Artigo

Trabalhando com Transações no PostgreSQL

O artigo trata da definição do conceito de transações, estados das transações e como o PostgreSQL trabalha com esse conceito. Além disso, demonstra através de um estudo de caso prático como lidar com os principais comandos de transação no PostgreSQL.

De que trata o artigo: O artigo trata da definição do conceito de transações, estados das transações e como o PostgreSQL trabalha com esse conceito. Além disso, demonstra através de um estudo de caso prático como lidar com os principais comandos de transação no PostgreSQL.

Para que serve: O conteúdo do artigo traz os benefícios de se implementar operações concorrentes no SGBD para que as mesmas não sejam conflitantes e deixem o banco de dados em um estado consistente.

Em que situação o tema é útil: Essa estratégia é útil quando está sendo desenvolvida qualquer aplicação que tenha acesso a banco de dados e que se

O termo transação refere-se a uma coleção de operações que formam uma única unidade de trabalho lógica. Por exemplo, a transferência de dinheiro de uma conta para outra é uma transação consistindo de duas atualizações, uma para cada conta.

Uma transação é uma unidade de execução do programa que acessa e possivelmente atualiza vários itens de dados. Para garantir a integridade dos dados, é necessário que o SGBD mantenha as seguintes propriedades das transações: atomicidade, consistência, isolamento e durabilidade.

- Atomicidade: uma transação é uma unidade atômica de processamento; ou ela será executada em sua totalidade ou não será de modo nenhum.
- Consistência: uma transação deve ser preservadora de consistência se sua execução completa fizer o banco de dados passar de um estado consistente para outro também consistente.
- Isolamento: uma transação deve ser executada como se estivesse isolada das demais. Isto é, a execução de uma transação não deve sofrer interferência de quaisquer outras transações concorrentes.
- Durabilidade: as mudanças aplicadas ao banco de dados por uma transação efetivada devem persistir no banco de dados. Essas mudanças não devem ser perdidas em razão de uma falha

Essas propriedades normalmente são conhecidas como propriedades ACID. Esse acrônimo é derivado da primeira letra de cada uma das quatro propriedades.

Quando se trabalham com transações, é necessário que se faça pelo menos duas ressalvas. A primeira é que em certas situações é interessante se agregar vários comandos como sendo integrantes de uma mesma transação, como, por exemplo, em uma transferência bancária que envolve a retirada de dinheiro de uma conta e o acréscimo em outra como se fosse apenas uma única operação lógica. A segunda

Na ausência de falhas, todas as transações são completadas com sucesso. Porém, uma transação nem sempre pode completar sua execução com sucesso. Caso isso ocorra, essa transação é considerada abortada.

Se tivermos que garantir a propriedade de atomicidade, uma transação abortada não pode ter efeito sobre o estado do banco de dados. Assim, qualquer mudança que a transação abortada tenha feito no banco de dados deve ser desfeita. Quando as mudanças causadas por uma transação abortada tiverem sido desfeitas, dizemos que a transação foi revertida (rolled back).

Uma transação que completa sua execução com sucesso é considerada confirmada (committed). Uma transação confirmada que realizou atualizações transforma o banco de dados em um novo estado consistente, que precisa persistir mesmo que haja uma falha no sistema. Quando uma transação tiver sido confirmada, não podemos desfazer seus efeitos abortando-a. A única forma de desfazer os efeitos de uma transação confirmada é executar uma transação de compensação.

Em resumo, uma transação precisa estar em um dos seguintes estados:

- Ativa: é o seu estado inicial. A transação permanece nesse estado enquanto está sendo executada.
- Parcialmente confirmada: é o estado depois que a instrução final foi executada.
- Falha: é o estado depois da descoberta de que a execução normal não pode mais prosseguir.
- • Abortada: estado depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação.
- Confirmada: estado após o término bem-sucedido.

A **Figura 1** apresenta o diagrama de estado de uma transação.

