



favorito (3) imprimir anotar marcar como lido dúvidas?

Transações no PostgreSQL: Níveis de Isolamento

 (6)  (0)

Veja neste artigo como funcionam os níveis de isolamento das transações no PostgreSQL.

Neste artigo aprenderemos sobre o uso de transações no [PostgreSQL](#), sendo o foco principal estudar os níveis de isolamento e como o banco comporta-se diante de cada um destes. Mas antes de iniciarmos com a prática, precisamos entender alguns conceitos básicos que servem para [todos os SGBDs](#), que é o conceito de Atomicidade.

O acrônimo **ACID** faz referência aos quatro princípios básicos que qualquer SGBD deve satisfazer para que possua esse título:

- A = Atomicidade;
- C = Consistência;
- I = Isolamento;
- D = Durabilidade.

Nosso foco aqui será estudar a Atomicidade, que é algo indivisível ou, em termos mais simples, execute ou não execute nada.

Notificações :) princípio pode ser visto diariamente em transações bancárias como, por exemplo, quando você fere algum valor da sua conta para alguma outra conta, é realizado um UPDATE na sua conta, izando seu saldo (saldo atual – valor transferido), e a conta destino, que também recebe um UPDATE de atualização de saldo (saldo atual + valor transferido), certo?

todo o processo é desfeito para garantir a consistência dos dados. A isso chamamos de Atomicidade. Perceba que neste ponto, a Atomicidade e Consistência estão diretamente ligados.

Se não existisse Atomicidade teríamos sérios problemas de consistência, vejamos:

1. João tem R\$ 5000,00 na sua conta e transfere R\$ 500,00 para Pedro.
2. Internamente um UPDATE é executado no saldo de João que agora é R\$ 4500,00.
3. Um UPDATE começa a ser executado na conta de Pedro para receber os R\$ 500,00 mas algum problema ocorre e uma falha é retornada.
4. João continua com os R\$ 4500,00 e Pedro não recebeu os R\$ 500,00.

Veja o tamanho do problema que temos a frente, inconsistência de dados, e o pior: trabalhando com valores monetários.

Então já ficou claro que a Atomicidade garante a execução completa de um conjunto de comandos, caso contrário, tudo será cancelado. Mas como usamos tal conceito na prática? É o que veremos na próxima seção.

Transações

Transação é o sinônimo de Atomicidade no mundo do SGBD, então daqui em diante começaremos a usar o termo transação e você leitor deve entender que todo conceito de Atomicidade explicado na seção anterior aplica-se aqui de forma idêntica e o estudaremos aplicado ao PostgreSQL.

Quando tratamos de concorrência em SGBD, a primeira ideia que deve vir a mente são as “Transações”, pois estas podem garantir a execução de múltiplas threads em um mesmo instante de tempo. No

PostgreSQL existem dois comandos de suma importância para iniciar e finalizar uma transação, como mostra a **Listagem 1**.

Listagem 1. Iniciando e parando uma transação

```
BEGIN - -iniciar  
- - comandos
```

Acima temos o BEGIN para dar início a uma transação e o COMMIT para finalizar esta transação de forma “positiva”, ou seja, persistir as alterações na base de dados. Por outro lado, o ROLLBACK também finaliza a transação, mas cancelando qualquer alteração realizada. Ou seja, qualquer [comando SQL](#) que você execute é tratado dentro de uma transação, como mostra a **Listagem 2**.

Listagem 2. Transação implícita

```
--BEGIN (implícito)
UPDATE conta SET saldo = 100 WHERE id = 1;
--COMMIT (implícito)
```

Ao realizar o comando da **Listagem 2** implicitamente você está usando BEGIN e COMMIT, isso porque obrigatoriamente todos os comandos são executados dentro de uma transação, independente do seu tamanho (seja uma linha ou mil linhas).

Para que seja possível voltar a pontos anteriores depois de um ROLLBACK, o PostgreSQL trabalha com o conceito de Snapshots, que são como “fotos” do momento atual do banco antes de iniciar a transação, sendo que esta que será “comitada” só terá sucesso se aquele determinado snapshot não conflitar com dados de outras transações, recurso que veremos posteriormente nos níveis de isolamento.

A partir daqui vamos criar uma estrutura para começar a trabalhar de fato com transações e entender seu funcionamento na prática.

Primeiramente criamos a nossa tabela “conta”, como mostra a **Listagem 3**.

Listagem 3. Tabela conta

```
create table teste.conta(
  id integer primary key,
  cliente varchar(255) not null,
  saldo numeric(15,2) default 0
```

Notificações :)

a vamos inserir alguns dados em nossa tabela, como mostra a **Listagem 4**.

