CHAPTER 7

Using Triggers and Scheduled Events

In addition to executing SQL statements and calling stored routines on an ad-hoc basis, MySQL 5.0 introduced database triggers, which allow these actions to be performed automatically by the server. This was not entirely unexpected—triggers and stored routines tend to go hand-in-hand, and both items were in demand from the user community—but it *was* a pleasant surprise to see MySQL 5.1 improve on this even further by introducing a new subsystem for scheduled events.

This event scheduler, together with MySQL's support for triggers, provide a powerful framework for automating database operations, one that can come in handy when constructing complex or lengthy application workflows. This chapter builds on the material in the previous chapter, introducing you to MySQL's implementation of triggers and scheduled events, and providing examples that demonstrate how they can be used in real-world applications.

Understanding Triggers

A *trigger*, as the name suggests, refers to one or more SQL statements that are automatically executed ("triggered") by the database server when a specific event occurs. Triggers can come in handy when automating database operations, and thereby reduce some of the load carried by an application. Common examples of triggers in use include:

- Logging changes in data
- Creating "snapshots" of data prior to a change (for undo functionality)
- Performing automatic calculations
- Changing data in one table in response to a change in another

A trigger is always associated with a particular table, and it can be set to execute either before or after the trigger event takes place. MySQL currently supports three types of trigger events: `INSERT`s, `UPDATE`s, and `DELETE`s.

A Simple Trigger

To understand how triggers work, let's consider a simple example: logging changes to the airline's flight database. Let's suppose that every time an administrator adds a new flight to the database, this action should be automatically logged to a separate table, along with the administrator's MySQL username and the current time. With a trigger, this is easy to do:

```
mysql> CREATE TRIGGER flight_ai
    ->   AFTER INSERT ON flight
    ->   FOR EACH ROW
    ->     INSERT INTO log (ByUser, Note, EventTime)
    ->     VALUES (CURRENT_USER(), 'Record added: flight', NOW());
Query OK, 0 rows affected (0.04 sec)
```

To define a trigger, MySQL offers the `CREATE TRIGGER` command. This command must be followed by the trigger name and the four key trigger components, namely:

- The trigger *event,* which can be any one of `INSERT`, `UPDATE`, or `DELETE`
- The trigger *activation time,* which can be either `AFTER` the event or `BEFORE` it
- The trigger's *subject table,* which is the table the trigger should be attached to
- The trigger *body,* which contains the SQL statements to be executed

---

NOTE *To create a trigger, a user must have the* `TRIGGER` *privilege (in MySQL 5.1.6+) or the* `SUPER` *privilege (in MySQL 5.0.x). Privileges are discussed in greater detail in Chapter 11.*

These components are illustrated in the previous example, which creates a trigger named *flight_ai.* The `FOR EACH ROW` clause in the trigger ensures that it is activated after every operation that adds a new record to the *flight* table and it, in turn, adds a record to the *log* table recording the operation. To see this trigger in action, try adding a new record to the *flight* table, as shown:

```
mysql> INSERT INTO flight (FlightID, RouteID, AircraftID)
    -> VALUES (900, 1141, 3452);
Query OK, 1 row affected (0.08 sec)
mysql> SELECT * FROM log\G
*************************** 1. row ******************
 RecordID: 2
   ByUser: root@localhost
     Note: Record added: flight
EventTime: 2009-01-09 15:40:46
1 row in set (0.00 sec)
```

It's easy to add another trigger, this one to log record deletions. Here's an example:

```
mysql> CREATE TRIGGER flight_ad
    ->   AFTER DELETE ON flight
    ->   FOR EACH ROW
    ->     INSERT INTO log (ByUser, Note, EventTime)
    ->     VALUES (CURRENT_USER(), 'Record deleted: flight', NOW());
Query OK, 0 rows affected (0.08 sec)
```

And now, when you delete a record, that operation should also be recorded in the log table:

```
mysql> DELETE FROM flight
    -> WHERE flightid = 900;
Query OK, 1 row affected (0.01 sec)
mysql> SELECT * FROM log\G
*************************** 1. row ***************
 RecordID: 3
   ByUser: root@localhost
     Note: Record deleted: flight
EventTime: 2009-01-09 15:42:42
*************************** 2. row ***************
 RecordID: 2
```

```
   ByUser: root@localhost
     Note: Record added: flight
EventTime: 2009-01-09 15:40:46
2 rows in set (0.00 sec)
```

How do I Name My Triggers?