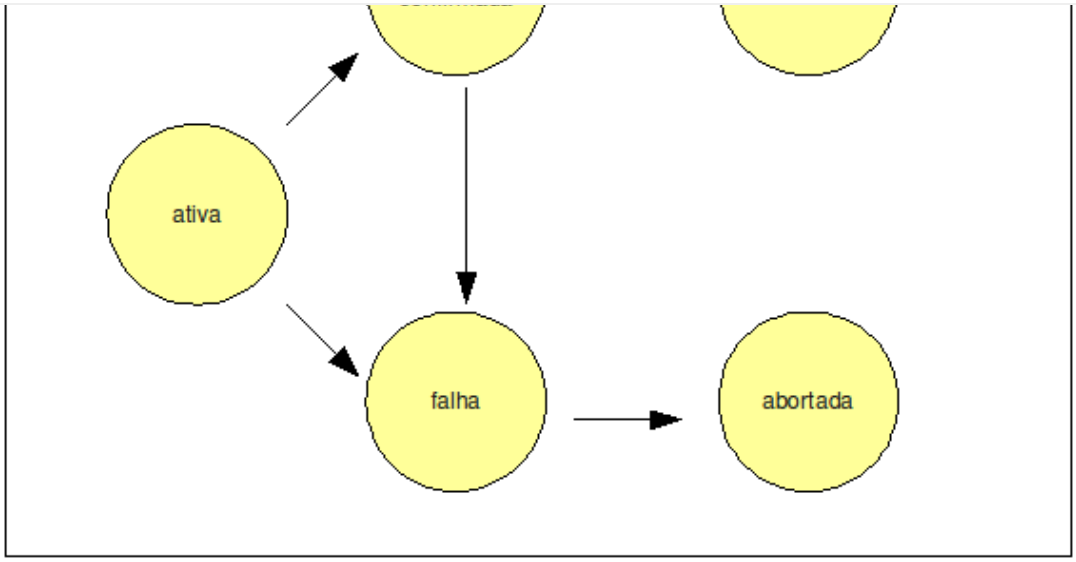

Figura 1. Diagrama dos estados de uma transação

Transações no PostgreSQL

Diferentemente dos SGBD's tradicionais, que usam bloqueios para controlar a simultaneidade, o PostgreSQL mantém a consistência dos dados utilizando o modelo multi-versão (Multiversion Concurrency Control, MVCC).

Isto significa que ao consultar o banco de dados, cada transação enxerga um estado do banco de dados, ou seja, como este era há um tempo atrás, sem levar em consideração o estado corrente dos dados subjacentes. Este modelo protege a transação para não enxergar dados inconsistentes, o que poderia ser causado por atualizações feitas por transações simultâneas nas mesmas linhas de dados, fornecendo um isolamento da transação para cada sessão do banco de dados.

A principal vantagem de utilizar o modelo de controle de simultaneidade MVCC em vez de bloqueios é que no MVCC os bloqueios obtidos para consultar dados (leitura) não conflitam com os bloqueios obtidos para escrever dados e, portanto, a leitura nunca bloqueia a escrita, e a escrita nunca bloqueia a leitura.

ROLLBACK.

O comando BEGIN inicia um bloco de comandos SQL que fazem parte de uma transação. A **Listagem 1** demonstra o comando que inicia uma transação.

Listagem 1. Comando que inicia uma transação

```
1 | BEGIN [ WORK | TRANSACTION ] [ modo_da_transação [, ...] ]
```

As palavras WORK e TRANSACTION são equivalentes e são também opcionais. Já a opção modo_da_transação deve conter um dos itens presentes na **Listagem 2**.

Listagem 2. Modos de uma transação

```
1 | ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
2 | READ WRITE | READ ONLY
```

Os modo_da_transação serão mencionados mais adiante na seção Isolamento de Transação.

Os comandos realizados após o BEGIN só serão persistidos em disco, e seus resultados só serão apresentados aos demais usuários do banco após a efetivação com o uso do comando COMMIT.

Uma transação é finalizada pelo comando COMMIT, o qual dispara a efetivação da transação no banco de dados e a torna visível para os demais usuários os resultados da

Listagem 3. Comando de confirmação de uma transação

```
1 | COMMIT [ WORK | TRANSACTION ]
```

Assim como no comando BEGIN, as palavras WORK e TRANSACTION também são opcionais.