Listagem 4. Inserindo dados

```
(3, 'MARIO', 450);  
(4, 'JOAQUIM', 40000)
```

Finalmente vamos começar a trabalhar com transações usando nossa tabela “teste.conta”, como mostra a **Listagem 5**.

Listagem 5. Transação com rollback

```
select * from teste.conta where id = 1;  
begin;  
update teste.conta set saldo = 120 where id = 1;  
rollback;  
select * from teste.conta where id = 1;
```

Na primeira linha mostramos os dados da consulta antes de iniciar a transação. Em seguida começamos uma transação e atualizamos o saldo da conta “1” e logo em seguida executamos um rollback. O resultado das execuções acima não tem nenhum efeito nos dados do SGBD, isso porque o rollback anulou todas as alterações e isso significa que o resultado do primeiro [SELECT](#) será igual ao resultado do último SELECT.

Outro recurso muito interessante do PostgreSQL é o “SavePoint”, que possibilita que salvar determinado ponto dentro de uma transação e volte para ele quando achar necessário, como mostra a **Listagem 6**.

Listagem 6. Usando savepoint

```
begin;  
update teste.conta set saldo = 120 where id = 1;  
savepoint savepoint_1;  
update teste.conta set saldo = saldo - 1000 where id = 2;  
select * from teste.conta where id = 2;  
rollback to savepoint_1;  
select * from teste.conta where id = 2;  
commit;
```

Os acompanhar o passo a passo:

Iniciamos a transação e executamos um update na conta 1, atualizando o saldo para 120;

Criamos um savepoint chamado savepoint_1, assim temos agora um ponto salvo em nossa transação e podemos voltar a ele quando necessário;

Notificações :)

4. Percebemos que o UPDATE que fizemos está errado, na conta 2, 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 da conta 2;
5. Agora executamos o SELECT na conta 2 e verificamos que o valor do saldo já estava como antes;
6. Executamos o commit para confirma as alterações.

Isolamento de Transações

Até o momento as transações foram muitos úteis em ambientes stand alone, onde trabalhamos apenas com uma thread por vez, sem concorrência. Em cenários que trabalham com tarefas concorrentes a estória muda de contexto e já 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, e antes de vermos quais são esses níveis e como aplicá-los, nós temos que entender que tipo de problemas podem ocorrer ao trabalhar em multithread.

São três os possíveis problemas que você encontrará: Dirty Read, Nonrepeatable Read e Phantom Read. Citaremos estes em inglês para que você não fique confuso quem é quem na hora de executar os comandos.

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

Notificações :)
instância, que isso ocorra. No PostgreSQL este tipo de problema nunca ocorrerá, pois a leitura suja é quando uma transação A altera um determinado valor e uma transação B lê este valor alterado. A ação A executa um rollback, ou seja, não confirma a persistência dos dados alterados, porém a ação B já leu este valor “errado”.

Listagem 7. Transações com Dirty Read

```
--TRANSAÇÃO A
begin;
UPDATE teste.conta SET cliente = 'JOAO II' WHERE id = 1;
SELECT * FROM teste.conta;
rollback;

--TRANSAÇÃO B
begin
--Vários comandos
UPDATE teste.conta SET saldo = 300 WHERE cliente = 'JOAO';
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 na **Listagem 7**.

1. Ambas as transações A e B são iniciadas.
2. A transação A executa primeiramente um UPDATE no cliente 'JOAO' mudando seu nome para 'JOAO II'.
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 pelo cliente 'JOAO' nada será feito pois o nome do cliente agora é 'JOAO II'.
4. Mesmo a transação A executando um rollback a transação B já está com o dado errado e a inconsistência de dados ocorre.

Se como dissemos anteriormente 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.

Notificações :)

unrepeatable read

problema para todos.

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. Vejamos um exemplo na **Listagem 8**.

Listagem 8. Transações com Nonrepeatable read

```
--Transação A
begin;
UPDATE teste.conta SET cliente = 'JOAO II' WHERE id = 1;
commit

--Transação B
begin
update teste.conta set saldo = 350 WHERE cliente = 'JOAO II';
commit;
```

Vejamos como testar a **Listagem 8** de forma adequada:

1. Executa o begin de ambas as transações para torna-las concorrentes;
2. Execute o UPDATE da transação A para mudar o nome de JOAO para JOAO II;
3. Agora execute o UPDATE da transação B para tentar mudar o saldo de JOAO II para 350. Você verá a seguinte mensagem: 0 rows affected. Isso ocorreu porque a transação A ainda não foi comitada e até o momento “JOAO” ainda não é “JOAO II”, 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.

Nonrepeatable read serve tanto para alterações como deleções, mas não para inserções que vejam adiante.

Notificações :)



Phantom Read

inserções de novos registros.

Quando um dado é inserido em uma transação A e esta transação é comitada/confirmada, então este dado 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.

Listagem 9. Transações com Phantom Read