Peter Gulutzan has suggested an easy-to-understand and consistent naming scheme for triggers in his article at http://dev.mysql.com/tech-resources/articles/mysql-triggers.pdf, which is also followed in this chapter: Name each trigger with the name of the table to which it is linked, with an additional suffix consisting of the letters *a* (for "after") or *b* (for "before"), and *i* (for "insert"), *u* (for "update") and *d* (for "delete"). So, for example, an `AFTER INSERT` trigger on the *pax* table would be named *pax_ai*.

The main body of the trigger is not limited only to single SQL statements; it can contain any of MySQL's programming constructs, including variable definitions, conditional tests, loops, and error handlers. `BEGIN` and `END` blocks are mandatory when the procedure body contains these complex control structures. In all other cases (such as the previous example, which contains only a single `INSERT`), they are optional.

NOTE *To avoid ambiguity, MySQL does not allow more than one trigger with the same trigger event and trigger time per table. This means that, for example, a table cannot have two* `AFTER INSERT` *triggers (although it can have separate* `BEFORE INSERT` *and* `AFTER INSERT` *triggers). Or, to put it another way, a table can have, at most, six possible triggers.*

To remove a trigger, use the `DROP TRIGGER` command with the trigger name as argument:

```
mysql> DROP TRIGGER flight_ad;
Query OK, 0 rows affected (0.03 sec)
```

TIP *Dropping a table automatically removes all triggers associated with it.*

To view the body of a specific trigger, use the `SHOW CREATE TRIGGER` command with the trigger name as argument. Here's an example:

```
mysql> SHOW CREATE TRIGGER flight_ad\G
*************************** 1. row ***************************
               Trigger: flight_ad
              sql_mode: STRICT_TRANS_TABLES
SQL Original Statement: CREATE DEFINER=`root`@`localhost`
  TRIGGER flight_ad
  AFTER DELETE ON flight
  FOR EACH ROW
```

```
    INSERT INTO log (ByUser, Note, EventTime)
    VALUES (CURRENT_USER(), 'Record deleted: flight', NOW());
  character_set_client: latin1
  collation_connection: latin1_swedish_ci
    Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

To view a list of all triggers on the server, use the `SHOW TRIGGERS` command. You can filter the output of this command with a `WHERE` clause, as shown:

```
mysql> SHOW TRIGGERS FROM db1 WHERE `Table` = 'flight'\G
*************************** 1. row ***************************
             Trigger: flight_ai
               Event: INSERT
               Table: flight
           Statement: INSERT INTO log (ByUser, Note, EventTime)
   VALUES (CURRENT_USER(), 'Record added: flight', NOW());
              Timing: AFTER
             Created: NULL
            sql_mode: STRICT_TRANS_TABLES
             Definer: root@localhost
character_set_client: latin1
collation_connection: latin1_swedish_ci
  Database Collation: latin1_swedish_ci
*************************** 2. row ***************************
             Trigger: flight_ad
               Event: DELETE
               Table: flight
           Statement: INSERT INTO log (ByUser, Note, EventTime)
   VALUES (CURRENT_USER(), 'Record deleted: flight', NOW());
              Timing: AFTER
             Created: NULL
            sql_mode: STRICT_TRANS_TABLES
             Definer: root@localhost
character_set_client: latin1
collation_connection: latin1_swedish_ci
  Database Collation: latin1_swedish_ci
2 rows in set (0.00 sec)
```

Trigger Security

The `CREATE TRIGGER` command supports an additional `DEFINER` clause, which specifies the user account whose privileges should be considered when executing the trigger. For the trigger to execute successfully, this user should have all the privileges necessary to perform the statements listed in the trigger body. By default, MySQL sets the `DEFINER` value to the user who created the trigger.

Here's an example:

```
mysql> CREATE DEFINER = 'jack@example.com'
    -> TRIGGER flight_ad
```

```
    ->   AFTER DELETE ON flight
    ->   FOR EACH ROW
    ->     INSERT INTO log (ByUser, Note, EventTime)
    ->     VALUES (USER(), 'Record deleted: flight', NOW());
Query OK, 0 rows affected (0.08 sec)
```

Which is Better: a `BEFORE` trigger or an `AFTER` trigger?
There's no hard-and-fast rule as to which trigger is "better"—it's like asking which flavor of ice cream is best. But if you're stuck trying to decide whether your code should run before or after a DML operation, the following rule of thumb (posted by Scott White in the online MySQL manual, at http://dev.mysql.com/doc/refman/5.0/en/create-trigger.html) might help: "Use `BEFORE` triggers primarily for constraints or rules, not transactions. Stick with `AFTER` triggers for most other operations, such as inserting into a history table or updating a denormalization."

Triggers and Old/New Values

Within the body of a trigger, it's possible to reference field values from both before and after the trigger event by prefixing the field name with the `OLD` and `NEW` keywords. This means that, for example, if you have an `UPDATE` trigger on a table, the SQL statements within the trigger body can access both the existing field values (`OLD`) and the new, incoming field values (`NEW`).

To illustrate this, consider the next example, which logs changes to the *flight* table and specifies the changed values as part of the log message:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER flight_au
    -> AFTER UPDATE ON flight
    -> FOR EACH ROW
    ->   BEGIN
    ->     DECLARE str VARCHAR(255) DEFAULT '';
    ->     IF OLD.FlightID != NEW.FlightID THEN
    ->       SET str = CONCAT(str, 'FlightID ',
    ->         OLD.FlightID, ' -> ', NEW.FlightID, ' ');
    ->     END IF;
    ->     IF OLD.RouteID != NEW.RouteID THEN
    ->       SET str = CONCAT(str, 'RouteID ',
    ->         OLD.RouteID, ' -> ', NEW.RouteID, ' ');
    ->     END IF;
    ->     IF OLD.AircraftID != NEW.AircraftID THEN
    ->       SET str = CONCAT(str, 'AircraftID ',
    ->         OLD.AircraftID, ' -> ', NEW.AircraftID);
    ->     END IF;
    ->     INSERT INTO log (ByUser, Note, EventTime)
```

```
    ->     VALUES (USER(),
    ->       CONCAT('Record updated: flight: ', str),
    ->       NOW());
    ->   END//
Query OK, 0 rows affected (0.00 sec)
```

In this example, the prefix `OLD` returns the pre-update value of the corresponding field, while the prefix `NEW` returns the post-update value of the field. Within the trigger body, `IF` conditional tests are used to check if the old and new values are the same; if not, the field is flagged and its old and new values are inserted as part of the log string.

`OLD` and `NEW` values typically appear together only in `UPDATE` triggers. This is only logical: `OLD` values are neither relevant nor supported in the case of `INSERT` triggers, while the same applies to `NEW` values for `DELETE` triggers.