O terceiro comando citado é o ROLLBACK, que por sua vez aborta a transação que está em andamento, impedindo que as alterações nos dados nela realizadas sejam persistidas no banco de dados. O comando que aborta uma transação está demonstrado na **Listagem 4**.

Listagem 4. Comando que aborta uma transação

```
1 | ROLLBACK [ WORK | TRANSACTION ]
```

Conhecendo Save Points

A partir da versão 8.0 do PostgreSQL, novas possibilidades surgem também para os desenvolvedores de aplicativos de banco de dados. O suporte a pontos de restauração (save points) é uma opção bastante interessante. Essa ampliação permite que se insiram em uma nova transação outras subtransações. Quando o banco de dados encontra uma definição SAVEPOINT, ele marca o status alcançado até aquele momento. As operações subseqüentes podem ser canceladas posteriormente por meio do comando ROLLBACK TO SAVEPOINT dentro da transação até esse ponto congelado. Caso não seja mais necessário um save point, então é possível liberá-lo usando o

Listagem 5. Comando de criação de um SAVEPOINT

```
1 SAVEPOINT nome
```

No comando da **Listagem 5** nome é o nome do SAVEPOINT que está sendo criado.

Para que se possa entender o seu uso, observe a **Listagem 6**. Na **Listagem 6**, sequencialmente, foi inserido o registro 1, em seguida criou-se o save point chamado SP, inseriu-se mais um registro de valor 2, executou-se um comando rollback para o save point SP, e por fim foi inserido um registro 3. Toda essa seqüência de passos resultou na inserção dos registros 1 e 2, em seguida foi cancelada a inserção do registro 2 através do comando rollback que retornou o banco de dados até o momento anterior à inserção deste registro, e por fim confirmou a inserção do registro de valor 3.

Listagem 6. Exemplo de uso de um SAVEPOINT

```
1 BEGIN TRANSACTION;
2 INSERT INTO tabela VALUES (1);
3 SAVEPOINT sp;
4 INSERT INTO tabela VALUES (2);
5 ROLLBACK TO SAVEPOINT sp;
6 INSERT INTO tabela VALUES (3);
7 COMMIT;
```

Esse recurso é especialmente útil no caso de transações mais longas. O que acontecia nas versões anteriores era que uma transação precisava ser cancelada por completo

A propriedade de isolamento é uma das quatro propriedades ACID (atomicidade, consistência, isolamento e durabilidade) que uma unidade lógica de trabalho deve ter para que seja qualificada como uma transação.

Ela consiste na habilidade para proteger transações dos efeitos de atualizações executadas por outras transações simultâneas. O nível de isolamento é realmente personalizável para cada transação.

A definição de níveis de isolamento da transação permite aos programadores trocarem o risco crescente de problemas de integridade por maior acesso simultâneo aos dados.

Diversos problemas podem ocorrer quando transações concorrentes são executadas de maneira descontrolada. Além disso, existem algumas anomalias que podem ocorrer quando transações são realizadas de forma não controlada. Tais anomalias são: Dirty Read, Non-Repeatable Read e Phantom Read. A seguir falaremos sobre cada um deles.

Dirty Read

Dirty Read, também conhecida como “Leitura suja”, ocorre quando uma transação T1 modifica determinada informação e, em seguida, uma outra transação T2 lê a mesma informação antes que T1 confirme (COMMIT) ou desfaça (ROLLBACK) tal operação. Caso T1 faça ROLLBACK, T2 leu uma informação que nunca chegou a existir oficialmente no banco de dados.

Non-Repeatable Read

Non-Repeatable Read ou “Leitura não-repetível” ocorre quando uma transação T1 pode ler um dado valor em uma tabela. Se num momento futuro uma outra transação

Phantom Read, também conhecida como “Leitura fantasma”, acontece quando uma transação T1 lê um grupo de registros que satisfazem uma dada condição e, em seguida, uma transação T2 modifica ou insere um registro que satisfaz essa condição e faz COMMIT. Em seguida, se T1 voltar a ler os registros nas mesmas condições, irá obter um resultado diferente.

