

11) Entendendo a execução das transações no PostgreSQL

Uso de Transações no PostgreSQL

Transação é cada execução de comando SQL que realiza leitura e ou escrita em bancos de dados. O PostgreSQL implementa transações resguardando as características ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

Atomicidade – uma transação é totalmente executada ou totalmente revertida sem deixar efeitos no banco de dados.

Consistência – os resultados são coerentes com as operações realizadas.

Isolamento – a execução de uma transação não interfere nem sofre interferência das demais transações em execução.

Durabilidade – o resultado das transações deve ser persistido fisicamente no banco de dados.

No PostgreSQL todo comando que executamos é internamente executado em transações, que gerencia a manutenção das características ACID.

Mas mesmo que o PostgreSQL aja dessa forma podemos querer agrupar alguns comandos em uma única transação como outras vezes podemos desabilitar o gerenciamento do PostgreSQL.

Transações no PostgreSQL

Na maioria das vezes, ao executar comandos no SGBD, não percebemos que transações são executadas em segundo plano, mas algumas vezes é útil iniciar (begin) e finalizar (commit) transações manualmente.

```
BEGIN  
update ...  
update ...  
END
```

Caso aconteça algum erro o PostgreSQL interrompe a transação com um ROLLBACK. Assim como também podemos executar um ROLLBACK manualmente para interromper a transação.

```
BEGIN  
update ...  
ROLLBACK  
update ...  
END
```

Um comando importante ao trabalhar com transações é o comando SET. Ele é usado para definir o nível de isolamento das transações.

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

De acordo com o SQL ANSI/ISO padrão, quatro níveis de isolamento de transações são definidos:

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

O PostgreSQL suporta o read committed e serializable.

Transações

Transação é um conceito fundamental de todo sistema de banco de dados. O ponto essencial da transação é englobar vários passos em uma única operação de tudo ou nada. Os estados intermediários entre os passos não são vistos pelas demais transações simultâneas e, se ocorrer alguma falha que impeça a transação chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de forma alguma.

Por exemplo, considere um banco de dados de uma instituição financeira contendo o saldo da conta corrente de vários clientes, assim como o saldo total dos depósitos de cada agência. Suponha que se deseje transferir \$100.00 da conta da Alice para a conta do Bob. Simplificando ao extremo, os comandos SQL para esta operação seriam:

```
UPDATE conta_corrente SET saldo = saldo - 100.00
  WHERE nome = 'Alice';
UPDATE filiais SET saldo = saldo - 100.00
  WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Alice');
UPDATE conta_corrente SET saldo = saldo + 100.00
  WHERE nome = 'Bob';
UPDATE filiais SET saldo = saldo + 100.00
  WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Bob');
```

Os detalhes destes comandos não são importantes aqui; o ponto importante é o fato de existirem várias atualizações distintas envolvidas para realizar esta operação tão simples. A contabilidade do banco quer ter certeza que todas estas atualizações foram realizadas, ou que nenhuma delas foi realizada. Com certeza não é interessante que uma falha no sistema faça com que Bob receba \$100.00 que não foi debitado da Alice. Além disso, Alice não continuará sendo uma cliente satisfeita se o dinheiro for debitado de sua conta e não for creditado na conta do Bob. É necessário garantir que, caso aconteça algo errado no meio da operação, nenhum dos passos executados até este ponto irá valer. Agrupar as atualizações em uma *transação* dá esta garantia. Uma transação é dita como sendo *atômica*: do ponto de vista das outras transações, ou a transação acontece por inteiro, ou nada acontece.

Desejamos, também, ter a garantia de estando a transação completa e aceita pelo sistema de banco de dados que a mesma fique definitivamente gravada, e não seja perdida mesmo no caso de acontecer uma pane logo em seguida. Por exemplo, se estiver sendo registrado um saque em dinheiro pelo Bob não se deseja, de forma alguma, que o débito em sua conta corrente desapareça por causa de uma pane ocorrida logo depois do Bob sair da agência. Um banco de dados transacional garante que todas as atualizações realizadas por uma transação ficam registradas em meio de armazenamento permanente (ou seja, em disco), antes da transação ser considerada completa.

Outra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando ao mesmo tempo, nenhuma delas deve enxergar as alterações incompletas efetuadas pelas outras. Por exemplo, se uma transação está ocupada totalizando o saldo de todas as agências, não pode ser visto o débito efetuado na agência da Alice mas ainda não creditado na agência do Bob, nem o contrário. Portanto, as transações devem ser tudo ou nada não apenas em termos do efeito permanente no banco de dados, mas também em termos de visibilidade durante o processamento. As atualizações feitas por uma transação em

andamento não podem ser vistas pelas outras transações enquanto não terminar, quando todas as atualizações se tornam visíveis ao mesmo tempo.

No PostgreSQL a transação é definida envolvendo os comandos SQL da transação pelos comandos [BEGIN](#) e [COMMIT](#). Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;  
UPDATE conta_corrente SET saldo = saldo - 100.00  
    WHERE nome = 'Alice';  
-- etc etc  
COMMIT;
```

Se no meio da transação for decidido que esta não deve ser efetivada (talvez porque tenha sido visto que o saldo da Alice ficou negativo), pode ser usado o comando [ROLLBACK](#) em vez do [COMMIT](#) para fazer com que todas as atualizações sejam canceladas.

O PostgreSQL, na verdade, trata todo comando SQL como sendo executado dentro de uma transação. Se não for emitido o comando BEGIN, então cada comando possuirá um BEGIN implícito e, se der tudo certo, um COMMIT, envolvendo-o. Um grupo de comandos envolvidos por um BEGIN e um COMMIT é algumas vezes chamado de *bloco de transação*.

Nota: Algumas bibliotecas cliente emitem um comando BEGIN e um comando COMMIT automaticamente, fazendo com que se obtenha o efeito de um bloco de transação sem perguntar se isto é desejado. Verifique a documentação da interface utilizada.

É possível controlar os comandos na transação de uma forma mais granular utilizando os *pontos de salvamento* (*savepoints*). Os pontos de salvamento permitem descartar partes da transação seletivamente, e efetivar as demais partes. Após definir o ponto de salvamento através da instrução SAVEPOINT, é possível cancelar a transação até o ponto de salvamento, se for necessário, usando ROLLBACK TO. Todas as alterações no banco de dados efetuadas pela transação entre o estabelecimento do ponto de salvamento e o cancelamento são descartadas, mas as alterações efetuadas antes do ponto de salvamento são mantidas.

Após cancelar até o ponto de salvamento, este ponto de salvamento continua definido e, portanto, é possível cancelar várias vezes. Ao contrário, havendo certeza que não vai ser mais necessário cancelar até o ponto de salvamento, o ponto de salvamento poderá ser liberado, para que o sistema possa liberar alguns recursos. Deve-se ter em mente que liberar ou cancelar até um ponto de salvamento libera, automaticamente, todos os pontos de salvamento definidos após o mesmo.

Tudo isto acontece dentro do bloco de transação e, portanto, nada disso é visto pelas outras sessões do banco de dados. Quando o bloco de transação é efetivado, as ações efetivadas se tornam visíveis como uma unidade para as outras sessões, enquanto as ações desfeitas nunca se tornam visíveis.

Recordando o banco de dados da instituição financeira, suponha que tivesse sido debitado \$100.00 da conta da Alice e creditado na conta do Bob, e descoberto em seguida que era para ser creditado na conta do Wally. Isso poderia ser feito utilizando um ponto de salvamento, conforme mostrado abaixo:

```
BEGIN;  
UPDATE conta_corrente SET saldo = saldo - 100.00  
    WHERE nome = 'Alice';  
SAVEPOINT meu_ponto_de_salvamento;  
UPDATE conta_corrente SET saldo = saldo + 100.00  
    WHERE nome = 'Bob';
```

```
-- uai ... o certo é na conta do Wally
ROLLBACK TO meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Wally';
COMMIT;
```

Obviamente este exemplo está simplificado ao extremo, mas é possível efetuar um grau elevado de controle sobre a transação através do uso de pontos de salvamento. Além disso, a instrução `ROLLBACK TO` é a única forma de obter novamente o controle sobre um bloco de transação colocado no estado interrompido pelo sistema devido a um erro, fora cancelar completamente e começar tudo de novo.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/tutorial-transactions.html> e em: <http://www.postgresql.org/docs/8.3/static/tutorial-transactions.html>

Isolamento da transação

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações simultâneas. Os fenômenos não desejados são:

dirty read (leitura suja)

A transação lê dados escritos por uma transação simultânea não efetivada (*uncommitted*). [1]

nonrepeatable read (leitura que não pode ser repetida)

A transação lê novamente dados lidos anteriormente, e descobre que os dados foram alterados por outra transação (que os efetivou após ter sido feita a leitura anterior). [2]

phantom read (leitura fantasma)

A transação executa uma segunda vez uma consulta que retorna um conjunto de linhas que satisfazem uma determinada condição de procura, e descobre que o conjunto de linhas que satisfazem a condição é diferente por causa de uma outra transação efetivada recentemente. [3]

Os quatro níveis de isolamento de transação, e seus comportamentos correspondentes, estão descritos na [Tabela 12-1](#).

Tabela 12-1. Níveis de isolamento da transação no SQL

Nível de isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	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 estrito do que o selecionado. Isto é permitido pelo padrão SQL: os quatro níveis de isolamento somente definem quais fenômenos não podem acontecer, não definem quais fenômenos devem acontecer. O motivo pelo qual o PostgreSQL só disponibiliza dois níveis de isolamento, é porque esta é a única forma de mapear os níveis de isolamento padrão na arquitetura de controle de simultaneidade multiversão que faz sentido. O comportamento dos níveis de isolamento disponíveis estão detalhados nas próximas subseções.

É utilizado o comando [SET TRANSACTION](#) para definir o nível de isolamento da transação.

12.2.1. Nível de isolamento Read Committed

O *Read Committed* (lê efetivado) é o nível de isolamento padrão do PostgreSQL. Quando uma transação processa sob este nível de isolamento, o comando SELECT enxerga apenas os dados efetivados antes da consulta começar; nunca enxerga dados não efetivados, ou as alterações efetivadas pelas transações simultâneas durante a 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 destes comandos de atualização nas outras linhas do banco de dados. Este comportamento torna o *Read Committed* inadequado para os comandos envolvendo condições de procura complexas. Entretanto, é apropriado para casos mais simples. Por exemplo, considere a atualização do saldo bancário pela transação mostrada abaixo:

```
BEGIN;  
UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 12345;
```

```
UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 7534;  
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*.

12.2.2. Nível de isolamento serializável

O nível *Serializable* fornece o isolamento de transação mais rigoroso. Este nível emula a execução serial das transações, como se todas as transações fossem executadas uma 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.

Quando o aplicativo receber esta mensagem de erro deverá interromper a transação corrente, e tentar executar novamente toda a transação a partir do início. Da segunda 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.

12.2.2.1. Isolamento serializável versus verdadeira serialidade

O significado intuitivo (e a definição matemática) de execução "serializável" é que quaisquer duas transações simultâneas efetivadas com sucesso parecem ter sido executadas de forma rigorosamente serial, uma após a outra — embora qual das duas parece ter ocorrido primeiro não pode ser previsto antecipadamente. É importante ter em mente que proibir os comportamentos indesejáveis listados na [Tabela 12-1](#) não é suficiente para garantir a verdadeira serialidade e, de fato, o modo serializável do PostgreSQL *não garante a execução serializável neste sentido*. Como exemplo será considerada a tabela `minha_tabela` contendo inicialmente

classe	valor
1	10
1	20
2	100
2	200

Suponha que a transação serializável A calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 1;
```

e insira o resultado (30) como valor em uma nova linha com classe = 2. Simultaneamente a transação serializável B calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 2;
```

e obtém o resultado 300, que é inserido em uma nova linha com classe = 1. Em seguida as duas transações efetivam. Nenhum dos comportamentos não desejados ocorreu, ainda assim foi obtido um resultado que não poderia ter ocorrido serialmente em qualquer ordem. Se A tivesse executado antes de B, então B teria calculado a soma como 330, e não 300, e de maneira semelhante a outra ordem teria produzido uma soma diferente na transação A.

Para garantir serialidade matemática verdadeira, é necessário que o sistema de banco de dados imponha o *bloqueio de predicado*, significando que a transação não pode inserir ou alterar uma linha que corresponde à condição WHERE de um comando de outra transação simultânea. Por exemplo, uma vez que a transação A tenha executado o comando `SELECT ... WHERE class = 1`, o sistema de bloqueio de predicado proibiria a transação B inserir qualquer linha com classe igual a 1

até que A fosse efetivada. [4] Um sistema de bloqueio deste tipo é de implementação complexa e de execução extremamente dispendiosa, uma vez que todas as sessões devem estar cientes dos detalhes de todos os comandos executados por todas as transações simultâneas. E este grande gasto em sua maior parte seria desperdiçado, porque na prática a maioria dos aplicativos não fazem coisas do tipo que podem ocasionar problemas (Certamente o exemplo acima é bastante irreal, dificilmente representando um programa de verdade). Portanto, o PostgreSQL não implementa o bloqueio de predicado, e tanto quanto saibamos nenhum outro SGBD de produção o faz.

Nos casos em que a possibilidade de execução não serial representa um perigo real, os problemas podem ser evitados através da utilização apropriada de bloqueios explícitos. São mostrados mais detalhes nas próximas seções.

