Artigo

Transações no MySQL

Este artigo descreve os conceitos de transações em banco de dados utilizando o MySQL como referência.

De que se trata o artigo

Este artigo descreve os conceitos de transações em banco de dados utilizando o MySQL como referência.

Para que serve:

Serve como guia introdutório ao tema de transações em banco de dados a fim de apresentar uma visão inicial e prática sobre o tema.

Resumo DevMan

Existem situações quando é vital a ordem em que as consultas são executadas, e que todos os comandos de query sejam consultados em grupo ou então que nenhum seja efetivado. Temos neste contexto a importância do conceito de transação. Neste contexto, este artigo descreve os conceitos de transações em banco de dados utilizando o MySQL como referência.

O que é uma transação?

Se você está se perguntando esta questão, provavelmente você está usando bancos de dados, onde normalmente não importa a ordem em que executamos transações, e se uma consulta falha, ela não tem impacto nas demais. Se estamos atualizando algum conjunto de conteúdo, normalmente não nos preocuparemos sobre quando a atualização é realizada, assim como o comando de leitura de dados que são executados. Similarmente, se uma atualização falha, as leituras podem ainda recuperar dados antigos nesse meio tempo.

No entanto, existem situações quando é vital a ordem em que as consultas são executadas, e que todos os comandos de query sejam consultados em grupo ou então que nenhum seja efetivado. O exemplo clássico para explicar o conceito de transações é descrito em um ambiente de um banco. Uma quantidade de dinheiro é transferida de uma conta de uma pessoa para a conta de outra pessoa, conforme o exemplo da **Listagem 1** onde é descrita uma transferência de R$ 500,00 entre 2 contas.

Listagem 1. Exemplo de transação em um sistema bancário

```
1 | UPDATE account1 SET balance=balance-500;
```

transação é simplesmente um número de queries individuais que são agrupadas e devem ser executadas juntas.

Uma pequena dose de ACID

Por um longo período, quando o MySQL não suportava transações, seus críticos concordavam que ele não estava de acordo com a regra ACID. O que eles queriam dizer é que o MySQL não estava de acordo com as quatro condições para as quais transações precisam estar aderentes para garantir a integridade dos dados. Essas quatro condições são:

• **Atomicidade:** Um átomo é dito como a menor particular possível, ou algo que não pode ser dividido. Atomicidade aplica este princípio para transações de banco de dados. As queries que compõem uma transação devem ser todas elas executadas ou nenhum delas deve ser executada (como o exemplo da **Listagem 1**).

• **Consistência:** Isso se refere às regras dos dados. Por exemplo, um corpo de artigo pode precisar de um cabeçalho de artigo associado a ele. Durante a transação (ou seja, dentro desta, durante a sua execução), esta regra pode ser quebrada, mas este estado não deve ser nunca visível de fora da transação.

• **Isolamento:** simples, os dados que estão sendo usados por uma transação não podem ser usados por qualquer outra transação até que a primeira transação seja completada. No exemplo da **Listagem 2**, temos inicialmente um saldo de conta com R$ 900,00. Existe um único depósito de R$ 100,00 e um saque de R$100,00, então o saldo da conta ao final deve permanecer o mesmo. No entanto, percebe-se que após sua execução o saldo fica em R$ 800,00, de forma que perdemos R$ 100,00. Essas 2 transações deveriam ter sido isoladas, e o resultado seria fornecido para a Conexão 2 apenas

```
Conexão 1: UPDATE conta1 SET saldo = 900+100;
Conexão 2: UPDATE conta1SET saldo = 900-100;
```

• **Durabilidade:** uma vez que uma transação for completada, seu efeito deve permanecer, e não ser reversível.

Transações em InnoDB

Transações são escritas entre declarações BEGINe COMMIT. Vamos criar uma tabela de exemplo do tipo *InnoDB* na **Listagem 3**, e ver como as transações funcionam.

Listagem 3. Criando uma tabela de exemplo

```
mysql> CREATE TABLE t (f INT) TYPE=InnoDB;
```

Agora vamos começar uma transação e inserir um registro em nossa tabela, conforme a **Listagem 4**.

Listagem 4. Escrevendo uma transação em MySQL

```
mysql> BEGIN;
mysql> INSERT INTO t(f) VALUES (1);

mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
+------+
1 row in set (0.02 sec)

mysql> ROLLBACK;
```

Leituras consistentes

Vamos tentar olhar a partir de duas conexões diferentes. Para isso, abriremos duas conexões com o banco de dados, conforme a **Listagem 5**.