Os quatro níveis de isolamento de transação e seus comportamentos correspondentes estão descritos na **Tabela 1.**

Os valores possível e impossível indicam que o fenômeno indesejado pode ou não ocorrer, dependendo de qual nível de isolamento esteja sendo utilizado.

Tabela 1. Níveis de Isolamento.

Nível de Isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncomitted	Impossível	Possível	Possível
Read comitted	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

No PostgreSQL pode ser requisitado qualquer um dos quatros níveis de isolamento padrão. Porém, internamente só existem dois níveis de isolamento distintos, correspondendo aos níveis de isolamento Read Committed e Serializable. Quando é selecionado o nível de isolamento Read Committed realmente obtém-se Read Committed, mas quando é selecionado Repeatable Read na realidade é obtido Serializable. Portanto, o nível de isolamento real pode ser mais restrito do que o selecionado.

Os quatro níveis de isolamento somente definem quais fenômenos não podem acontecer, mas não definem quais fenômenos devem acontecer. O motivo pelo qual o

Para se definir o nível de isolamento da transação, utiliza-se o comando SET TRANSACTION.

A **Listagem 7** demonstra a utilização do comando SET TRANSACTION.

Listagem 7. Exemplo de uso do comando SET TRANSACTION

```
1 | SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

O nível de isolamento de uma transação determina quais dados a transação pode enxergar quando outras transações estão processando ao mesmo tempo.

Se o nível de isolamento for o Read Committed, os comandos enxergarão apenas as linhas efetivadas (commit) antes do início da sua execução. Este é o padrão do PostgreSQL.

Por outro lado, se o nível de isolamento for o Serializable, a transação corrente enxerga apenas as linhas efetivadas (commit) antes da primeira consulta ou instrução que modifique os dados ter sido executada nesta transação.

Intuitivamente, o nível serializável significa que, duas transações concorrentes deixam o banco de dados no mesmo estado que estas duas transações, executadas uma após a outra em qualquer ordem, deixaria.

A seguir veremos um pouco mais sobre cada nível de isolamento.

Nível de isolamento Read Committed

execução da consulta (entretanto, o SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). Na verdade, o comando SELECT enxerga um instantâneo do banco de dados, como este era no instante em que a consulta começou a executar. Deve ser observado que dois comandos SELECT sucessivos podem enxergar dados diferentes, mesmo estando dentro da mesma transação, se outras transações efetivarem alterações durante a execução do primeiro comando SELECT.

Os comandos UPDATE, DELETE e SELECT FOR UPDATE se comportam do mesmo modo que o SELECT para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início do comando. Entretanto, no momento em que foi encontrada alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação simultânea. Neste caso, a transação que pretende atualizar fica aguardando a transação de atualização que começou primeiro efetivar ou desfazer (se ainda estiver executando).

Se a transação de atualização que começou primeiro desfizer as atualizações, então seus efeitos são negados e a segunda transação de atualização pode prosseguir com a atualização da linha original encontrada. Se a transação de atualização que começou primeiro efetivar as atualizações, a segunda transação de atualização ignora a linha caso tenha sido excluída pela primeira transação de atualização, senão tenta aplicar sua operação na versão atualizada da linha. A condição de procura do comando (a cláusula WHERE) é avaliada novamente para verificar se a versão atualizada da linha ainda corresponde à condição de procura. Se corresponder, a segunda transação de atualização prossegue sua operação começando a partir da versão atualizada da linha.

Devido à regra acima, é possível um comando de atualização enxergar um instantâneo inconsistente: pode enxergar os efeitos dos comandos simultâneos de atualização que afetam as mesmas linhas que está tentando atualizar, mas não enxerga os efeitos

Listagem 8.

Listagem 8. Exemplo de problema ao usar Read Commited

