

TRANSAÇÕES



Wagner Bianchi

Certified MySQL 5.0 Developer

Certified MySQL 5.0 Database Administrator

Certified MySQL Cluster Database Administrator





Artigo recomendado:

- http://imasters.uol.com.br/artigo/7755/mysql/stored_procedures_-_transacoes/

Transações

- ✎ “*Transação é uma unidade lógica de trabalho, envolvendo diversas operações de bancos dados*” – (C. J. Date, Cambridge - UK 1964);
- ✎ Os conceitos existentes de transação de bancos de dados relacionais ou objeto-relacionais estão diretamente ligados aos matemáticos, autores e pesquisadores Christopher J. Date e Edgar Frank Codd, ambos autores de vários livros e também do modelo relacional, introduzido por Codd ainda quando trabalhavam juntos na IBM, em 1983 no SGBD chamado SQL/DS;
- ✎ Com maior abrangência, podemos definir transações em bancos de dados com um aglomerado **sequencial** de comandos SQL, executando várias consultas de transformação de dados, produzindo algum resultado, podendo alterar ou não o estado atual dos dados do banco de dados;

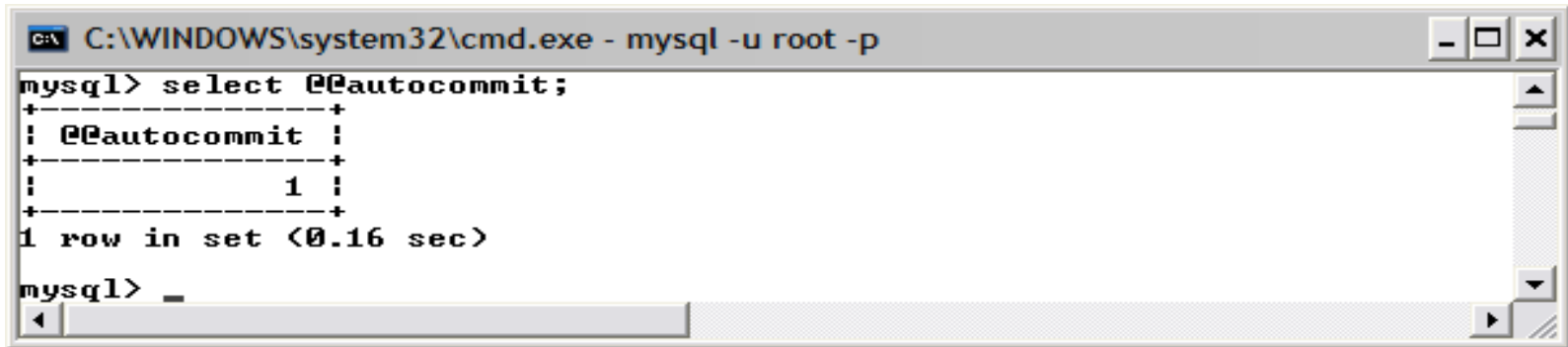
Transações

- Transações podem ser **implícitas** ou **explícitas**;
- Todo comando DML (*Data Manipulation Language*) ou DDL (*Data Manipulation Language*) que é enviado ao SGBD é considerado uma transação implícita, ou seja, após dispararmos o comando, via terminal, por exemplo, o SGBD se encarregará de enviar um **COMMIT** ao final;
- ALTER TABLE, BEGIN, CREATE INDEX, DROP INDEX, DROP TABLE, LOAD MASTER DATA, LOCK TABLES, LOAD DATA INFILE, RENAME TABLE, START TRANSACTION, UNLOCK TABLES** sofrem um **COMMIT** implícito;
- Temos uma variável de ambiente do MySQL que controla tal *feature*, denominada *autocommit*, que pode ser acessada através do seguinte comando:

```
SELECT @@autocommit;
```