Notas

- [1] *dirty read* — A transação SQL T1 altera uma linha. Em seguida a transação SQL T2 lê esta linha antes de T1 executar o comando COMMIT. Se depois T1 executar o comando ROLLBACK, T2 terá lido uma linha que nunca foi efetivada e que, portanto, pode ser considerada como nunca tendo existido. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [2] *nonrepeatable read* — A transação SQL T1 lê uma linha. Em seguida a transação SQL T2 altera ou exclui esta linha e executa o comando COMMIT. Se T1 tentar ler esta linha novamente, pode receber o valor alterado ou descobrir que a linha foi excluída. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [3] *phantom read* — A transação SQL T1 lê um conjunto de linhas N que satisfazem a uma condição de procura. Em seguida a transação SQL T2 executa comandos SQL que geram uma ou mais linhas que satisfazem a condição de procura usada pela transação T1. Se depois a transação SQL T1 repetir a leitura inicial com a mesma condição de procura, será obtida uma coleção diferente de linhas. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [4] Em essência, o sistema de bloqueio de predicado evita leituras fantasmas restringindo o que é escrito, enquanto o MVCC evita restringindo o que é lido.

Fontes: <http://pgdocptbr.sourceforge.net/pg80/transaction-iso.html> e em: <http://www.postgresql.org/docs/8.3/static/transaction-iso.html>

SET TRANSACTION

Nome

SET TRANSACTION -- define as características da transação corrente

Sinopse

```
SET TRANSACTION modo_da_transação [, ...]  
SET SESSION CHARACTERISTICS AS TRANSACTION modo_da_transação [, ...]
```