```
1 BEGIN;
2 UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 12345;
3 UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 7534;
4 COMMIT;
```

Se duas transações deste tipo tentarem mudar ao mesmo tempo o saldo da conta 12345, é claro que desejamos que a segunda transação comece a partir da versão atualizada da linha da conta. Como cada comando afeta apenas uma linha predeterminada, permitir enxergar a versão atualizada da linha não cria nenhum problema de inconsistência.

Como no modo Read Committed cada novo comando começa com um novo instantâneo incluindo todas as transações efetivadas até este instante, de qualquer modo os próximos comandos na mesma transação vão enxergar os efeitos das transações simultâneas efetivadas. O ponto em questão é se, dentro de um único comando, é enxergada uma visão totalmente consistente do banco de dados.

O isolamento parcial da transação fornecido pelo modo Read Committed é adequado para muitos aplicativos, e este modo é rápido e fácil de ser utilizado. Entretanto, para aplicativos que efetuam consultas e atualizações complexas, pode ser necessário garantir uma visão do banco de dados com consistência mais rigorosa que a fornecida pelo modo Read Committed.

Nível de isolamento serializável

após a outra, em série, em vez de simultaneamente. Entretanto, os aplicativos que utilizam este nível de isolamento devem estar preparados para tentar executar novamente as transações, devido a falhas de serialização.

Quando uma transação está no nível serializável, o comando SELECT enxerga apenas os dados efetivados antes da transação começar; nunca enxerga dados não efetivados ou alterações efetivadas durante a execução da transação por transações simultâneas (entretanto, o comando SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). É diferente do Read Committed, porque o comando SELECT enxerga um instantâneo do momento de início da transação, e não do momento de início do comando corrente dentro da transação. Portanto, comandos SELECT sucessivos dentro de uma mesma transação sempre enxergam os mesmos dados.

Os comandos UPDATE, DELETE e SELECT FOR UPDATE se comportam do mesmo modo que o comando SELECT para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início da transação. Entretanto, alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação simultânea no momento em que foi encontrada. Neste caso, a transação serializável aguarda a transação de atualização que começou primeiro efetivar ou desfazer as alterações (se ainda estiver executando). Se a transação que começou primeiro desfizer as alterações, então seus efeitos são negados e a transação serializável pode prosseguir com a atualização da linha original encontrada. Porém, se a transação que começou primeiro efetivar (e realmente atualizar ou excluir a linha, e não apenas selecionar para atualização), então a transação serializável é desfeita com a mensagem “ERRO: não foi possível serializar o acesso devido a atualização simultânea”, porque uma transação serializável não pode alterar linhas alteradas por outra transação após a transação serializável ter começado

vez em diante, a transação passa a enxergar a alteração efetivada anteriormente como parte da sua visão inicial do banco de dados e, portanto, não existirá conflito lógico em usar a nova versão da linha como ponto de partida para atualização na nova transação.

Deve ser observado que somente as transações que fazem atualizações podem precisar de novas tentativas; as transações somente para leitura nunca estão sujeitas a conflito de serialização.

O modo serializável fornece uma garantia rigorosa que cada transação enxerga apenas visões totalmente consistentes do banco de dados. Entretanto, o aplicativo deve estar preparado para executar novamente a transação quando atualizações simultâneas tornarem impossível sustentar a ilusão de uma execução serial. Como o custo de refazer transações complexas pode ser significativo, este modo é recomendado somente quando as transações efetuando atualizações contêm lógica suficientemente complexa a ponto de produzir respostas erradas no modo Read Committed. Habitualmente, o modo serializável é necessário quando a transação executa vários comandos sucessivos que necessitam enxergar visões idênticas do banco de dados.

Estudo de Caso – Vídeo Locadora

Para demonstrar como se trabalha com o conceito de transação no PostgreSQL, foi criado um banco de dados de uma locadora de DVD fictícia.

O banco de dados em questão possui as tabelas cliente, locacao, filme e tipo_filme.

A tabela cliente armazena as informações referentes aos clientes da locadora (identificador e seu nome), onde os mesmos poderão realizar vários empréstimos de filmes diferentes.

A tabela filme guarda informações sobre os filmes existentes na locadora,

tipo um filme é, a tabela tipo_filme armazena informações sobre quantos dias aquele tipo de filme pode ficar em posse do cliente e o valor da multa diário por atraso na devolução.

Por fim, a tabela locacao armazena informações inerentes ao aluguel de filmes por um cliente. Ela guarda informações sobre o cliente responsável pelo aluguel (chave estrangeira da tabela cliente), o filme alugado (chave estrangeira da tabela filme), a data em que um filme foi alugado, a data prevista para sua devolução, a data que o filme foi realmente devolvido, seu status (em locação, atrasado, devolvido, devolvido com atraso) e, caso seja devolvido em atraso, armazena também o valor da multa (valor da multa do tipo do filme X quantidade de dias de atraso).

A **Figura 2** apresenta o diagrama Entidade-Relacionamento do banco de dados em estudo, apresentando as tabelas, seus atributos, suas chaves-primárias e estrangeiras e o relacionamento entre as tabelas, seguindo a descrição do estudo de caso apresentada anteriormente.

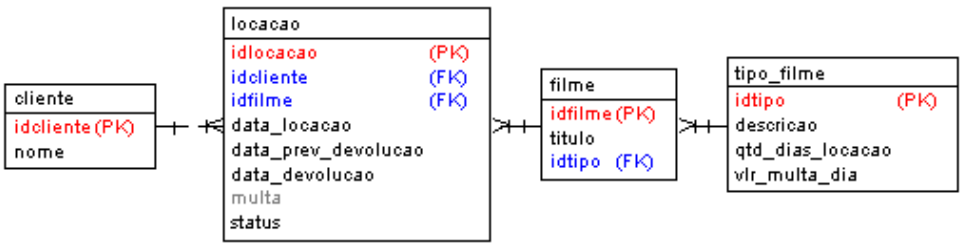

Figura 2. Diagrama entidade-relacionamento do banco de dados de uma Locadora.

Criando o Banco de Dados Locadora

A versão 8.3 do PostgreSQL foi utilizada para a demonstração dos comandos de

Listagem 9. Comando SQL para criação do banco locadora

