- Transações:

Inicialmente criar o banco de dados ‘estoque’:

```
CREATE DATABASE estoque

CREATE TABLE produto (cod_pro integer  primary key,
                      nome_pro varchar(20),
                      qtd_pro  int)

INSERT INTO produto VALUES (100, 'shampoo' , 30), (110, 'creme dental', 10), (130, 'fio dental', 23);

INSERT INTO produto VALUES (120, 'sabonete' , 32), (140, 'toalha', 60), (150, 'escova de dentes', 18);
```

1 - Exemplo – Rollback:

```
SELECT * FROM produto WHERE  cod_pro = 110;

 BEGIN;
        UPDATE produto SET qtd_pro= 40 WHERE cod_pro = 110;
        ROLLBACK;
        SELECT * FROM produto WHERE  cod_pro = 110;
END;
```

2 - Exemplo – Savepoint:

```
BEGIN;
  UPDATE produto SET qtd_pro= 40 WHERE cod_pro = 110;
  SAVEPOINT PONTO_1;
  UPDATE produto SET qtd_pro= 50 WHERE cod_pro = 120;
  SELECT * FROM produto WHERE  cod_pro = 120;
  ROLLBACK TO PONTO_1;
SELECT * FROM produto WHERE  cod_pro = 120;
  COMMIT;

SELECT * FROM produto WHERE  cod_pro = 110 OR cod_pro = 120 ;
```

Execução passo a passo:
- Iniciamos a transação e executamos um update para o produto 110;
- Criamos um savepoint chamado savepoint_1, assim temos agora um ponto salvo em nossa transação e podemos voltar a ele quando necessário;
- Atualizamos o produto 120 e depois executamos um SELECT para checar como ficou;

- Percebemos que o UPDATE que fizemos está errado, no produto 120, e queremos desfazer só ele, evitando a perda de toda a transação. Sendo assim, usamos o comando “rollback to savepoint_1”, que nos leva exatamente ao momento anterior ao update do produto 120;
- Agora executamos o SELECT do produto 120 e verificamos que a quantidade já estava como antes;
- Executamos o commit para confirma as alterações.

- Durabilidade

Exemplo:

```
BEGIN ;
  UPDATE produto SET qtd_pro= 40 WHERE cod_pro = 130;
COMMIT ;
```

- Isolamento e Consistência:

Em cenários que trabalhamos com tarefas concorrentes precisamos pensar em como isolar determinadas transações, não permitindo que uma alteração de dados na transação A afete a execução da transação B.

Os níveis de isolamento só existem porque existem problemas ao trabalhar-se com transações concorrente, que podem ocorrer ao trabalhar em multithread.

São três os possíveis problemas que você encontrará: Dirty Read, Nonrepeatable Read e Phatom Read.

Dirty Read

Conhecido também como “leitura suja”, é um dos piores problemas que podem ocorrer relacionados a integridade do banco de dados, sendo que a maioria dos bancos não permitem, em nenhuma circunstância, que isso ocorra. No PostgreSQL este tipo de problema nunca ocorrerá, pois a leitura suja ocorre quando uma transação A altera um determinado valor e uma transação B lê este valor alterado. A transação A executa um rollback, ou seja, não confirma a persistência dos dados alterados, porém a transação B já leu este valor “errado”.

Mesmo sabendo que esta situação não poderá ocorrer no PostgreSQL, vamos simular o que aconteceria se isso fosse permitido**.**

Exemplo: Transações com Dirty Read

```
  --TRANSAÇÃO A

  SELECT * FROM produto WHERE  cod_pro = 130;
BEGIN;
  UPDATE produto SET nome_pro = 'fita dental' WHERE cod_pro = 130;
    SELECT * FROM produto WHERE  cod_pro = 130;
  ROLLBACK ;
```

```
  --TRANSAÇÃO B
BEGIN;
--Vários comandos
UPDATE produto SET qtd_pro = 100  WHERE nome_pro = 'fio dental' ;
  COMMIT;
```

Ambas as transações acima devem ser executadas em instâncias diferentes, ou seja, se você estiver usando uma ferramenta como o PgAdmin você deve abrir duas janelas de consulta e executar cada uma em uma janela separada.