onde modo_da_transação é um entre:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ  
UNCOMMITTED }  
READ WRITE | READ ONLY
```

Descrição

O comando SET TRANSACTION define as características da transação corrente. Não produz nenhum efeito nas próximas transações. O comando SET SESSION CHARACTERISTICS define as características da transação usadas como padrão nas próximas transações na sessão. Estes padrões podem ser mudados para uma transação individual pelo comando SET TRANSACTION.

[\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#)

As características da transação disponíveis são o nível de isolamento da transação e o modo de acesso da transação (leitura/escrita ou somente para leitura).

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

READ COMMITTED

O comando consegue ver apenas as linhas efetivadas (*commit*) antes do início da sua execução. Este é o padrão.

SERIALIZABLE

Todos os comandos da transação corrente podem ver apenas as linhas efetivadas antes da primeira consulta ou comando de modificação de dados ter sido executado nesta transação.

O padrão SQL define dois níveis adicionais, READ UNCOMMITTED e REPEATABLE READ. No PostgreSQL READ UNCOMMITTED é tratado como READ COMMITTED, enquanto REPEATABLE READ é tratado como SERIALIZABLE.

O nível de isolamento da transação não pode ser mudado após a primeira consulta ou comando de modificação de dado (SELECT, INSERT, DELETE, UPDATE, FETCH ou COPY) da transação ter sido executado. Para obter informações adicionais sobre o isolamento de transações e controle de simultaneidade deve ser consultado o [Capítulo 12](#).

O modo de acesso da transação determina se a transação é para leitura/escrita, ou se é somente para

leitura. Ler/escrever é o padrão. Quando a transação é somente para leitura, não são permitidos os seguintes comandos SQL: INSERT, UPDATE, DELETE e COPY FROM, se a tabela a ser escrita não for uma tabela temporária; todos os comandos CREATE, ALTER e DROP; COMMENT, GRANT, REVOKE, TRUNCATE; também EXPLAIN ANALYZE e EXECUTE se o comando a ser executado estiver entre os listados. Esta é uma noção de somente para leitura de alto nível, que não impede todas as escritas em disco.

Observações

Se for executado o comando SET TRANSACTION sem ser executado antes o comando START TRANSACTION ou BEGIN, parecerá que não produziu nenhum efeito, uma vez que a transação termina imediatamente.

É possível não utilizar o comando SET TRANSACTION, especificando o modo_da_transação desejado no comando BEGIN ou no comando START TRANSACTION.

Os modos de transação padrão da sessão também podem ser definidos através dos parâmetros de configuração [default_transaction_isolation](#) e [default_transaction_read_only](#) (De fato, SET SESSION CHARACTERISTICS é apenas uma forma verbosa equivalente a definir estas variáveis através do comando SET). Isto significa que os valores padrão podem ser definidos no arquivo de configuração, via ALTER DATABASE, etc. Para obter informações adicionais deve ser consultada a [Seção 16.4](#).

Compatibilidade

Os dois comandos estão definidos no padrão SQL. No padrão SQL SERIALIZABLE é o nível de isolamento padrão da transação; no PostgreSQL normalmente o padrão é READ COMMITTED, mas pode ser mudado conforme mencionado acima. Devido à falta de bloqueio de predicado, o nível SERIALIZABLE não é verdadeiramente serializável. Para obter mais informações deve ser consultada a [Capítulo 12](#).

No padrão SQL existe uma outra característica de transação que pode ser definida por estes comandos: o tamanho da área de diagnósticos. Este conceito é específico da linguagem SQL incorporada e, portanto, não é implementado no servidor PostgreSQL.

O padrão SQL requer a presença de vírgulas entre os modo_da_transação sucessivos, mas por razões históricas o PostgreSQL permite que estas vírgulas sejam omitidas.

Notas

- [1] Oracle — O comando SET TRANSACTION é utilizado para estabelecer a transação corrente como apenas de leitura ou de leitura e escrita, estabelecer o *nível de isolamento*, ou atribuir a sessão para um segmento de *rollback* especificado. As operações realizadas pelo comando SET TRANSACTION afetam apenas a transação corrente, não afetando outros usuários ou outras transações. A transação termina quando é executado o comando COMMIT ou o comando ROLLBACK. O banco de dados Oracle efetiva implicitamente a transação antes e após a execução de um comando da linguagem de definição de dados (DDL). A cláusula ISOLATION LEVEL especifica como as transações que contêm modificações no banco de dados são tratadas. A definição SERIALIZABLE especifica o modo de isolamento da transação como serializável, conforme definido no padrão SQL92. Se a transação serializável contiver

comando da linguagem de manipulação de dados (DML) que tenta atualizar qualquer recurso que possa ter sido atualizado por uma transação não efetivada no início da transação serializável, então o comando da DML falhará. A definição READ COMMITTED é o comportamento padrão de transação do banco de dados Oracle. Se a transação contiver comandos da DML requerendo bloqueios de linha mantidos por outras transações, então o comando da DML aguardará até os bloqueios de linha serem liberados. [Oracle® Database SQL Reference 10g Release 1 \(10.1\) Part Number B10759-01](#) (N. do T.)

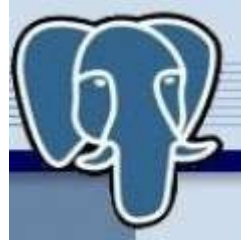
- [2] SQL Server — O comando SET TRANSACTION ISOLATION LEVEL controla o comportamento de bloqueio e versão da linha dos comandos Transact-SQL emitidos por uma conexão com o SQL Server. Os níveis de isolamento são: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SNAPSHOT e SERIALIZABLE. [SQL Server 2005 Books Online — SET TRANSACTION ISOLATION LEVEL \(Transact-SQL\)](#) (N. do T.)
- [3] DB2 — O comando SET CURRENT ISOLATION atribui um valor ao registrador especial CURRENT ISOLATION. Este comando não está sob controle da transação. [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)
- [4] DB2 — O comando CHANGE ISOLATION LEVEL muda a maneira do DB2 isolar os dados de outros processos enquanto o banco de dados está sendo acessado. Os níveis de isolamento são: CS (cursor stability); NC (no commit). Não suportado pelo DB2; RR (repeatable read); RS (read stability); UR (uncommitted read). [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)

Fontes: <http://pgdocptbr.sourceforge.net/pg80/sql-set-transaction.html> e em: <http://www.postgresql.org/docs/8.3/static/sql-set-transaction.html>

Transações com o PostgreSQL - Parte I

De Juliano Ignácio

Este será um artigo dividido em três partes. Trata-se de algo muito importante, um dos principais motivos pelo grande sucesso dos gerenciadores de banco de dados (SGBD/DBMS).



O gerenciamento das transações são feitos pelo PostgreSQL, no entanto, existem comandos, instruções e configurações que definem o seu comportamento. É exatamente isto que veremos neste conjunto de artigos.

Você irá ver como o PostgreSQL permite juntar diversas alterações a serem efetuadas na base de dados como uma unidade única de trabalho, ou seja, quando você tem um conjunto de mudanças que devem ser todas executadas ou, se ocorrer algum problema, não será feito nada.

O QUE SÃO AS TRANSAÇÕES