Triggers and More Complex Applications

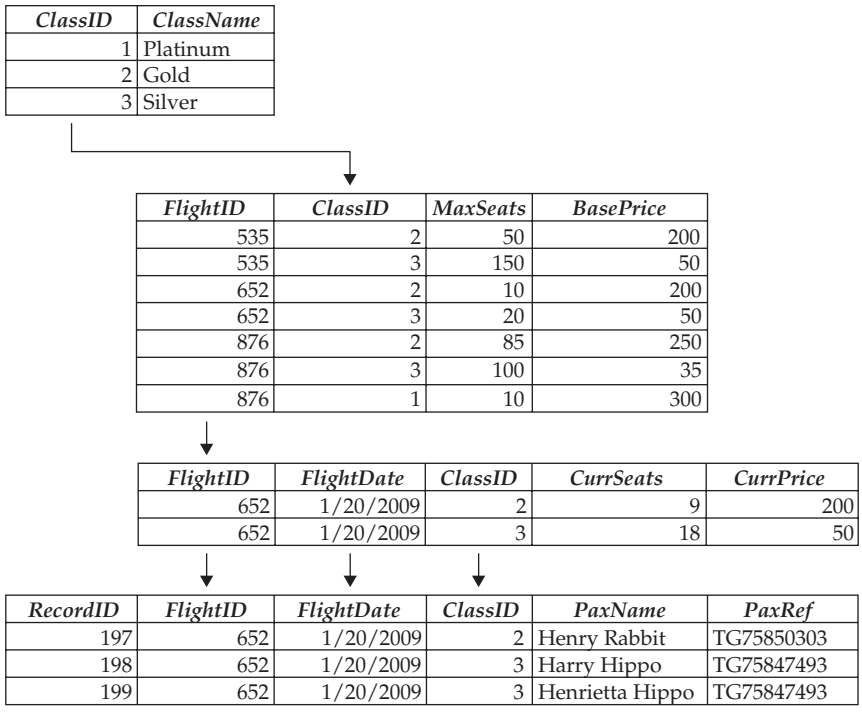
Let's look at another, more complex example. Consider that an airline has a limited inventory of seats per flight and flight class, and the seat inventory for each flight needs to be updated on a continual basis as passengers book their flights. Consider also that the airline would like to automatically increase the price of tickets as the flight begins to fill up in order to increase its profit margin.

Figure 7-1 explains how this information is stored in the example database.

- Passenger records for each flight and class combination are recorded in the *pax* table.
- The live seat inventory for a particular flight-and-class combination can be found in the *stats* table.
- The *pax* and *stats* tables are linked to each other by means of the common *FlightID, FlightDate,* and *ClassID* fields.
- The maximum number of seats possible in each class of a particular flight, together with the base (starting) ticket price, is recorded in the *flightclass* table.

So, for example, flight #652 which operates on the Orly-Budapest route, has a maximum of 10 seats available in Gold class at a base price of $200 and 20 seats available in Silver class at a base price of $100.

```
mysql> SELECT FlightID, ClassID, MaxSeats, BasePrice
    -> FROM flightclass WHERE FlightID=652;
+----------+---------+----------+-----------+
| FlightID | ClassID | MaxSeats | BasePrice |
+----------+---------+----------+-----------+
|      652 | 2       |       10 |       200 |
|      652 | 3       |       20 |        50 |
+----------+---------+----------+-----------+
2 rows in set (0.00 sec)
```



ClassID	*ClassName*
1	Platinum
2	Gold
3	Silver

FlightID	*ClassID*	*MaxSeats*	*BasePrice*
535	2	50	200
535	3	150	50
652	2	10	200
652	3	20	50
876	2	85	250
876	3	100	35
876	1	10	300

FlightID	*FlightDate*	*ClassID*	*CurrSeats*	*CurrPrice*
652	1/20/2009	2	9	200
652	1/20/2009	3	18	50

RecordID	*FlightID*	*FlightDate*	*ClassID*	*PaxName*	*PaxRef*
197	652	1/20/2009	2	Henry Rabbit	TG75850303
198	652	1/20/2009	3	Harry Hippo	TG75847493
199	652	1/20/2009	3	Henrietta Hippo	TG75847493

FIGURE 7-1 Passenger, flight, and seat information

Looking into the *stats* table for this flight on January, 20, 2009, we see that there are currently 9 seats available in Gold class and 18 seats available in Silver class—that is, three passengers are currently scheduled to fly on that day.

```
mysql> SELECT ClassID, CurrSeats, CurrPrice
    -> FROM stats WHERE FlightID=652
    -> AND FlightDate = '2009-01-20';
+---------+-----------+-----------+
| ClassID | CurrSeats | CurrPrice |
+---------+-----------+-----------+
|       2 |         9 |       200 |
|       3 |        18 |        50 |
+---------+-----------+-----------+
2 rows in set (0.00 sec)
```

With this information at hand, it becomes possible to construct a trigger that automatically handles updating the live seat inventory in the *stats* table. Every time a passenger books a flight, a new record is inserted into the *pax* table. So an `AFTER INSERT` trigger on this table can be used to automatically reduce the seat inventory in the *stats* table by 1 on every record insertion.