Imaginando que o PostgreSQL aceite Dirty Read vejamos o que aconteceria nesse exemplo.

1. Ambas as transações A e B são iniciadas.
2. A transação A executa primeiramente um UPDATE no produto 'fio dental' mudando seu nome para 'fita dental'.
3. Como a transação B executa vários comandos antes de chegar no seu UPDATE, quando ela chegar neste ponto o UPDATE da transação A já foi feito e quando a transação B tentar fazer um UPDATE no produto 'fio dental' nada será feito pois o nome do produto agora é 'fita dental'
4. Mesmo a transação A executando um rollback a transação B já está com o dado errado e a inconsistência de dados ocorre.

O PostgreSQL não permite que isso ocorra então ao realizar o UPDATE na transação B tudo funcionará normalmente visto que houve um rollback na transação A.

Nonrepeatable read

Este tipo de problema não chega a ser um problema em alguns tipos de sistemas concorrentes, por isso o padrão do PostgreSQL é permitir que isto ocorra.

No ***Dirty Read*** tínhamos um dado que foi alterado e depois não foi confirmado causando uma inconsistência a uma outra transação que o lê, porém neste caso os dados alterados são confirmados, através do *commit*. No ***Nonrepeatable read*** dados alterados e confirmados são “sentidos” pela outra transação que ainda não foi finalizada.

Exemplo: Transações com Nonrepeatable read

```
  --Transação A
BEGIN;
UPDATE produto SET nome_pro = 'fita dental' WHERE cod_pro = 130;
  COMMIT;

  --Transação B
BEGIN;
UPDATE produto SET qtd_pro = 400  WHERE nome_pro = 'fita dental' ;
  COMMIT;
```

Vejamos como testar o exemplo:

1. Executa o begin de ambas as transações para torná-las concorrentes;
2. Execute o UPDATE da transação A para mudar o nome de 'fio dental' para 'fita dental';
3. Agora execute o UPDATE da transação B para tentar mudar a quantidade de produto da 'fita dental' para 400. Você verá a seguinte mensagem: 0 rows affected. Isso ocorreu porque a transação A ainda não foi comitada e até o momento 'fio dental' ainda não é 'fita dental'", se a atualização ocorresse então teríamos um caso de Dirty Read;
4. Agora execute o COMMIT da transação A;
5. Depois execute novamente o UPDATE da transação B, agora você verá que 1 registro foi alterado (1 rows affected), isso porque a transação A já foi comitada/confirmada.

O nonrepeatable read serve tanto para alterações como deleções, mas não para inserções .

Phantom Read

Por fim temos o último problema que pode ocorrer em transações concorrentes, as leituras fantasmas. A lógica do Phantom Read é quase a mesma do Nonrepeatable read, tendo apenas uma única diferença: as inserções de novos registros.

Quando um dado é inserido em uma transação A e esta transação é comitada/confirmada, então este dando pode ser lido por uma transação B que ainda não foi confirmada. Lembre-se que no Nonrepeatable read nós tínhamos apenas as alterações e deleções como sendo propagadas para outras transações.

Exemplo: Transações com Phantom Read

```
  --Transação A
 BEGIN;
  INSERT INTO produto
 VALUES (155, 'sabonete' , 28);
  COMMIT;

  --Transação B
BEGIN;
  SELECT * FROM produto;
  COMMIT;
```

Vejamos como funciona o Phantom read no exemplo:

1. Inicie ambas as transações com o “begin”.
2. Execute o insert do novo registro na transação A.
3. Execute o SELECT na transação B. Você terá o seguinte retorno: 0 rows retrieved. Isso ocorre porque a transação A ainda não foi confirmada e voltamos a dizer que Dirty Read não são permitidas, seja para alterações, deleções ou inserções.
4. Execute o commit da transação A.
5. Agora execute o SELECT da transação B. Seu retorno será 1 linha contendo os

valores do novo registro inserido na transação A: 155, 'sabonete' , 28.

Entendendo os Níveis de Isolação

Os níveis de isolação configuram quais “problemas” podem e não podem ocorrer, é como dizer ao SGBD que você deseja que ele permita que ocorra um Dirty Read, ou um Nonrepeatable read ou um Phantom Read.