Antes de mais nada precisamos saber como os dados são atualizados no PostgreSQL. Geralmente quando mostramos exemplos de alterações de dados em uma base, usamos uma única declaração para que esta alteração seja feita. Porém, no mundo real, encontramos diversas situações onde precisamos fazer diversas mudanças às quais seriam impossíveis de serem efetuadas em uma única declaração. Você ainda precisa que todas estas alterações obtenham sucesso, ou nenhuma delas deve ser efetuada, caso surja algum problema em qualquer ponto deste grupo de mudanças.




Um exemplo clássico é a transferência de dinheiro entre duas contas bancárias, podendo ser representadas até mesmo em tabelas distintas, e precisamos que uma conta seja debitada e a outra, creditada. Se debitarmos um valor de uma conta e ocorrer algum problema ao tentarmos creditar este valor, qualquer que seja o motivo, precisamos retornar o valor à primeira conta, ou sendo sendo mais coerente, este valor nunca chegou a ser debitado da primeira conta. Nenhum banco poderia manter-se no negócio caso eventualmente "perdesse" o dinheiro durante uma transação financeira entre duas contas (ou qualquer outra, logicamente).



Em bancos de dados relacionais baseados no ANSI SQL, como o PostgreSQL, isto é possível graças ao recurso de nome **TRANSAÇÃO**.

Uma transação é uma unidade lógica de trabalho que não deve ser subdividida.

Mas o que sabemos sobre unidade lógica de trabalho? Isto é simplesmente um conjunto de mudanças a serem efetuadas na base de dados onde todas devem ter sucesso, ou todas devem falhar. Exatamente como a transferência de dinheiro entre contas bancárias como mencionado acima. No PostgreSQL, estas mudanças são controladas por três instruções:

	BEGIN WORK declara o início de uma transação;
	COMMIT WORK indica que todos os elementos da transação foram executados com sucesso e podem agora serem persistidos e acessados por todas as demais transações concorrentes ou subsequentes;
	ROLLBACK WORK indica que a transação será abandonada e todas as mudanças feitas nos dados pelas instruções em SQL serão canceladas. O banco de dados se apresentará aos seus usuários como se nenhuma mudança tivesse ocorrido desde a instrução BEGIN WORK;

OBS: Tanto **COMMIT WORK** e **ROLLBACK WORK** podem omitir a palavra **WORK**, mas, em **BEGIN WORK** é obrigatório.

A padronização ANSI / ISO SQL não define a instrução BEGIN WORK, define que as transações iniciam-se automaticamente, mas esta é uma extensão muito comum presente - e requerida - em muitos bancos de dados relacionais.

Um segundo aspecto sobre as transações é que qualquer transação no banco de dados é isolada de outras transações que estão ocorrendo ao mesmo tempo. De uma forma ideal, cada transação se comportaria como se tivesse acesso exclusivo ao banco de dados. Infelizmente, como veremos nos próximos artigos, existem níveis de isolamento que, na prática, para atingir melhores performances algo será comprometido.

ACID : ÁCIDO, AGRE, AZEDO,... ?!!!

Não.. não tem nada haver com culinária...

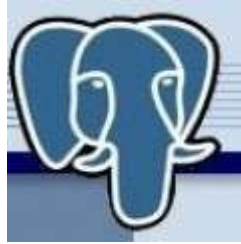
ACID é um mnemônico (uma sigla) que descreve as propriedades que uma transação deve possuir.

Atomic Atômica	Uma transação, mesmo sendo um conjunto de ações, deve ser executada como uma unidade única. Uma transação deve ser executada exatamente uma única vez, sem nenhuma dependência. Em nosso exemplo bancário, a movimentação financeira deve ser atômica. O débito em uma conta e o crédito em outra devem ambos acontecer como sendo uma única ação, mesmo se diversas instruções consecutivas em SQL fossem requeridas.
Consistent Consistente	Ao término de uma transação, o sistema deve ser deixado em um estado consistente, respeitando as integridades de dados e relacional da base sendo manipulada. No exemplo da movimentação bancária, ao final da transação, ambas as contas devem ter sido atualizadas com os montantes corretos.
Isolated Isolada	Isto significa que cada transação, não importando quantas transações estão sendo executadas neste momento no banco de dados, deve parecer ser independente de todas as outras transações. Imagine caixas eletrônicos separados, onde, diferentes pessoas querem executar a mesma operação bancária para a mesma conta. Transações processando duas ações concorrentes devem comportar-se como se cada uma delas estivessem operando com acesso exclusivo à base de dados. Na prática, nós sabemos que as coisas não são tão simples assim. Nós trataremos deste tópico nos próximos artigos.
Durable Durável	Uma vez que a transação seja completada, ela deve permanecer completada. Uma vez que o dinheiro tenha sido transferido com sucesso entre as contas, ele deve permanecer transferido, mesmo que a máquina que está rodando o gerenciador de banco de dados pare por falta de energia elétrica. No PostgreSQL, assim como na maioria dos bancos de dados relacionais, isto é conseguido através de um arquivo que descreve as transações (<i>transaction logfile</i>). A maneira este arquivo funciona é bastante simples. Assim que uma transação é executada, as ações não são somente executadas na base de dados, mas também gravadas neste arquivo. Assim que a transação é completada, uma marca é gravada para dizer que a transação foi finalizada, e os dados gravados no arquivo <i>logfile</i> são forçados a permanecerem armazenados, então isto torna o banco de dados seguro mesmo que ele "caia". A durabilidade da transação acontece sem a intervenção do usuário, ou seja, é uma operação executada pelo próprio banco de dados.

Transações com o PostgreSQL - Parte II

Esta é a segunda parte da série de artigos que está dividida em três momentos.

Vimos no artigo anterior o que são as transações, quais instruções são necessárias, e quais as propriedades que uma transação deve possuir.

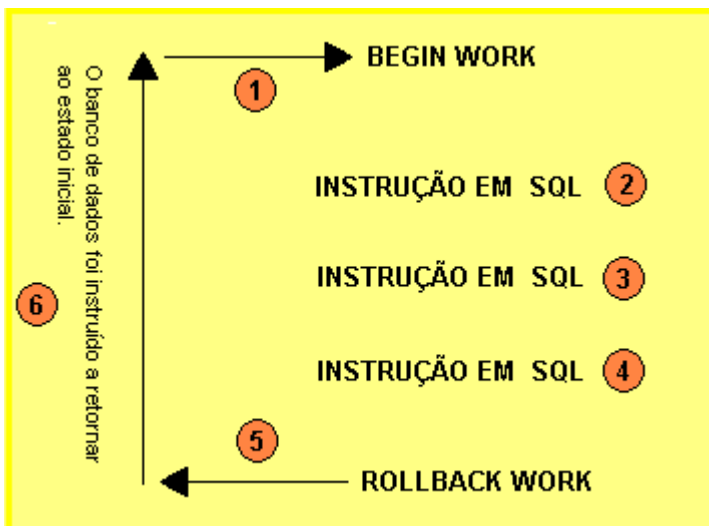


TRANSAÇÕES DE UM ÚNICO USUÁRIO

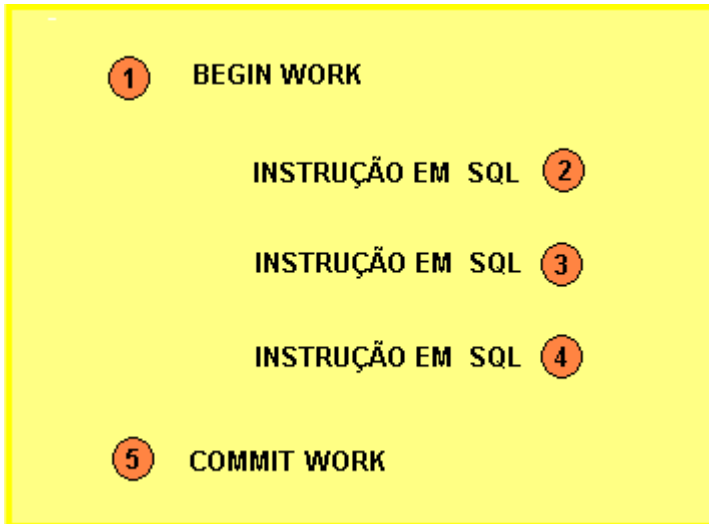
Antes de nós olharmos aspectos mais complexos das transações e como se comportam dentro da concorrência dos usuários do banco de dados, nós precisamos saber primeiro como se comporta com um único usuário.

Mesmo dessa maneira simplista de trabalhar, há vantagens reais em usar o controle de transações. O maior benefício do controle da transação é que permite que sejam executadas diversas instruções SQL, e então, ao final do processo, permitir que você possa desfazer o trabalho que já fez, caso queira isto. Usando uma transação, a aplicação não necessita se preocupar em armazenar quais alterações foram sendo feitas ao banco de dados para desfazê-las então, se necessário. Você simplesmente pede ao sistema gerenciador de banco de dados para desfazer todo o processo de uma só vez.

A seqüência se apresenta da seguinte maneira:



Se você decidir que todas as alterações no banco de dados são válidas até o passo 5, e mesmo assim, você deseja aplicá-las ao banco de dados para então se tornarem permanentes, então, a única coisa que você tem que fazer é substituir a instrução **ROLLBACK WORK** pela instrução **COMMIT WORK**:



Após o passo 5, as mudanças ocorridas no banco de dados são COMETIDAS (escritas, confirmadas), e podem ser consideradas permanentes, dessa maneira não serão perdidas caso haja falta de energia elétrica, problemas no disco ou erros na aplicação.

REPETINDO

Uma transação é uma unidade lógica de trabalho que não deve ser subdividida.

Abaixo está um exemplo bem simples, onde é alterado uma única linha em uma tabela de cadastro chamada `tbl_usuario`, trocando o conteúdo da coluna nome de 'Juliano' por 'Mauricio', e então usando a instrução **ROLLBACK WORK** para cancelar a alteração:

```
USANDO o ROLLBACK WORK
postgres=# BEGIN WORK;
BEGIN
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)

postgres=# UPDATE tbl_usuario
postgres=# SET nome='Mauricio'
postgres=# WHERE usuid=1;
UPDATE 1
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)

postgres=# ROLLBACK WORK;
ROLLBACK
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)
postgres=#
```

```

USANDO O COMMIT WORK
postgres=# BEGIN WORK;
BEGIN
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)

postgres=# UPDATE tbl_usuario
postgres=# SET nome='Mauricio'
postgres=# WHERE usuid=1;
UPDATE 1
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)

postgres=# COMMIT WORK;
ROLLBACK
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)
postgres=#

```

LEMBRE-SE: Tanto **COMMIT WORK** e **ROLLBACK WORK** podem omitir a palavra **WORK**, mas, em **BEGIN WORK** é obrigatório.

LIMITAÇÕES ...mas não por muito tempo.

Há alguns pontos a serem considerados quando estamos falando de transações:

Primeiramente, o PostgreSQL ainda não trabalha com transações aninhadas, este é um recurso muito esperado que está sendo inserido na versão 7.4 (que já está em fase de beta testes). Hoje no PostgreSQL, se você tentar executar uma instrução **BEGIN WORK** enquanto ainda se encontra em uma transação, o PostgreSQL irá ignorar este último comando e emitirá a seguinte mensagem:

WARNING: BEGIN: already a transaction in progress

Assim que o PostgreSQL puder trabalhar com transações aninhadas, o conceito de **SAVE POINTS** estará disponível e será possível marcar mais de um ponto dentro de uma transação principal e, retornar (**ROLLBACK WORK**) até um determinado ponto (**SAVE POINT**) ao invés de retornar toda a transação.

Justamente por não trabalhar com transações aninhadas, é de boa prática manter as transações no menor tamanho possível (e viável). Tanto o PostgreSQL quanto qualquer outro banco de dados relacional, tem que executar muito trabalho para assegurar que transações de diferentes usuários devem ser executadas separadamente. Uma consequência disto é que partes do banco de dados envolvidas numa transação, freqüentemente precisam se tornar parcialmente inacessível (trancadas), para garantir que as transações sejam mantidas separadas.

Embora o PostgreSQL tranque o banco de dados automaticamente nestas ocasiões, uma transação de longa duração geralmente impossibilita que outros usuários acessem os dados envolvidos nesta transação, até que a mesma seja completada ou cancelada. Imagine que uma aplicação iniciou uma transação assim que a pessoa chegou cedo ao escritório, sentou para trabalhar, e deixou uma transação rodando o dia inteiro enquanto executava diversas alterações no banco de dados. Supondo que esta pessoa "COMITOU" a transação somente no final do expediente, a performance do banco de dados, e a capacidade de outros usuários acessarem os dados desta enorme transação foram severamente impactados.

PODE PARECER ÓBVIO: ...mas não deixe uma transação aberta (iniciada) e saia pra tomar um café, ou mesmo falar com alguém, isto pode comprometer seriamente o andamento dos trabalhos na empresa que necessitem do banco de dados para executá-los.

Uma instrução **COMMIT WORK** geralmente é executada muito rápido, uma vez que - geralmente - tem muito pouco trabalho a ser executado. Retornar transações no entanto, normalmente envolve muito mais trabalho para o banco de dados executar, pois, além dos trabalhos já executados até então dentro da transação, deve desfazer os mesmos. Ou seja, se você inicia uma transação que demora 3 minutos para ser executada e, ao final você decide executar um **ROLLBACK WORK** para retornar à situação inicial, então, não espere que seja executado instantaneamente. Isto poderá demorar facilmente mais do que 3 minutos para restaurar todas as alterações já executadas pela transação.

Transações com o PostgreSQL - Parte III

Esta é a terceira, porém, ainda não, a última parte deste artigo originalmente dividido em três partes. Haverá uma quarta parte, pois, após os vários e-mail que recebi com relação à este artigo, decidi refiná-lo um pouco mais, o que fez com que eu acrescentasse uma parte a mais.



Nos artigos anteriores, vimos como as transações precisam trabalhar em ambientes com múltiplos usuários de forma concorrente e como cada transação deve ser isolada uma das outras, nos importando então somente, verificar como se comporta uma única transação. Agora, iremos retornar às regras **ACID** e olhar mais de perto o que significa o **I** de **ACID**, Isolated (Isolado).

NÍVEIS DE ISOLAMENTO ANSI

Como já foi mencionado, um dos aspectos mais dificultosos dos bancos de dados relacionais é o isolamento entre diferentes usuários em relação às atualizações feitas na base. Claro que atingir o isolamento não é difícil. Simplesmente, permita somente uma única conexão para o banco de dados, com somente uma transação sendo processada por vez. Isto garantirá à você um isolamento completo entre as diferentes transações. A dificuldade aparece quando procuramos atingir um isolamento mais prático sem prejudicar significativamente a performance, prevenindo o acesso multi-usuário ao banco de dados.

O verdadeiro isolamento é extremamente difícil de ser atingido sem uma alta degradação de performance como consequência, o padrão ANSI / ISO SQL define diferentes níveis de isolamento que os bancos de dados podem implementar. Frequentemente, um banco de dados relacional irá

implementar pelo menos um destes níveis como default (padrão) e, normalmente, permitem aos usuários especificar pelo menos um outro nível de isolamento para ser utilizado.

Antes que possamos entender os padrões de níveis de isolamento, nós precisamos nos familiarizar com algumas outras terminologias. Embora o comportamento padrão do PostgreSQL se mostre suficiente na maioria dos casos, há circunstâncias onde seja útil entendê-lo em detalhes.

FENÔMENOS INDESEJÁVEIS

O padrão ANSI / ISO SQL define os níveis de isolamento em termos de fenômenos indesejáveis que podem acontecer em bancos de dados multi-usuários quando interagem com as transações.

Dirty Read (leitura suja)

O *Dirty Read* ocorre quando algumas instruções em SQL em uma transação lêem dados que foram alterados por uma outra transação, porém, a transação que alterou os dados ainda não completou (**COMMIT**) seu bloco de instruções.

Como já mostrado antes (de maneira insistente), uma transação é um bloco ou unidade lógica de trabalho que deve ser atômica. Mesmo que todos os elementos de uma transação ocorram ou nenhuma delas ocorra. Até que uma transação tenha sido completada (**COMMITed**), sempre haverá a possibilidade de que ela falhe, ou seja abandonada através de uma instrução **ROLLBACK WORK**. Por isso, nenhum outro usuário do banco de dados deveria ver estas mudanças antes da transação ser completada.

Para ilustrar isso, considerando que diferentes transações podem ser vistas através da coluna **nome** da tabela **tbl_usuario** que possui o valor 1 na coluna **usuid**. Assumindo que o *Dirty Read* está permitido, o que o PostgreSQL nunca permitirá, então só veremos o comportamento da transação como um exemplo teórico:

UM EXEMPLO DE <i>DIRT READ</i>			
Transação 1	Dados vistos pela Transação 1	O que o <i>Dirty Read</i> em outra transação veria	O que outras transações veriam se o <i>Dirty Read</i> não ocorresse
postgres=# BEGIN WORK; BEGIN	Juliano	Juliano	Juliano
postgres=# UPDATE tbl_usuario SET nome='Marcio' WHERE usuid=1; UPDATE 1	Marcio	Marcio	Juliano
postgres=# COMMIT WORK; COMMIT	Marcio	Marcio	Marcio
postgres=# BEGIN WORK; BEGIN	Marcio	Marcio	Marcio
postgres=# UPDATE tbl_usuario SET nome='Andrea' WHERE usuid=1; UPDATE 1	Andrea	Andrea	Marcio
postgres=# ROLLBACK WORK; ROLLBACK	Marcio	Marcio	Marcio

Note que uma leitura suja (*Dirty Read*) permite que outras transações possam ver os dados que ainda não foram completados (**COMMITed**). Isto significa que eles podem ver mudanças que podem ser descartadas em seguida, por causa de uma instrução **ROLLBACK WORK** por exemplo.

O PostgreSQL NÃO permite dirty read.

Unrepeatable Reads (leituras não podem ser repetidas)

Um *Unrepeatable Read* (leitura que não pode ser repetida) ocorre quando uma transação lê um conjunto de dados, depois mais tarde, re-lê os dados, e descobre que os mesmos mudaram. Isto é muito menos sério do que o *Dirt Read*. Vamos ver como isso se parece:

UM EXEMPLO DE UNREPEATABLE READ			
Transação 1	Dados vistos pela Transação 1	O que o <i>Unrepeatable Read</i> em outra transação veria	O que outras transações veriam se o <i>Unrepeatable Read</i> não ocorresse
postgres=# BEGIN WORK; BEGIN	Marcio	Marcio	Marcio
postgres=# UPDATE tbl_usuario SET nome='Mauricio' WHERE usuid=1; UPDATE 1	Mauricio	Marcio	Marcio
postgres=# COMMIT WORK; COMMIT	Mauricio	Mauricio	Marcio
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1; (1 row)	Mauricio	Mauricio	Mauricio

Note que o *Unrepeatable Read* significa que a transação pode ver as mudanças completadas (**COMMITadas**) por outras transações, mesmo que embora uma transação de leitura não tenha sido completada. Se as leituras que não podem ser repetidas forem prevenidas, então outras transações não poderão ver as alterações no banco de dados até que eles próprios sejam completados (**COMMIT**).

Por padrão, o PostgreSQL não permite *Unrepeatable Reads*, embora, como veremos adiante, nós poderemos alterar este comportamento padrão.

O PostgreSQL NÃO permite unrepeatable reads.

Phantom Reads (leituras fantasmas)

Este problema ocorre quando uma nova tupla (registro) aparece na tabela, enquanto uma transação diferente está atualizando a tabela, e a nova tupla deveria ter sido atualizada, mas não foi.

Suponha que temos duas transações atualizando a tabela `tbl_usuario`. A primeira está adicionando uma vírgula ao fim do nome, e a segunda adicionando um novo nome:

UM EXEMPLO DE PHANTOM READ	
Transação 1	Transação 2
postgres=# BEGIN WORK; BEGIN	postgres=# BEGIN WORK; BEGIN
postgres=# UPDATE tbl_usuario SET nome = nome ','; UPDATE 1	postgres=# INSERT INTO tbl_usuario (nome) VALUES ('Adriana'); COMMIT
postgres=# COMMIT WORK; COMMIT	postgres=# COMMIT WORK; COMMIT

concatenada.

Leituras fantasmas são muito raras, e quase impossíveis de demonstrar, no entanto, geralmente você não precisa se preocupar com isto, embora por padrão o PostgreSQL permitirá algum tipo de *Phantom Reads*.