Here's the code:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER pax_ai
    -> AFTER INSERT ON pax
    -> FOR EACH ROW
    ->   BEGIN
    ->     UPDATE stats AS s
    ->       SET s.CurrSeats = s.CurrSeats - 1
    ->       WHERE s.FlightID = NEW.FlightID
    ->       AND s.FlightDate = NEW.FlightDate
    ->       AND s.ClassID = NEW.ClassID;
    ->   END//
Query OK, 0 rows affected (0.03 sec)
```

Similarly, every time a cancellation occurs, the corresponding record will be deleted from the passenger manifest, and an `AFTER DELETE` trigger can be used to simultaneously increase the seat inventory by 1:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER pax_ad
    -> AFTER DELETE ON pax
    -> FOR EACH ROW
    ->   BEGIN
    ->     UPDATE stats AS s
    ->       SET s.CurrSeats = s.CurrSeats + 1
    ->       WHERE s.FlightID = OLD.FlightID
    ->       AND s.FlightDate = OLD.FlightDate
    ->       AND s.ClassID = OLD.ClassID;
    ->   END//
Query OK, 0 rows affected (0.01 sec)
```

See this in action by inserting a new passenger record into the *pax* table and then reviewing the *stats* table:

```
mysql> INSERT INTO pax
    -> (FlightID, FlightDate, ClassID, PaxName, PaxRef)
    -> VALUES (652, '2009-01-20', 3,
    -> 'Igor Iguana', 'TR58304888');
Query OK, 1 row affected (0.01 sec)
mysql> SELECT ClassID, CurrSeats, CurrPrice
    -> FROM stats WHERE FlightID=652
    -> AND FlightDate = '2009-01-20';
+---------+-----------+-----------+
| ClassID | CurrSeats | CurrPrice |
+---------+-----------+-----------+
|       2 |         9 |       200 |
|       3 |        17 |        50 |
+---------+-----------+-----------+
2 rows in set (0.00 sec)
```

And if you remove a passenger record, the seat inventory should tick upwards by one.

Automatically increasing (or decreasing) the ticket price as the seat count reduces (or increases) can be accomplished by defining different "slabs" of seat utilization and adjusting the current price upwards or downwards by a fixed percentage depending on the current slab. So, for example, the airline might decide that once 25 percent of the seats in a class are sold, the price should automatically increase by 50 percent. Similarly, once 75 percent of the seats are sold, the price should once again increase by 50 percent.