Listagem 5. Abrindo 2 conexões com o banco de dados

```
-- Conexão 1
mysql> BEGIN;
mysql> INSERT INTO t (f) VALUES (1);
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

-- Conexão 2
mysql> SELECT * FROM t;
Empty set (0.02 sec)
```

O ponto importante é que executando a mesma *query* a partir de diferentes conexões (uma dentro de uma transação e outra fora) produz resultados diferentes. Agora, vamos fazer um *commit* da transação de primeira conexão e re-executar a *query* da conexão 2, e seu resultado está descrito na **Listagem 6**.

Listagem 6. Executando a consulta 2 após um commit na conexão 1

```
-- Conexão 1:
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

```
12 |
```

Este comportamento é chamado de *leitura consistente*. Qualquer SELECT retorna um resultado antigo até que exista uma transação mais recente que tenha sido completada, com exceção da conexão que está fazendo a atualização, como exibido anteriormente. Por padrão, as tabelas *InnoDB* do MySQL realizam leitura consistente.

Commits automáticos

O MySQL ainda faz *commits* automaticamente em declarações que não são partes de uma transação. Os resultados de qualquer declaração de UPDATE ou INSERT não precedidas por um BEGIN estarão imediatamente visíveis para todas as conexões. Podemos mudar este comportamento através do comando exibido na **Listagem 7**.

Listagem 7. Desabilitando commits automáticos

```
1 | mysql> SET AUTOCOMMIT=0;
```

Agora, vejamos na **Listagem 8** o que acontece se não iniciarmos uma transação especificamente com BEGIN.

Listagem 8. Verificando o comportamento de operações sem BEGIN

```
1 | -- Conexão 1:
2 | mysql> INSERT INTO t (f) VALUES (2);
3 | mysql> SELECT * FROM t;
4 | +------+
5 | | f    |
6 | +------+
7 | |    1 |
8 | |    2 |
```

```
|    1 |
+------+
1 row in set (0.00 sec)
```

Para vermos o resultado se tivéssemos o commit da transição automático, primeiro devemos configurar o parâmetro AUTOCOMMITdo MySQL para 1. Feito isto, repetimos os comandos. Veremos na **Listagem 9** que o resultado será diferente.

Listagem 9. Repetindo as consultadas da Listagem 8 com commit automático

```
mysql> SET AUTOCOMMIT=0;

-- Conexão 1:
mysql> COMMIT;
mysql> INSERT INTO t (f) VALUES (3);

-- Conexão 2:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

Desta vez o *commit* na transação é feito imediatamente e fica visível para qualquer outra conexão, mesmo sem especificar a declaração COMMIT.

Bloqueios de leitura para atualização

Algumas vezes, a leitura consistente padrão não é aquela que você deseja. Existem

Listagem 10. Consultas em 2 conexões acessando o mesmo registro

```
-- Conexão 1:
mysql> BEGIN;
mysql> SELECT MAX(f) FROM t;
+--------+
| MAX(f) |
+--------+
|      3 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO t(f) VALUES (4);

-- Conexão 2:
mysql> BEGIN;
mysql> SELECT MAX(f) FROM t;
+--------+
| MAX(f) |
+--------+
|      3 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO t(f) VALUES(4);

mysql> COMMIT;

-- Conexão 1:
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
```

atualização. Isso especifica que nenhuma outra conexão pode ler este dado até que a transação seja completada. Na **Listagem 11** está apresentado um bloqueio de atualização. Primeiro, deletamos um registro incorretamente, então começamos uma transação na conexão 1 e inserimos um registro. Enquanto isso, na conexão 2 realizamos uma consulta na mesma tabela que esta sendo atualizada.

Listagem 11. Testando o bloqueio de atualização

```
mysql> DELETE FROM t WHERE f=4;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)

-- Conexão 1:
mysql> BEGIN;
mysql> INSERT INTO t(f) VALUES (4);
Query OK, 1 row affected (0.00 sec)

-- Conexão 2:
mysql> SELECT MAX(f) FROM t FOR UPDATE;
```

Nenhum resultado é retornado – o MySQL está aguardando a transação ativa ser completada, e só então ela retorna os resultados. O código que completa esta operação está exibido na **Listagem 12**. Os resultados são agora retornados à conexão 2. Note que a leitura pode levar muito tempo, e então precisaremos esperar pelos resultados.