Lost Updates (atualizações perdidas)

Atualizações perdidas é um pouco diferente dos três casos anteriores, que são geralmente problemas de nível de aplicação, não estando relacionado com a maneira como o banco de dados trabalha. Um *Lost Update* por outro lado, ocorre quando duas mudanças diferentes são escritas no banco de dados, e a segunda mudança causa a perda da primeira.

Suponha dois usuários que estão utilizando uma aplicação baseada em console, que está atualizando o cadastro de usuários:

UM EXEMPLO DE LOST UPDATES			
Usuário 1	Dados vistos pelo Usuário 2	Usuário 1	Dados vistos pelo Usuário 2
Tentando alterar o nome 'Marcio' para 'Maurice'		Tentando alterar o sobrenome 'Costa' para 'Fischer'	
postgres=# BEGIN WORK; BEGIN		postgres=# BEGIN WORK; BEGIN	
postgres=# SELECT nome, sobrenome FROM tbl_usuario WHERE usuid=1;	Marcio, Costa	postgres=# SELECT sobrenome, nome FROM tbl_usuario WHERE usuid=1;	Costa, Marcio
postgres=# UPDATE tbl_usuario SET nome='Maurice', sobrenome = 'Costa' WHERE usuid=1; UPDATE 1	Maurice, Costa		
postgres=# COMMIT WORK; COMMIT			Maurice, Costa
		postgres=# UPDATE tbl_usuario SET nome='Marcio', sobrenome='Fischer' WHERE usuid=1; UPDATE 1	
	Maurice, Costa	postgres=# COMMIT WORK; COMMIT	Marcio, Fischer
	arcio, Fischer		arcio, Fischer

O **nome** alterado pelo Usuário 1 é perdido, não por haver algum erro no banco de dados, mas porque o Usuário 2 leu o **nome**, então o manteve por alguns momentos, e gravou-o novamente no banco de dados, destruindo a mudança que o Usuário 1 tinha feito. O banco de dados está quase isolando corretamente os dois conjuntos de dados, mas a aplicação ainda está perdendo dados.

Há diversas maneiras de contornar este problema, e decidir qual a maneira mais apropriada irá depender de cada aplicação. Como um primeiro passo, as aplicações devem se atentar a manter as transações o mais curtas possíveis, nunca segurando-as em execução por muito mais tempo além do estritamente necessário. Como um segundo passo, as aplicações deveriam escrever de volta os dados que ele têm alterado.. Estes dois passos irá preveni-los de muitas ocorrências de perda de atualizações, incluindo o erro mostrado acima.

No próximo artigo: *Set Transaction Isolation Level, Auto Commit, Locking, Deadlocks, etc.*

Transações com o PostgreSQL - Parte IV

Agora sim! A quarta e última parte deste artigo. Até aqui nós vimos como as transações são úteis mesmo em banco de dados de um só usuário, permitindo-nos agrupar instruções SQL em uma unidade atômica, à qual pode tanto ser completada, quanto abandonada.



Também vimos como as transações trabalham em ambientes multi-usuários. Vimos os significados das regras de cada letra do acrônimo **ACID** e o que significam em termos de banco de dados. Também vimos termos padrões **ANSI** de "fenômenos indesejáveis".

NÍVEIS DE ISOLAMENTO ANSI / ISO

Usando a nossa mais recente terminologia, nós estamos agora na posição de entender o caminho definido pelo ANSI / ISO para diferentes níveis de isolamento que um banco de dados pode usar. Cada nível ANSI / ISO é uma combinação dos primeiros três tipos de comportamentos indesejáveis mostrados logo abaixo:

Definição do Nível de Isolamento ANSI / ISO	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	<i>Possible</i>	<i>Possible</i>	<i>Possible</i>
Read Committed	<i>Not Possible</i>	<i>Possible</i>	<i>Possible</i>
Repeatable Read	<i>Not Possible</i>	<i>Not Possible</i>	<i>Possible</i>
Serializable	<i>Not Possible</i>	<i>Not Possible</i>	<i>Not Possible</i>

Você pode ver que, assim como o nível de isolamento se move de "Read Uncommitted" para "Read Committed" e "Repeatable Read" para o último "Serializable", os tipo de comportamentos indesejáveis que podem acontecer em degraus de redução.

O nível de isolamento está configurado para usar o comando **SET TRANSACTION ISOLATION LEVEL**, usando a seguinte sintaxe:

SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }

Por padrão, o modo será configurado para "Read Committed".

Note que o PostgreSQL, no momento em que foi escrito, não pode prover o nível intermediário "Repeatable Read", nem o nível de entrada "Read Uncommitted". Geralmente, "Read Uncommitted" é um comportamento pobre que poucos bancos de dados o oferece como opção, e seria rara uma aplicação que fosse forte (ou temerária) o bastante para escolher usá-la.

Similarmente, o nível intermediário "Repeatable Read", provê somente proteção adicional contra "Phantom Reads", o qual nós dissemos ser extremamente raros, então a falta deste nível não oferece

maiores consequências. É muito comum os bancos de dados oferecerem menos opções do que o conjunto completo de possibilidades, e prover "Read Committed" e "Serializable" como solução.

MODO ENCADEADO (Auto Commit) E NÃO ENCADEADO

Através deste tópico, nós teremos que usar explicitamente **BEGIN WORK** e **COMMIT** (ou **ROLLBACK**) **WORK** para delimitar nossas transações. É muito comum no entanto, nós executarmos diretamente diversas alterações em nossas bases de dados, sem uma instrução **BEGIN WORK** ser vista.

Por padrão o PostgreSQL opera em modo Auto Commit, algumas vezes referenciado como modo encadeado ou modo de transações implícitas, onde cada instrução SQL que pode modificar um dado age como se ela fosse uma transação completa com seus próprios direitos. Isto é realmente ótimo quando se trata de interação através de linhas de comando, mas não tão bom quando se trata de sistemas reais, onde temos que ter acesso à transações com instruções explícitas **COMMIT** ou **ROLLBACK**.

Em outros servidores SQL que implementam diferentes modos, você normalmente é obrigado a digitar um comando explícito para alterar o modo. No PostgreSQL, tudo o que você precisa fazer é digitar a instrução **BEGIN COMMIT**, que o PostgreSQL automaticamente entra no modo encadeado até que uma instrução **COMMIT** (ou **ROLLBACK**) **WORK** seja digitada.

O padrão SQL não define uma instrução **BEGIN WORK**, porém, a maneira do PostgreSQL tratar transações, com um **BEGIN WORK** explícito, é muito comum.

BLOQUEIO (Locking)

Muitos bancos de dados implementam transações, em particular isolando diferentes transações entre alguns usuários, usando bloqueios para restringir o acesso aos dados por outros usuários. Simplificando, existem dois tipos de bloqueios:

- Bloqueio compartilhado (share), que permite a outros usuários ler, mas não atualizar os dados;
- Bloqueio exclusivo (exclusive), que previne que outras transações até mesmo leiam os dados;

Por exemplo, o servidor irá bloquear os registros que estão sendo alterados por uma transação, até que a transação esteja completa, quando os bloqueios são automaticamente liberados. Tudo isto é feito automaticamente, geralmente sem que os usuários saibam ou se preocupem com o que está acontecendo.

A atual mecânica e estratégia necessária para bloqueios é altamente complexa, possuindo diversos tipos diferentes de bloqueios para serem usados conforme as circunstâncias. A documentação do PostgreSQL descrevem sete tipos diferentes de permutação de bloqueios. O PostgreSQL também implementa um mecanismo específico para isolar transações utilizando um modelo de múltiplas versões, que reduz conflitos entre bloqueios, aumentando significativamente a performance comparado com outros esquemas.

Felizmente, os usuários de bancos de dados geralmente precisam se preocupar sobre bloqueios somente em duas circunstâncias, evitando deadlocks (o "abraço mortal", e se recuperando dele) e, num bloqueio explícito através de uma aplicação.

O ABRAÇO MORTAL (Deadlocks)

O que acontece quando duas diferentes aplicações tentam e alteram os mesmos dados na mesma hora? É fácil de ver, inicie duas sessões do psql, e preste atenção em alterar o mesmo registro em

ambas sessões:

DEADLOCK	
Sessão 1	Sessão 2
UPDATE row 14;	UPDATE row 15;
UPDATE row 15;	UPDATE row 14;

Neste ponto, ambas as sessões são bloqueadas, pois, cada uma delas está aguardando a outra para serem liberadas.

Inicie duas sessões do psql, e tente a seguinte sequência de comandos:

UM EXEMPLO DE DEADLOCK	
Sessão 1	Sessão 2
postgres=# BEGIN WORK; BEGIN	postgres=# BEGIN WORK; BEGIN
postgres=# UPDATE tbl_usuario SET nome='Mauricio' WHERE usuid=2;	postgres=# UPDATE tbl_usuario SET nome='Juliano' WHERE usuid=1;
postgres=# UPDATE tbl_usuario SET nome='Julio' WHERE usuid=1;	postgres=# UPDATE tbl_usuario SET nome='Marcio' WHERE usuid=2;

Você verá que ambas as sessões serão bloqueadas, e então, depois de uma curta pausa, ocorrerá o seguinte erro em uma das sessões:

ERROR:Deadlock detected.

See the lock(1) manual page for a possible cause.

A outra sessão irá continuar. A sessão que resultou na mensagem de deadlock é desfeita (rolled back), e as mudanças efetuadas até então, perdidas. A sessão que continuou poderá executar o restante das instruções que porventura ainda tenha até que a instrução COMMIT WORK seja executada, e torne as alterações do banco de dados permanente.

Não há uma maneira de saber qual sessão o PostgreSQL irá interromper e qual continuará. Ele irá testar e escolher a que considerar de menor custo de processamento, mas, isto está longe de ser uma ciência perfeita.

BLOQUEIO EXPLÍCITO

Ocasionalmente, você pode achar que o bloqueio automático que o PostgreSQL provê não é suficiente para suas necessidades, e se encontrará em uma situação onde precisa bloquear explicitamente tanto alguns registros como toda uma tabela. Você deve evitar bloqueios explícitos tanto o quanto puder. O padrão SQL nem sequer define um modo de bloquear toda uma tabela, isto é uma extensão do PostgreSQL.

È possível somente bloquear registros ou tabelas de dentro de uma transação. Assim que a transação termina, todos os bloqueios impostos pela transação são automaticamente liberados. Também não há uma maneira de liberar um bloqueio explicitamente durante uma transação, por uma simples razão que, liberando o bloqueio em um registro que está sendo alterado durante uma transação, deve possibilitar que outra aplicação o altere também, o que impossibilitaria um **ROLLBACK** de

desfazer a transação em si até a alteração inicial.

Bloqueando Registros

Para bloquear um conjunto de registros, nós simplesmente utilizamos a instrução **SELECT**, adicionando a cláusula **FOR UPDATE**:

```
SELECT * FROM tbl_usuario WHERE nome like 'J%' FOR UPDATE
```

Bloqueando Tabelas

Com o PostgreSQL também é possível bloquear uma tabela, porém, geralmente, isso não é muito recomendado, pois a performance de outras transações serão seriamente afetadas.

A sintaxe é:

```
LOCK [ TABLE ] table
```

```
LOCK [ TABLE ] table IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
```

```
LOCK [ TABLE ] table IN SHARE ROW EXCLUSIVE MODE
```

Geralmente, aplicações que necessitam bloquear uma tabela, usam simplesmente:

```
LOCK [ TABLE ] table
```

que é a mesma coisa que:

```
LOCK [ TABLE ] table IN ACCESS EXCLUSIVE MODE
```

Embora transações e bloqueios não seja um assunto - diga-se de passagem - muito interessante, porém, é de extrema importância um entendimento geral sobre os mesmos, ter uma boa noção de como estes recursos irão contribuir no desenvolvimento de aplicações sólidas, além de interagir com o servidor para minimizar implicações de performance e aumentar a segurança nos ambientes atuais.

Links do original:

http://imasters.uol.com.br/artigo/1067/postgresql/transacoes_com_o_postgresql_-_parte_i/

http://imasters.uol.com.br/artigo/1082/postgresql/transacoes_com_o_postgresql_-_parte_ii/

http://imasters.uol.com.br/artigo/1097/postgresql/transacoes_com_o_postgresql_-_parte_iii/

http://imasters.uol.com.br/artigo/1109/postgresql/transacoes_com_o_postgresql_-_parte_iv/