```
1  CREATE DATABASE locadora;
```

As tabelas do banco de dados locadora são criadas pelos comandos da **Listagem 10**.

Listagem 10. Comandos SQL para criação das tabelas do banco locadora

```
1  Create table cliente
2  (
3  idcliente Integer NOT NULL,
4  nome Varchar NOT NULL,
5  primary key (idcliente)
6  ) Without Oids;
7  Create table filme
8  (
9  idfilme Integer NOT NULL,
10 titulo Varchar(30) NOT NULL,
11 idtipo Integer NOT NULL,
12 primary key (idfilme)
13 ) Without Oids;
14 Create table locacao
15 (
16 idlocacao Serial NOT NULL,
17 idcliente Integer NOT NULL,
18 idfilme Integer NOT NULL,
19 data_locacao Date NOT NULL,
20 data_prev_devolucao Date NOT NULL,
21 data_devolucao Date NOT NULL,
22 multa Numeric(9,2),
23 status Char(1) NOT NULL,
24 primary key (idlocacao)
25 ) Without Oids;
```

```
31  vlr_multa_dia Numeric(9,2) NOT NULL,
32  primary key (idtipo)
33  ) Without Oids;
34  Alter table locacao add foreign key (idcliente) references cliente (idcl
35  restrict;
36  Alter table locacao add foreign key (idfilme) references filme (idfilme)
37  Alter table filme add foreign key (idtipo) references tipo_filme (idtipo
38  restrict;
```

Trabalhando com transações no banco de dados Locadora

Após termos criado todas as tabelas do nosso banco de dados, iremos agora trabalhar com os comandos de manipulação de transações.

Anteriormente, neste artigo, já foi mencionada a função de cada um dos comandos (BEGIN, ROLLBACK, COMMIT, SAVEPOINT e ROLLBACK TO SAVEPOINT) que nos possibilitam manipular transações de banco de dados. A **Listagem 11** apresenta uma transação que foi efetivada com sucesso.

Listagem 11. Comandos de uma transação efetiva com sucesso

```
1  BEGIN;
2  INSERT INTO cliente VALUES (1, 'RANGEL');
3  COMMIT;
```

Na **Listagem 11**, o comando BEGIN serve para iniciar explicitamente uma nova transação. Neste exemplo, foi inserido um novo cliente na locadora (com o nome

Em alguns casos, têm-se a necessidade de desfazer alguma operação seja por uma falha ou por algum erro que ocorra durante a sua execução.

Para desfazer as modificações feitas por uma transação e deixar o banco de dados em um estado consistente, devemos, por exemplo, fazer executar um comando igual ao descrito na **Listagem 12**.

Listagem 12. Comandos para desfazer modificações feitas por uma transação