Adding this logic entails modifying the previously defined triggers, as shown:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER pax_ai
    -> AFTER INSERT ON pax
    -> FOR EACH ROW
    ->   BEGIN
    ->     DECLARE u FLOAT DEFAULT 0;
    ->     DECLARE cs, ms, bp, cp INT DEFAULT 0;
    ->     UPDATE stats AS s
    ->       SET s.CurrSeats = s.CurrSeats - 1
    ->       WHERE s.FlightID = NEW.FlightID
    ->       AND s.FlightDate = NEW.FlightDate
    ->       AND s.ClassID = NEW.ClassID;
    ->     SELECT s.CurrSeats, s.CurrPrice INTO cs, cp
    ->       FROM stats AS s
    ->       WHERE s.FlightID = NEW.FlightID
    ->       AND s.FlightDate = NEW.FlightDate
    ->       AND s.ClassID = NEW.ClassID;
    ->     SELECT fc.MaxSeats, fc.BasePrice INTO ms, bp
    ->       FROM flightclass AS fc
    ->       WHERE fc.FlightID = NEW.FlightID
    ->       AND fc.ClassID = NEW.ClassID;
    ->     SET u = 1 - (cs/ms);
    ->     IF (u >= 0.25 AND u < 0.75 AND cp != ROUND(bp * 1.5)) THEN
    ->       UPDATE stats AS s
    ->         SET s.CurrPrice = ROUND(bp * 1.5)
    ->         WHERE s.FlightID = NEW.FlightID
    ->         AND s.FlightDate = NEW.FlightDate
    ->         AND s.ClassID = NEW.ClassID;
    ->     END IF;
    ->     IF (u >= 0.75 AND cp != ROUND(bp * 2.25)) THEN
    ->       UPDATE stats AS s
    ->         SET s.CurrPrice = ROUND(bp * 2.25)
    ->         WHERE s.FlightID = NEW.FlightID
    ->         AND s.FlightDate = NEW.FlightDate
    ->         AND s.ClassID = NEW.ClassID;
    ->     END IF;
    ->   END//
Query OK, 0 rows affected (0.00 sec)
```

This looks complicated, but it really isn't! The trigger begins by first updating the seat inventory and then retrieving the current seat availability, the maximum seats possible, the current price, and the base price for that particular flight/class combination. It then calculates the seat utilization ratio and updates the current price, depending on whether this ratio is between 25 and 75 percent or greater than 75 percent.

It's also necessary to update the price if passengers cancel their reservation. Here's the revised `AFTER DELETE` trigger:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER pax_ad
    -> AFTER DELETE ON pax
    -> FOR EACH ROW
    ->   BEGIN
    ->     DECLARE u FLOAT DEFAULT 0;
    ->     DECLARE cs, ms, bp, cp INT DEFAULT 0;
    ->     UPDATE stats AS s
    ->       SET s.CurrSeats = s.CurrSeats + 1
    ->       WHERE s.FlightID = OLD.FlightID
    ->       AND s.FlightDate = OLD.FlightDate
    ->       AND s.ClassID = OLD.ClassID;
    ->     SELECT s.CurrSeats, s.CurrPrice INTO cs, cp
    ->       FROM stats AS s
    ->       WHERE s.FlightID = OLD.FlightID
    ->       AND s.FlightDate = OLD.FlightDate
    ->       AND s.ClassID = OLD.ClassID;
    ->     SELECT fc.MaxSeats, fc.BasePrice INTO ms, bp
    ->       FROM flightclass AS fc
    ->       WHERE fc.FlightID = OLD.FlightID
    ->       AND fc.ClassID = OLD.ClassID;
    ->     SET u = 1 - (cs/ms);
    ->     IF (u < 0.25 AND cp != bp) THEN
    ->       UPDATE stats AS s
    ->         SET s.CurrPrice = bp
    ->         WHERE s.FlightID = OLD.FlightID
    ->         AND s.FlightDate = OLD.FlightDate
    ->         AND s.ClassID = OLD.ClassID;
    ->     END IF;
    ->     IF (u >= 0.25 AND u < 0.75 AND cp != ROUND(bp * 1.5)) THEN
    ->       UPDATE stats AS s
    ->         SET s.CurrPrice = ROUND(bp * 1.5)
    ->         WHERE s.FlightID = OLD.FlightID
    ->         AND s.FlightDate = OLD.FlightDate
    ->         AND s.ClassID = OLD.ClassID;
    ->     END IF;
    ->     IF (u >= 0.75 AND cp != ROUND(bp * 2.25)) THEN
    ->       UPDATE stats AS s
    ->         SET s.CurrPrice = ROUND(bp * 2.25)
    ->         WHERE s.FlightID = OLD.FlightID
```

```
    ->         AND s.FlightDate = OLD.FlightDate
    ->         AND s.ClassID = OLD.ClassID;
    ->     END IF;
    ->   END//
Query OK, 0 rows affected (0.00 sec)
```

Let's try it by booking two passengers in Gold class on that flight:

```
mysql> INSERT INTO pax
    -> (FlightID, FlightDate, ClassID, PaxName, PaxRef)
    -> VALUES (652, '2009-01-20', 2,
    -> 'Gerry Giraffe', 'TR75950888');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO pax
    -> (FlightID, FlightDate, ClassID, PaxName, PaxRef)
    -> VALUES (652, '2009-01-20', 2,
    -> 'Adam Anteater', 'TR88404015');
Query OK, 1 row affected (0.00 sec)
```

