Transactions are the core function of maintaining data consistency in the SQL world. Basically, all mainstream relational databases support the transaction feature, and even some non-relational databases software can support the transaction feature (e.g. Mongo DB - Transaction). You can imagine its importance in practical systems.

Transactions are important because there are some statements that we want to completely succeed or to completely fail, but a partial success is not an option. An example is transferring money to the bank.

Therefore, we highly recommend that you learn this concept well!

Sample table

products

id	price	title
1	10	chair
2	100	desk
3	200	bike
4	200	motorcycle
5	300	headphone
6	300	phone

To help solidify the new concepts, you can follow along by editing the provided table below:

Schema SQL

```
CREATE SCHEMA `new_schema` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;


CREATE TABLE `new_schema`.`products` (
  `id` INT NOT NULL AUTO_INCREMENT COMMENT 'This is the primary index',
```

```
  `title` VARCHAR(45) NOT NULL DEFAULT 'N/A',
  `price` INT NULL,
  PRIMARY KEY (`id`)
);
INSERT INTO `new_schema`.`products` (`id`, `price`, `title`) VALUES
  (1, 10, 'chair'),
  (2, 100, 'desk'),
  (3, 200, 'bike'),
  (4, 200, 'motorcycle'),
  (5, 300, 'headphone'),
  (6, 300, 'phone');
```

Query SQL

```
SELECT * FROM `new_schema`.`products`;
```

Result

Query #1 **Execution time: 1ms**

id	title	price
1	chair	10
2	desk	100
3	bike	200
4	motorcycle	200
5	headphone	300
6	phone	300

Concept

The usage of transaction syntax is somewhat different from other SQL syntax. The grammar we learned in the past was one-time execution, and the implementation result is simple, either success or failure.

When using transaction, it's like a box in which you can put many SQL statements and execute them together.

The transaction has state, when we execute the transaction, all SQL statements in the transaction will be marked as queries belonging to the transaction state. With subsequent logical judgments, you can decide the results of these queries are indeed stored, or restore the results caused by these queries.

Basic Statement

First of all, let's see how to set the transaction state for our database:

```
--First Query
START TRANSACTION;

-- Second Queries
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 5;
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 6;
```

Before starting, notice that this example is divided into two parts which are executed at different times. The first part is `START TRANSACTION`, and the second part consists of two `UPDATE` statements.

First, let's consider the `START TRANSACTION` keyword. When this statement is executed, the SQL we execute later on will enter the transaction state. That is to say, when we execute the `UPDATE` statements that update the price of the product `id = 5` and `id = 6` to 500, it will be marked as a transaction state.

When an SQL statement is marked as transactional, it means that its execution result will not be stored until the COMMIT command has been executed.

So, the complete command with COMMIT will be as follows:

```
-- First Query
START TRANSACTION;
```

```
-- Second Queries
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 5;
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 6;

-- Third Query
COMMIT;
```

In MySQL, if you use MySQL Workbench to simulate our samples, it will use the auto-commit mode in default. It means that commit will be executed in the background by default, which will affect query test results that don't include "START TRASACTION". For those who are interested in learning more, please refer to: MySQL Autocommit Description.

Restore Statement

Real life is always full of surprises, and the probability of encountering bugs in the code is very high. Thus, there is a transaction mechanism for solving this problem.

Take the previous example, and suppose that a bug is triggered when we are only halfway through the queries for updating the price. In this case, the whole batch of updates should be discarded.. In order to undo the changes caused by these queries, we need to use the `ROLLBACK` keyword.

Here is an example of how to use `ROLLBACK`:

```
-- First Query
START TRANSACTION;

-- Second Queries
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 5;
..
-- error happened here!
..
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 6;

-- Third Query
ROLLBACK;
```

Please pay attention to how we arranged the statements. In the `Second Queries` section, if an error occurs in the middle, we will skip the second `UPDATE` statement and directly jump to the third

section (`ROLLBACK`). This will then restore all the results to what they were before we started the transaction.

By rolling back when we encounter an error, we can avoid inconsistencies in the data (some product prices have been updated, while others are still at their original prices).

In most projects, transaction, commit, and rollback will be controlled by applications in other programming languages. However, in some large projects, it is possible to have transaction statements that are purely controlled by SQL. At this time, other SQL functions are usually used like this:

```
START TRANSACTION;

SELECT `new_schema`.`products` WHERE id = 5;
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 5;

IF (@correct) THEN
  COMMIT;
ELSE
  ROLLBACK;
END IF;
```

The logic of this code is if there are no errors, then we will commit, otherwise, we will roll back.

It has a relatively more complex structure than a normal SQL statement. Interested readers are welcome to refer to MySQL Compound Statement about the control process in SQL.

`@correct` is the SQL variable, reference: MySQL user variables