Transações

- ✎ O padrão da variável *autocommit* é 1, implicando diretamente no comportamento do SGBD para cada comando que é enviado pelo usuário de um banco de dados;
- ✎ Após cada comando, a declaração **COMMIT** é enviada automaticamente para tornar permanente no log aquela transação – (mais à frente veremos com detalhes as declarações **COMMIT** e **ROLLBACK** e seus significados);



```
C:\WINDOWS\system32\cmd.exe - mysql -u root -p
mysql> select @@autocommit;
+-----+
| @@autocommit |
+-----+
|             1 |
+-----+
1 row in set (0.16 sec)
mysql> _
```

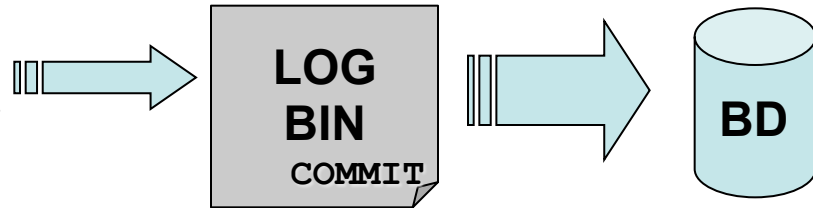
Transações

- ❏ Caso configuremos o *autocommit* como 0, teremos uma nova situação em que, caso não enviemos um **COMMIT** explicitamente após a uma transação implícita, a última transação não se tornará permanente no arquivo de dados, pois, o SGBD considerará como uma transação ainda aberta;
- ❏ O log binário do MySQL é responsável por registrar as transações em grupos – caso uma transações tenha mais de um comando -, ou uma a uma caso a transação seja formada de apenas um comando. Para que as modificações se tornem permanentes no arquivo de dados, o log ainda deve registrar um **COMMIT** ao final de cada transação;

Transações

- ❏ O MySQL trabalha intensamente com o log-bin para checar quais dados devem ser gravados nas páginas de dados em disco ou se devem sofrer um **ROLLBACK**; com a limpeza do log e retirada dele as transações que não sofreram um **COMMIT**;
- ❏ **ROLLBACK** é o contrário de **COMMIT**;
- ❏ O MySQL checa a última transação ou grupos de transação que não receberam um **COMMIT** ao final e desfaz todas as modificações o banco de dados;

```
START TRANSACTION;  
-- comando 1  
SELECT @A:=SUM(salary)  
      FROM table1 WHERE type=1;  
-- comando 2  
UPDATE table2  
      SET summary=@A WHERE type=1;  
COMMIT;
```



Transações

- Normalmente, as transações não só no MySQL, mas, na maioria dos SGBD's relacionais são iniciadas em meio à procedimentos armazenados;
- Antes de iniciarmos de fato com os comandos de transação, temos que saber o que é o bloqueio de tabelas, as chamadas **contenções**;
- O MySQL controla as contenções de tabelas de várias formas. Na verdade, esse controle é intrínseco ao Storage Engine utilizado pelas tabelas que se encontram em meio a uma transação, seja ela explícita ou implícita;
- Quando estamos utilizando uma tabela, controlada pelo Storage Engine **MyISAM**, o bloqueio de tabelas será realizado em nível de tabela, ou seja, a tabela será travada por completo, enquanto a transação atual não terminar. **Tabelas MyISAM somente suportam transações implícitas**;

Transações

- ↳ Tabelas InnoDB tem bloqueio em nível de linha, ou seja, caso uma dada transação A esteja atualizando uma linha de uma tabela, a transação B terá que esperar até que A termine suas atividades para atualizar esta mesma linha, nessa mesma tabela;
- ↳ O Storage Engine mais utilizado para ambientes que precisam de suporte à transação é o INNODB. Este Engine tem suporte à transações e contempla em 100% o modelo de transação ACID, que é um acrônimo para as propriedades:




- **Atomicidade;**
- **Consistência;**
- **Isolamento;**
- **Durabilidade;**

- ↳ Leitura complementar:

<http://databases.about.com/od/specificproducts/a/acid.htm>




Transações - ACID

Atomicidade;

-  Toda transação deverá ser atômica, o verdadeiro “*tudo ou nada*”. Um exemplo interessante seria uma transferência bancária entre contas de mesmo banco. Imagine que subtraímos o saldo da conta A e checamos a existência da conta B. Caso a conta B exista, somamos o saldo à conta B;
-  Agora imaginemos que, ao checar a existência da conta B, esta conta não existe! A quantia subtraída da conta A não poderá simplesmente desaparecer, afinal, esse “dinheiro” tem dono;
-  O comando seguinte falhará e haverá um **ROLLBACK**, restaurando o valor antes subtraído à conta de origem, no caso, a conta A;

Transações - ACID

Consistência:

-  Transações devem preservar a consistência do banco de dados, ou seja, transforma um estado consistente do banco de dados em outro estado consistente, sem necessariamente preservar o estado de consistência em todos os pontos intermediários;
-  O ponto de inconsistência plena do estado do banco de dados no exemplo da transferência bancária é exatamente no momento em que o saldo é subtraído de uma conta para ser adicionado na outra;
-  O banco é coloca em um estado sem consistência e depois retorna ao estado consistente diferente do de antes – *leve em consideração o conceito já visto de transação*;

Transações - ACID

Isolamento:

- Transações são isoladas umas das outras, de acordo com o nível de isolamento definido no momento em que a transação se inicia, que no MySQL são: **REPEATABLE READ**, **READ COMMITTED**, **READ UNCOMMITTED** e **SERIALIZABLE**;
- Também chamados de **TRANSACTION ISOLATION LEVEL** e derivados dos termos do SQL:ANSI 1992, o INNODB no MySQL tem como padrão em todas as instalações o level **REPEATABLE READ**;
- Podemos checar qual é o **ISOLATION LEVEL** atual com o seguinte comando:

```
SHOW VARIABLES LIKE '%isolation%';
```

Transações - ACID



Isolamento:

REPEATABLE READ: (SNAPSHOT) terá a mesma leitura de um dado mesmo que um **SELECT** se repita, provendo sempre o mesmo resultado para diferentes execuções da mesma consulta. Nesse nível de isolamento, se a leitura não fosse repetida, teríamos os conhecidos *phantoms*, que acontecem entre um **SELECT** e ou outro, com uma atualização dos dados nesse espaço de tempo. Isso não é possível com o Storage Engine INNODB;

READ COMMITTED: permite que a transação atual leia/manipule somente os dados já permanentes ou “comitados” por outras transações. Dados já atualizados mas que ainda não receberam um **COMMIT** explícito ainda não serão vistos, sendo estes invisíveis. Como esse nível de isolamento permite “*non-repeatable reads*”, *phantoms* podem ocorrer.

Transações - ACID

Isolamento:

-  **READ UNCOMMITTED:** permite que uma transação veja manipulações não “comitadas” de outras transações, o que pode acarretar problemas com leituras sujas – *dirty reads* - e *non-repeatable reads* que colaboram para o aparecimento dos *phantoms*;
-  **SERIALIZABLE:** esse nível de isolamento, isola completamente uma transação de outra, ou seja, enquanto uma trabalha a outra aguarda para poder iniciar o seu trabalho. Similar ao nível **READ REPEATABLE**, diferindo que as linhas selecionadas por uma transação não são nem lidas e nem atualizadas por outra transação, “**uma de cada vez**”;