Listagem 12. Completando a consulta aos dados que estavam bloqueados

```
| MAX(f) |
+--------+
|      4 |
+--------+
1 row in set (4.23 sec)

mysql> INSERT INTO t(f) VALUES(5);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
+------+
5 rows in set (0.00 sec)
```

Bloqueios de leitura para compartilhamento

Um outro tipo de bloqueio garante que você sempre leia os últimos dados, mas ele não é parte da transação que está atualizando os dados. Este é o LOCK IN SHARE MODE. Ele irá interromper qualquer atualização ou exclusão da linha que está sendo lida, e se o último dado ainda não foi comitado, irá aguardar até que a transação seja comitada antes de retornar qualquer resultado. Vemos um exemplo na **Listagem 13**. Enquanto isso, uma segunda conexão tenta realizar uma atualização. A atualização aguarda até que o bloqueio da outra conexão seja liberado.

Listagem 13. Completando a consulta aos dados que estavam bloqueados

```
1 row in set (0.00 sec)

-- Conexão 2:
mysql> UPDATE t SET f = 55 WHERE f=5;
Query OK, 0 rows affected (6.95 sec)
Rows matched: 0  Changed: 0  Warnings: 0

-- Conexão 1:
mysql> COMMIT;

-- Conexão 2:
mysql> UPDATE t SET f = 55 WHERE f=5;
Query OK, 1 row affected (43.30 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)
```

Agora iremos ver como eles afetam o comportamento do bloqueio transacional padrão.

Níveis de isolamento de transação

Um nível de isolamento de transação configura o comportamento transacional padrão. Os exemplos anteriores usavam a configuração padrão. Agora veremos como mudar o

• **READ UNCOMMITTED:** pouco transacional, esta configuração permite as chamadas "leituras-sujas", onde *queries* dentro de uma transação são afetadas por mudanças não comitadas em outra transação.

• **READ COMMITTED:** mudanças comitadas são visíveis dentro de outra transação. Isso significa que *queries* idênticas dentro de uma transação podem retornar resultados diferentes. Esta é a configuração padrão em vários SGBD's.

• **REPEATABLE READ:** o nível de isolamento padrão para tabelas InnoDB. Dentro de uma transação, todas as leituras são consistentes.

• **SERIALIZABLE:** atualizações não são permitidas em outras transações se uma transação está executando uma *query* de SELECT ordinária, ou seja, *queries* são tratadas como se elas possuíssem um LOCK IN SHARE MODE, que vimos em ação nas seções passadas.

Tabelas *InnoDB* suportam todos os quarto níveis de isolamento de transação padrões de SQL. É preciso ter cuidado quando converter código de outros SGBD's, pois eles podem não suportar todos os quatro tipos.

Voltaremos a usar a tabela *t* usada nos exemplos anteriores.

Se você não fez nenhuma mudança específica para o nível de isolamento de transação, ela deverá ser uma leitura repetível (REPEATABLE READ), como pode ser observado com o comando da **Listagem 14**.

Listagem 14. Verificando o nível de isolamento de transação

```
1 mysql> SELECT @@tx_isolation;
2 +-----------------+
```

No exemplo da **Listagem 15**, começamos uma transação e vemos se um INSERT comitado em outra transação está visível no meio da transação.

Listagem 15. Exemplo de leitura repetível

```
-- Conexão 1
mysql> BEGIN;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)

-- Conexão 2
mysql> BEGIN;
mysql> INSERT INTO t VALUES(6);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

```
-- Conexão 1
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)

mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Esta é a essência da leitura repetível. A query SELECT retorna um resultado consistente dentro da transação, e novos registros adicionados a partir de outra janela durante a transação não estão imediatamente visíveis. Para um resultado ser visível, tanto a transação da atualização como qualquer outra transação que esteja aberta precisam ser comitados.

Leituras não comitadas – Uncommitted Reads

```
mysql> SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

Usaremos duas conexões para o exemplo da **Listagem 18**, pois o novo nível de isolamento de transação só faz efeito para novas conexões realizadas após a sua configuração.

Listagem 18. Exemplo de leitura não comitada

```
-- Conexão 1:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)

-- Conexão 2:
mysql> BEGIN;
mysql> INSERT INTO t VALUES (7),(8);

-- Conexão 1:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
```

Isso é conhecido como uma leitura suja – os novos registros não foram ainda comitados pela segunda transação, mas eles já estão visíveis para a primeira transação. Isso pode ser um problema, pois os comandos da conexão 2 podem ser desfeitos através de uma operação de rollback, impactando diretamente nos resultados gerados pela conexão 1, pois esta usou dados ainda não comitados (ver **Listagem 19**).

Listagem 19. Desfazendo as operações não comitadas

```
-- Conexão 2:
mysql> ROLLBACK;

-- Conexão 1:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Existem perigos com este nível de isolamento e ele dribla as regras das transações. Você poderia apenas usar este nível onde você realmente não se preocupa com consistência dos resultados, mas se preocupa com os potenciais impactos dos bloqueios no desempenho.

