customer WHERE customer_id = 15;		Dave	Dave



**Table 9-4.** *Phantom Reads*

Transaction 1	Transaction 2
BEGIN	BEGIN
UPDATE item SET sell_price = sell_price + 1;	INSERT INTO item(...) VALUES(...);
COMMIT	COMMIT



**Table 9-5. *Lost Updates***

User 1	Data Seen by User 1	User 2	Data Seen by User 2
Attempting to change the selling price from 21.95 to 22.55		Attempting to change the cost price from 15.23 to 16.00	
BEGIN		BEGIN	
SELECT cost_price, sell_price FROM item WHERE item_id = 1;	15.23, 21.95	SELECT cost_price, sell_price FROM item WHERE item_id = 1;	15.23, 21.95
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1;			
	15.23, 22.55		
COMMIT			15.23, 22.55
		UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1;	
	15.23, 22.55		16.00, 21.95
		COMMIT	
	16.00, 21.95		16.00, 21.95





► Evitando o Lost Update – solução não ideal, mas que reduz os riscos de um Lost Update

**Table 9-6.** *An Application Work-Around to Lost Updates*

User 1	Data Seen by User 1	User 2	Data Seen by User 2
Attempting to change the selling price from 21.95 to 22.55		Attempting to change the selling price from 21.95 to 22.99	
BEGIN		BEGIN	
Read sell_price WHERE item_id = 1	21.95	Read sell_price WHERE item_id = 1	21.95
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1 AND sell_price = 21.95;			
	22.55		21.95
COMMIT			
			22.55
		UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1 AND sell_price = 21.95;	
		Update fails with row not found, since the sell_price has been changed	