Transações - ACID

- Os níveis de isolamento são importantes no contexto em que existam muitas transações acontecendo ao mesmo tempo. Os níveis trabalham os bloqueios nas linhas das tabelas ou nas tabelas para que não haja falta de consistência após o término de uma transação;
- As leituras consistentes são possíveis devido ao multiversionamento feito pelo INNODB. Uma linha poderá ter várias versões sendo trabalhadas em meio às transações, de acordo com o nível de isolamento selecionado;
- Tal *versionamento* de linhas em meio à transações, que ocorre de acordo com o nível de isolamento configurado é chamado de **MVCC** (*Multi-versioning Concurrency Control*);
- Para que esse versionamento aconteça de forma consistente, ele depende também do log de *undo* (localizado junto com os arquivos de Tablespace);

Transações - ACID

- Para alterar o nível de isolamento atual do MySQL, podemos editar o arquivo de opções (my.ini/my.cnf), e dentro do agrupamento [mysqld], colocamos a seguinte linha (reinicie o MySQL após esta alteração):

```
[mysqld]  
transaction-isolation=READ-COMMITTED
```

- Podemos alterar a partir do Terminal ou Prompt de comando, através do seguinte comando, já logado no MySQL, através do mysql client:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL isolation_level;  
SET SESSION TRANSACTION ISOLATION LEVEL isolation_level;  
SET TRANSACTION ISOLATION LEVEL isolation_level;
```


Transação - ACID

- ☞ Somente usuários do banco de dados com privilégio **SUPER** poderão utilizar o primeiro comando, que seta o nível de isolamento de forma global;
- ☞ Qualquer usuário poderá utilizar a segunda e a terceira declaração, pois, só afetarão a sessão corrente. Após a desconexão o comando será desconsiderado;



DEADLOCKS

- Deadlocks são um problema clássico em banco de dados transacionais, mas eles não são perigosos, a menos que eles sejam tão freqüentes que você não possa executar certas transações;
- Normalmente você tem que escrever suas aplicações de forma que elas sempre estejam preparada a reexecutar uma transação se for feito um **ROLLBACK** por causa de *deadlocks*;
- Uma dada transação **A** precisa atualizar uma linha da tabela **Y**, mas a transação para soltar o bloqueio desta linha, precisa adquirir um bloqueio da linha que **A** está lendo neste momento;
- O INNODB tem mecanismos que detectam os possíveis deadlocks e faz um **ROLLBACK** nas transações. Caso ocorra, a menor transação – aquela que movimenta menor número de bytes – sofrerá um **ROLLBACK**;

Exercícios

- Com base no assunto apresentado até aqui, resolva a **LISTA 2** de exercícios, valendo pontos.

Principais Comandos

- Quando se deseja trabalhar com o transações em bancos de dados contidos no MySQL, a primeira coisa que temos a fazer é entender o funcionamento do modo autocommit, que como já vimos, por padrão é configurado como 1, ou seja, ON;
- Quando *autocommit* está setado como 1, um **COMMIT** implícito é disparado ao fim de cada comando. Quando setado como 0, desabilitado, todos os comandos seguintes a esta configuração são encarados pelo SGBD como parte de uma transação, for a os comandos que recebem um **COMMIT** implícito mesmo com *autocommit* desabilitado;
- O comando **START TRANSACTION**, utilizado para iniciar uma transação, ignora o modo autocommit. Quando um procedimento dispara a declaração, autocommit é setado automaticamente para 0, sendo necessário um **COMMIT** ou **ROLLBACK** para finalizar a transação;

Principais Comandos

- Os comandos para se iniciar uma transação e que anulam o modo autocommit são:
 - **START TRANSACTION;**
 - **BEGIN WORK;**
 - **BEGIN** (diferente de BEGIN ... END);
- Caso se inicie uma transação com qualquer das declarações acima e autocommit esteja habilitado, ele é implicitamente desabilitado até que a transação finalize;
- Podemos então, iniciar transações de duas maneiras no MySQL, desabilitando explicitamente o autocommit mode ou utilizando das declarações supracitadas para desabilitar automaticamente antes de rodar os comandos;

Principais Comandos

↳ Iniciando uma transação, desabilitando autocommit mode explicitamente:

```
SET AUTOCOMMIT=0;
```

```
...declarações da transação 1...
```

```
[COMMIT | ROLLBACK];
```

```
SET AUTOCOMMIT=0;
```

```
...declarações da transação 2...
```

```
[COMMIT | ROLLBACK];
```

↳ Demonstrações:

- Iniciar uma transação desabilitando autocommit explicitamente;
- Utilizar COMMIT e ROLLBACK;

Principais Comandos

↳ Iniciando transações e suspendendo automaticamente o modo autocommit:

[START TRANSACTION | BEGIN WORK | BEGIN]

...declarações da transação 1...

[COMMIT | ROLLBACK]

[START TRANSACTION | BEGIN WORK | BEGIN]

...declarações da transação 2...

[COMMIT | ROLLBACK]

↳ Demonstrações:

- Iniciar uma transação desabilitando autocommit implicitamente;
- Utilizar COMMIT e ROLLBACK;

Principais Comandos

- Para se ter um **ROLLBACK** das transações de modo funcional e efetivo, sem correr riscos, é fundamental garantir que o modo autocommit esteja desabilitado explicita ou implicitamente;
- Somente para salientar, **ROLLBACK** é retornar todas as modificações feitas em dados por uma transação caso haja algum erro para completar esta de forma integral;
- Podemos digitar o comando **START TRANSACTION** e iniciar nossa transação comando a comando e depois entrar com um **ROLLBACK**. Veremos que tudo aquilo que fizemos comando a comando foi retornado ao momento anterior;
- O MySQL utiliza o log-binário e o tablespace do INNODB, que contém o segmento de rollback e informações de undo log para retornar ao momento anterior, transações que falharam;

Principais Comandos

- É permitido fazermos um **ROLLBACK** parcial da transação, tanto explicitamente quanto em meio à procedimentos armazenados – *Stored Procedures*;
- Utilizamos os **SAVEPOINT**'s para demarcar os pontos em meio às transações. Utilizando um **SAVEPOINT** nomeado, podemos solicitar que o SGBD faça um **ROLLBACK** até determinado ponto da transação, não voltando necessariamente tudo o que foi feito;
- Múltiplos **SAVEPOINT**'s podem ser criados em meio à transações. Para retornar uma transação até um determinado ponto onde foi detado um **SAVEPOINT**, utilizamos a seguinte sintaxe:

ROLLBACK TO SAVEPOINT *savepoint_name*;

Exemplo

```
create procedure sp_transferencia(IN id_origem INT, IN id_destino INT, valor decimal(10,2))
begin
    -- o begin não significa o início da transação
    start transaction;
    -- aqui o transação se inicia
    select count(id) into @conta_origem from conta where id =id_origem;
    select count(id) into @conta_destino from conta where id =id_destino;
    if(@conta_destino != 0 && @conta_origem != 0) then
        select saldo into @saldo_origem from conta where id =id_origem;
        if(@saldo_origem >= valor) then
            update conta set saldo =saldo-valor where id =id_origem;
            update conta set saldo =saldo+valor where id =id_destino;
            select 'Transferência realizada com sucesso.' As "Internet Banking Says:";
        else
            select 'Saldo insuficiente.' As "Internet Banking Says:";
        end if;
    else
        select 'Problema com as contas informadas.' As "Internet Banking Says:";
        rollback;
    end if;
    commit;
end;
//
K, 0 rows affected (0.00 sec)
```

Exemplo

```
mysql> call sp_transferencia(1,3,100)//
```

```
+-----+  
| Internet Banking Says:                |  
+-----+  
| Problema com as contas informadas.    |  
+-----+  
1 row in set (0.00 sec)
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> call sp_transferencia(1,2,100)//
```

```
+-----+  
| Internet Banking Says:                |  
+-----+  
| Transferência realizada com sucesso.  |  
+-----+  
1 row in set (0.00 sec)
```

```
Query OK, 0 rows affected (0.00 sec)
```

Fim!