Leituras Comitadas – Committed Reads

Um exemplo do uso deste nível de isolamento está apresentado na **Listagem 21**.

Listagem 21. Exemplo do uso do nível de isolamento Committed Read

```
-- Conexão 1:
mysql>BEGIN;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)

-- Conexão 2:
mysql> BEGIN;
mysql> INSERT INTO t VALUES (7),(8);

-- Conexão 1:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

```
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|    8 |
+------+
8 rows in set (0.00 sec)

mysql> COMMIT;
```

Uma diferença importante é que um INSERT não comitado não tem qualquer impacto na transação da Conexão 1. Uma vez que a transação da conexão 2 for comitada, o resultado ficou visível para a primeira transação. É também importante distinguir a diferença entre este e o nível de isolamento de transação READ REPEATABLE visto anteriormente. Com READ COMMITTED, mudanças são visíveis quando outras transações comitam suas mudanças. Com REPEATABLE READ, mudanças apenas são visíveis quando ambas as transações são comitadas, e apenas em uma nova transação. Entender este ponto importante traz a essência da diferença entre os dois níveis.

Serial – Serializable

O nível serial trata de bloqueios além daqueles realizados no nível REPEATABLE READ. Neste nível, todas as *queries* de SELECT regulares são tratadas como se elas possuíssem um LOCK IN SHARE MODE anexado. A **Listagem 22** apresenta um exemplo do seu uso. Por causa da declaração de SELECT da primeira conexão, o UPDATE está bloqueado, exatamente como um LOCK IN SHARE MODE ordinário.

```
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|    8 |
+------+

-- Conexão 2:
mysql> BEGIN;
mysql> UPDATE t SET f=88 WHERE f=8;

-- Conexão 1:
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

-- Conexão 2:
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|   88 |
+------+
8 rows in set (0.00 sec)
```

estivéssemos usando um tipo de tabela não transacional MyISAM), como demonstrado na **Listagem 23**.

Listagem 23. Exemplo do uso do nível de isolamento SERIALIZABLE

```
Conexão 1:
mysql> INSERT INTO t VALUES(1);

Conexão 2:
mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

Podemos contornar este comportamento configurando o parâmetro AUTOCOMMIT para 0, de forma que todas as declarações são tratadas como se uma declaração BEGIN precedesse o comando, como no exemplo da **Listagem 24**. Novamente surge uma thread, onde a segunda conexão aguarda por um *commit* na primeira conexão.

Listagem 24. Testando a desabilitação do parâmetro autocommit

```
Conexão 1:
mysql> SET AUTOCOMMIT=1;
mysql> INSERT INTO t VALUES(2);

Conexão 2:
mysql> SELECT * FROM t;

Conexão 1:
mysql> COMMIT;
```

```
+------+
2 rows in set (0.00 sec)
```

Bloqueio de tabela para todos os tipos

As tabelas usam bloqueio por linha, em que só a linha que está sendo manipulada é bloqueada. Isso significa que outras linhas podem ainda ser manipuladas, reduzindo o risco de contenção, mas também é um processo menos otimizado que o bloqueio por tabela se a maioria das *queries* são consultas. Além disso, todos os tipos de tabela podem fazer uso do bloqueio por tabela, usando a declaração LOCK TABLE.

Existem dois tipos de bloqueios – *read* (leitura) e *write* (escrita). Um bloqueio de leitura permite apenas leitura na tabela, sem escrita, enquanto um bloqueio de escrita apenas permite que conexões leiam e escrevam nas tabelas – todas as outras conexões são bloqueadas. Vamos ver isso em ação no exemplo da **Listagem 25**. No primeiro trecho, a segunda conexão trava, aguardando a tabela ser desbloqueada. No entanto, uma leitura na segunda conexão é processada imediatamente, como descrito no segundo trecho.

Listagem 25. Exemplificando bloqueios de leitura

```
Conexão 1:
  mysql> LOCK TABLE t READ;

  Conexão 2:
  mysql> INSERT INTO t VALUES(1);

  Conexão 1:
  mysql> UNLOCK TABLES;

```

```
+------+
|    1 |
+------+
1 row in set (0.00 sec)

  Conexão 1:
  mysql> UNLOCK TABLES;
```

Vamos tentar a mesma declaração com um bloqueio de escrita, na **Listagem 26**. No primeiro momento a conexão trava aguardando pelo bloqueio ser liberado.