```
1 BEGIN;
2 INSERT INTO cliente VALUES (2 , 'MYCHELLE');
3 ROLLBACK;
```

A **Listagem 12** mostra uma transação que foi iniciada (BEGIN), e em seguida fez uma inserção do registro MYCHELLE na tabela cliente. No entanto, por algum motivo deseja-se que a transação seja desfeita. Para isso, foi utilizado o comando ROLLBACK. O comando ROLLBACK deixou a transação no estado que se encontrava anteriormente, ou seja, não inseriu a cliente MYCHELLE, e deixou o banco de dados apenas com os clientes anteriormente cadastrados (no nosso caso apenas o cliente RANGEL). Isto pode ser observado fazendo uma consulta (select) simples na tabela cliente, conforme **Figura 3**.

Imagine que estejamos inserindo dois tipos de filmes (tabela tipo_filme), conforme a **Listagem 13.** Neste caso, desejamos apenas desfazer o segundo INSERT, ou seja, não queremos desfazer a transação por completo.

Listagem 13. Comandos SQL usando SAVEPOINT

```
1  BEGIN;
2  INSERT INTO tipo_filme VALUES (1, 'LANCAMENTO', 1, 2.00);
3  SAVEPOINT ponto_marcado;
4  INSERT INTO tipo_filme VALUES (2, 'CATALOGO', 2, 1.00);
5  ROLLBACK TO ponto_marcado;
6  COMMIT;
```

Na **Listagem 13**, depois de iniciada a transação (BEGIN), é inserido o tipo de filme “lançamento”. Após a inserção deste registro, cria-se um SAVEPOINT chamado de ponto_marcado. A partir daí, podemos fazer uso em qualquer ponto da nossa transação do comando ROLLBACK TO para retornar para este ponto. Continuando nesta listagem, em seguida é inserido o tipo de filme “catalogo”.

Ao final, a **Listagem 13** utiliza o comando ROLLBACK TO ponto_marcado para voltar até o SAVEPOINT definido anteriormente. O resultado da execução do comando ROLLBACK TO ponto_marcado é que ele desfaz apenas o segundo INSERT (tipo de filme “catalogo”), já que o SAVEPOINT criado só aparece após a primeira inclusão na tabela.

Podemos visualizar isto através de uma consulta simples na tabela tipo_filme, conforme a **Figura 4**.

Saída de Dados | Explain | Mensagens | Histórico

	idtipo integer	descricao character var	qtd_dias_locacao integer	vlr_multa_dia numeric(9,2)
1	1	LANCAMENTO	1	2.00

Figura 4. Consulta na tabela tipo_filme.

Em seguida, o comando executado é o COMMIT que irá persistir as modificações no banco de dados.

Uma vez que podemos criar “savepoints”, também podemos liberá-los. Destruir um ponto de salvamento (SAVEPOINT) faz com que este não fique mais disponível como ponto para desfazer (rollback), mas não ocasiona nenhum outro comportamento visível pelo usuário.

Através da **Listagem 14** podemos visualizar a destruição de um ponto de salvamento.

Listagem 14. Comando de destruição de um SAVEPOINT

```
1 BEGIN;
2 INSERT INTO filme VALUES (1, 'ERA DO GELO 3', 1);
3 SAVEPOINT ponto_de_salvamento;
4 INSERT INTO filme VALUES (2, 'EFEITO BORBOLETA', 1);
5 RELEASE SAVEPOINT ponto_de_salvamento;
6 COMMIT;
```

Com a transação iniciada na **Listagem 14**, foi inserido um filme (“Era do gelo 3”) e após isso, foi criado um SAVEPOINT com o nome ponto_de_salvamento.

Em seguida, foi inserido um outro filme (“Efeito borboleta”), e após esta inclusão foi utilizado o comando RELEASE SAVEPOINT ponto_de_salvamento, que, por sua vez,

cadastrados com sucesso.

Figura 5. Consulta na tabela filme.

O comando RELEASE SAVEPOINT é utilizado para indicar que o aplicativo não deseja mais que o ponto de salvamento especificado seja mantido. Após este comando ser submetido, não é mais possível desfazer até o ponto de salvamento. Se for tentado usar um ponto de salvamento já liberado, ocorrerá um erro durante a execução do comando, de forma que toda a transação será abortada.

A **Listagem 15** demonstra a destruição de um SAVEPOINT e em seguida tenta-se retornar (usando ROLLBACK TO) até aquele SAVEPOINT que já foi liberado.

Listagem 15. Transação utilizando o comando RELEASE SAVEPOINT