```
--Transação A
begin;
insert into teste.conta(id,cliente,saldo)
values (5,'NOVO CLIENTE',503);
commit

--Transação B
begin
SELECT * FROM teste.conta;
commit;
```

Vejamos como funciona o Phantom read no exemplo da **Listagem 9**.

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: 5;"NOVO CLIENTE";503.00

Notificações :)

tendendo os Níveis de Isolação

Vamos finalmente ao ponto que gostaríamos depois de explicar todos os detalhes necessários para seja possível o entendimento desta seção. Os níveis de isolação configuram quais “problemas” podem

Como já vimos não é possível permitir o Dirty Read no PostgreSQL, mas é possível controlar os outros “problemas”, permitindo ou não. Usamos aspas na palavra problemas pois neste caso vale ao DBA considerar se este é um problema ou não.

Existem quatro níveis de isolamento segundo o padrão SQL, e estes não são específicos somente para o PostgreSQL: Read Uncommitted, Read Committed, Repeatable Read, Serializable. Vamos entendê-los:

1. **Read Uncommitted:** 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 isolamento todos os problemas (os 3 citados nas seções anteriores) podem ocorrer sem restrição. É muito difícil que esse nível seja aplicado na prática pois poderíamos ter sérios problemas de consistência, por isso ele é considerado mais acadêmico, apenas para fins de estudos. O PostgreSQL não possui esse nível de isolamento, evitando assim que este seja configurado.
2. **Read Committed:** Neste nível de isolamento não podem ocorrer Dirty Reads mas são permitidos Nonrepeatable reads e Phantom Reads. Este é o nível padrão do PostgreSQL. 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 Committed.
3. **Repeatable Read:** 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).

Veamos como ficou nossa tabela de associação: Isolamento x Problemas (**Tabela 1**).

Notificações :)	de Isolamento	Dirty Read	Nonrepeatable read	Phantom Read
	Uncommitted	Sim	Sim	Sim
	Committed	Não	Sim	Sim

Tabela 1. Isolamento X Problemas

Porém ainda não terminamos, o PostgreSQL tem algumas peculiaridades em relação aos níveis de isolamento mostrados acima.

Primeiramente vamos ver como usar um destes níveis dentro de uma transação, como mostra a **Listagem 10**.

Listagem 10. Usando nível de isolamento no PostgreSQL

```
--Transação A
begin;
update teste.conta set saldo = 120 where id = 1;
commit;

--Transação B
begin
set transaction ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM teste.conta where id = 1;
commit;
```

Vamos entender o funcionamento da **Listagem 10**:

1. Por padrão, como dissemos anteriormente o PostgreSQL vem configurado com o nível de isolamento Read Committed, 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 isolamento nós chamamos o “set transaction ISOLATION LEVEL REPEATABLE READ”, assim toda a transação terá uma isolamento diferente das outras.

Executamos o update na transação A e logo em seguida o commit.

O normal do PostgreSQL seria mostrar o resultado alterado ao executar o SELECT na transação B, mas como mudamos o nível de isolamento para aceitar apenas Phantom Reads, ao realizar o SELECT o saldo da conta 1 continuará inalterado.

Notificações :)

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 UNCOMMITTED é tratado como READ COMMITED, e é exatamente por isso que você não consegue permitir o Dirty Read.

O objetivo principal deste artigo foi mostrar o uso dos níveis de isolamento no **PostgreSQL**, porém isso não seria possível sem antes passar por todo o caminho necessário da Atomicidade, Transação, Problemas de Concorrência para só então chegar na isolamento de transação. Mostramos em detalhes como funcionam cada um dos níveis e quais tipos de problemas eles permitam que ocorram ou não.

Vale ressaltar ainda que por tratarmos estes como “Problemas” não quer dizer que de fato são problemas e podem ser encarados como soluções para alguns casos, e é exatamente por isso que os níveis de isolamento podem ou não permitir que eles ocorram dependendo da regra de cada projeto em especial.

por **Ronaldo Lanhellas**

 (6)  (0)

Ficou com alguma dúvida?

Notificações :)



DEVMEDIA

Login

APIs

Fale conosco



Hospedagem web por Porta 80 Web Hosting

Notificações :)