Listagem 26. Exemplificando bloqueios de escrita

```
Conexão 1:
  mysql> LOCK TABLE myisam_table WRITE;

  Conexão 2:
  mysql> SELECT * FROM myisam_table;

  Conexão 1:
  mysql> UNLOCK TABLE;

  Conexão 2:
  mysql> SELECT * FROM myisam_table;
+------+
| f    |
+------+
|    1 |
+------+
1 row in set (6.22 sec)
```

Se pensarmos um pouco sobre isso, você pode perguntar o que acontece se a conexão que cria um bloqueio de leitura tenta inserir um registro. Visto que é proibido em um bloqueio de leitura, mesmo para a conexão de origem, e se a conexão tiver que

```
ERROR 1099: Table 't' was locked with a READ lock and can't be updated

mysql> UNLOCK TABLE;

```

Prioridade de bloqueio

Bloqueios de escrita possuem uma prioridade maior que bloqueios de leitura. Um bloqueio de escrita será sempre obtido na frente de quaisquer bloqueios anteriores de leituras que estão aguardando. Vamos demonstrar isso na **Listagem 28**. A conexão trava, aguardando pelo bloqueio mais antigo ser liberado.

Listagem 28. Exemplo de priorização de bloqueios

```
Conexão 1:
mysql> LOCK TABLE t WRITE;

Conexão 2:
mysql> LOCK TABLE t READ;
```

Agora teremos, na **Listagem 29**, dois bloqueios aguardando, um bloqueio de leitura, e um bloqueio de escrita requisitado após o bloqueio de leitura. Quando liberarmos o primeiro bloqueio, o bloqueio gerado na Terceira conexão, pois é o bloqueio de escrita precedente, é obtido. Apenas após a liberação do bloqueio da conexão 3 a conexão 2 fica ativa.

Listagem 29. Continuidade do exemplo de priorização de bloqueios

```
Conexão 3:
  mysql> LOCK TABLE t WRITE;

```

Não é preciso muita imaginação para entender porque bloqueio de tabela pode impactar no desempenho se existem muitos bloqueios de escrita.

Algumas vezes queremos que um bloqueio de escrita tenha prioridade mais baixa que um bloqueio de leitura. Usando a mesma tabela e o mesmo conjunto de bloqueios feito antes, exceto que a terceira conexão é um bloqueio de prioridade baixa, vamos ver o que acontece na **Listagem 30**. A conexão trava, aguardando pelo bloqueio mais antigo ser liberado.

Listagem 30. Exemplo com mudança de priorização nos bloqueios

```
1  Conexão 1:
2  mysql> LOCK TABLE t WRITE;
3
4  Conexão 2:
5  mysql> LOCK TABLE t READ;
```

Na continuidade do exemplo (**Listagem 31**), é feito o bloqueio da conexão 3 e em seguida a liberação do bloqueio da conexão 1. Desta vez, a conexão 2 fica ativa, enquanto que o bloqueio de escrita na conexão 3 ainda fica aguardando, e só após a liberação do bloqueio da conexão 2 que a conexão 3 fica ativa.

Listagem 31. Continuidade com mudança de priorização nos bloqueios

```
1  Conexão 3:
2  mysql> LOCK TABLE t LOW_PRIORITY WRITE;
3
4  Conexão 1:
5  mysql> UNLOCK TABLE;
6
7  Conexão 2:
```

uma transação seja feita completamente ou nada será persistido, normalmente encontrado em transações.

Vamos voltar à nossa tabela e ver o funcionamento deste recurso, na **Listagem 32**. Dentro da mesma transação, iremos inserir dois novos registros, um antes do savepoint e outro após.

Listagem 32. Exemplo com savepoints

```
mysql> INSERT INTO t VALUES(9);
mysql> SAVEPOINT x;
mysql> INSERT INTO t VALUES(10);

mysql> ROLLBACK TO SAVEPOINT x;

mysql> SELECT * FROM t;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|   88 |
|    9 |
+------+
9 rows in set (0.00 sec)
```

O primeiro INSERT foi realizado – efetivamente um savepoint pode ser chamado de COMMIT UNTIL (commit até), então qualquer coisa antes do savepoint será comitada. O segundo INSERT, depois do savepoint, é desfeito (rollback).

podem levar a infinitas horas de diversão se você não tomar cuidado e não tomar conhecimento das sutilezas.

Os SGBDs cada vez mais têm investido em recursos que apóiam transações em bancos de dados. É preciso ficar sempre atento às inovações nesta área.

Por **Ian**
Em 2011