Por exemplo, não é possível permitir o Dirty Read no PostgresSQL, mas é possível controlar os outros “problemas”, permitindo ou não.

Existem quatro níveis de isolamento segundo o padrão SQL, e estes não são específicos somente para o PostgresSQL: Read Uncommited, Read Commited, Repeatable Read, Serializable.

1. **Read Uncommited:** Este é o nível menos isolado e o como o próprio nome já sugere, ele permite a leitura antes da confirmação. É exatamente o caso do Dirty Read que estudamos logo no início. Neste nível de isolação todos os problemas (os 3 citados nas seções anteriores) podem ocorrer sem restrição. É muito difícil que esse nível seja aplicado na prática pois poderiamos ter sérios problemas de consistência, por isso ele é considerado mais acadêmico, apenas para fins de estudos. O PostgreSQL não possui esse nível de isolação, evitando assim que este seja configurado.
2. **Read Commited:** Neste nível de isolação não podem ocorrer Dirty Reads mas são permitidos Nonrepeatable reads e Phantom Reads. Este é o nível padrão do PostgresSQL. Por este motivo que você conseguiu executar todos os testes das listagens abaixo sem problemas, pois o PostgreSQL tem como padrão o nível de isolamento Read Commited.
3. **Repeatable Readed:** Aqui apenas ocorrem Phantom Reads. O SGBD bloqueia o conjunto de dados lidos de uma transação, não permitindo leitura de dados alterados ou deletados mesmo que comitados pela transação concorrente, porém ele permite a leitura de novos registros comitados por outras transações.
4. **Serializable:** Este é o nível mais isolado que não permite nenhum tipo de problema (Dirty Read, Nonrepeatable read e Phantom Read).

Tabela de associação: Isolamento x Problemas (**Tabela 1**).

Nível de Isolamento	Dirty Read	Nonrepeatable read	Phantom Read
Read Uncommited	**Sim**	**Sim**	**Sim**
Read Commited	**Não**	**Sim**	**Sim**
Repeatable Read	**Não**	**Não**	**Sim**
Serializable	**Não**	**Não**	**Não**

Tabela 1. Isolamento X Problemas

O PostgreSQL tem algumas peculiaridades em relação aos níveis de isolação mostrados acima.

O exemplo a seguir mostra um destes níveis dentro de uma transação:

Exemplo 1: Usando nível de isolação no PostgreSQL

```
  --Transação A
BEGIN;
    UPDATE produto SET qtd_pro= 40 WHERE cod_pro = 110;
COMMIT;

  --Transação B
  BEGIN;
  set transaction ISOLATION LEVEL REPEATABLE READ;
  SELECT * FROM produto  where cod_pro = 110;
COMMIT;
```

Vamos entender o funcionamento do exemplo:

1. Por padrão, como dissemos anteriormente o PostgreSQL vem configurado com o nível de isolação Read Commited, mas neste caso vamos mudar a sua configuração para Repeatable Read.
2. Iniciamos as duas transações com o begin.
3. Na transação que gostaríamos de configurar o tipo de isolação nós chamamos o “set transaction ISOLATION LEVEL REPEATABLE READ”, assim toda a transação terá uma isolação diferente das outras.
4. Executamos o update na transação A e logo em seguida o commit.
5. O normal do PostgreSQL seria mostrar o resultado alterado ao executar o SELECT na transação B, mas como mudamos o nível de isolação para aceitar apenas Phantom Reads, ao realizar o SELECT a quantidade do produto continuará inalterada.

A mesma lógica aplica-se aos outros níveis, onde o mais isolado (serializable) não irá permitir que você veja nenhuma alteração, deleção ou inserção que forem realizadas fora da sua transação, totalmente isolada do “mundo exterior”.

No PostgreSQL há alguns tratamentos diferentes do padrão para os níveis mostrados acima e isso é muito importante para que você não pense que a explicação foi feita de forma errônea:

1. O nível REPEATABLE READ é tratado como SERIALIZABLE, ou seja, não permite leituras fantasmas.
2. O nível READ UNCOMMITED é tratado como READ COMMITED, e é exatamente por isso que você não consegue permitir o Dirty Read.