```
1 BEGIN;
2 INSERT INTO filme VALUES (3, 'A DERIVA', 1);
3 SAVEPOINT ponto_de_salvamento;
4 INSERT INTO filme VALUES (4, 'TRANSFORMERS', 1);
5 RELEASE SAVEPOINT ponto_de_salvamento;
6 ROLLBACK TO ponto_de_salvamento;
7 COMMIT;
```

Após a execução da transação da **Listagem 15**, podemos verificar na **Figura 6** que ao

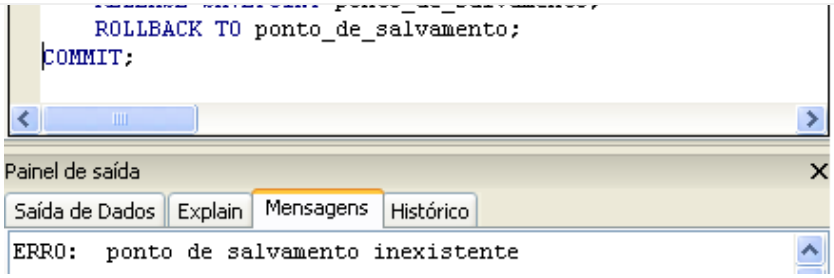

Figura 6. Execução da transação da **Listagem 13**.

Podemos ainda verificar, através da **Figura 7**, que a transação além de ter causado um erro, foi desfeita por completo, desfazendo os dois INSERTs e deixando a tabela filme apenas com os registros que foram inseridos antes da execução da referida transação.

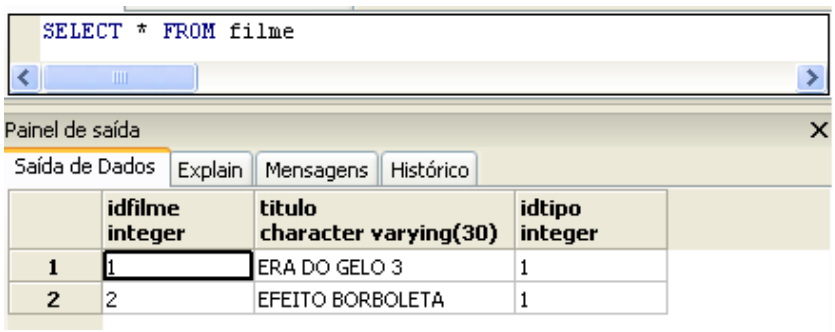

Figura 7. Consulta na tabela filme.

Conclusão

Existem vários tipos de aplicações que têm a necessidade de um processamento concorrente, e o uso de transações faz com consigamos atingir os objetivos das regras ACID.

Como foi visto, o PostgreSQL possui um bom controle de transações, além de possuir

Referencias

GONZAGA, J. L. Dominando o PostgreSQL. Rio d Janeiro: Editora Ciência ModernaLtda., 2007. p. 187-204.

ELMASRI, Ramez E.; NAVATHE, Shamkant. Sistemas de Banco de Dados. 4 ed.São Paulo: Pearson Addsison Wesley, 2005. p. 544-552.

DAMAS, Luis. SQL – Structered Query Language. 6 ed. São Paulo: LTC, 2007.396p.

SILBERSCHATZ, Abraham; KORTH, Henry F. Sistema de Banco de Dados. 5 ed.Rio de Janeiro: Elsevier, 2006. p. 409-415.

PostgreSQL

OLIVEIRA, Álvaro Mendes de. Curso de PostgreSQL – Unidade VIII – Controle de Transações.

Por **Willamys**
Em 2009