Since 7 of the 10 available seats are now booked, the 25 percent threshold has been crossed and a price rise should automatically occur. Look in the *stats* table, and you'll see that the ticket price for the flight in Gold class has risen by 50 percent, from $200 to $300.

```
mysql> SELECT ClassID, CurrSeats, CurrPrice
    -> FROM stats WHERE FlightID=652
    -> AND FlightDate = '2009-01-20';
+---------+-----------+-----------+
| ClassID | CurrSeats | CurrPrice |
+---------+-----------+-----------+
|       2 |         7 |       300 |
|       3 |        17 |        50 |
+---------+-----------+-----------+
2 rows in set (0.01 sec)
```

Triggers and Constraints

Now, if you're sharp-eyed, you'll have noticed that there's a glaring problem in the previous example: It's possible to keep adding passengers until the seat inventory falls below zero. While this is theoretically possible in one sense (a negative seat inventory might well be considered overbooking, a fairly common airline practice these days), let's assume that, for our airline at least, showing a negative value for seats available on a flight is a Bad Thing.

This occurs, quite naturally, because while the trigger in the previous example is pretty good at increasing and decreasing the seat inventory in response to passenger bookings and cancellations, it doesn't include any checks that prevent the available seat count falling below zero or rising above the maximum number of seats specified for that class. To make things even more…ahem, airtight, the trigger should be updated to check for these upper and lower limits, and allow the `INSERT` into the *pax* table only if these range constraints are not violated.

And therein lies the problem. Unlike Oracle, which allows you to abort a trigger with the `RAISE APPLICATION ERROR` statement, MySQL does not currently offer any mechanism to abort a trigger or to raise an error in the event that a user-specified constraint is not met. This is a key limitation of MySQL's current implementation of triggers, and has generated a large amount of discussion in the MySQL user forums… as well as a creative workaround!

The fundamental principle of this workaround is simple: Deliberately generate a MySQL error by performing an illegal operation, thereby forcing MySQL to abort execution of the trigger. There are various ways in which this can be done, including:

- Inserting a value into a nonexistent field
- Inserting a `NULL` value into a field with the `NOT NULL` constraint
- Calling a nonexistent stored routine

The end result of all these operations is the same: a fatal error, which will cause MySQL to terminate execution of the statement causing the error. If this statement is enclosed within a `BEFORE` trigger, the resulting error will force MySQL to abort trigger execution, as well as the `INSERT`, `UPDATE`, or `DELETE` statement that is supposed to follow it.

To illustrate this in action, consider the following trivial example: a trigger that only allows new airports to be registered in the *airport* table if they have at least three runways:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER airport_bi
    -> BEFORE INSERT ON airport
    -> FOR EACH ROW
    -> BEGIN
    ->   IF NEW.NumRunways < 3 THEN
    ->     CALL i_dont_exist;
    ->   END IF;
    -> END//
Query OK, 0 rows affected (0.06 sec)
```

Now, try it out:

```
mysql> INSERT INTO airport
    -> (AirportID, AirportCode, AirportName,
    -> CityName, CountryCode, NumRunways,
    -> NumTerminals) VALUES (207, 'LTN',
    -> 'Luton Airport', 'London', 'GB',
    -> 2,1);
ERROR 1305 (42000): PROCEDURE db1.i_dont_exist does not exist
```

In this case, because the specified constraint in the `BEFORE INSERT` trigger isn't met, a deliberate error is generated, which causes the failure of the `INSERT` altogether. On the other hand, if you were to try the same query specifying three or more runways, the `INSERT` statement would execute successfully.

Now, let's use a couple of BEFORE triggers on the *pax* table to enforce the constraints discussed at the beginning of this section:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER pax_bi
    -> BEFORE INSERT ON pax
    -> FOR EACH ROW
    -> BEGIN
    ->   DECLARE cs INT DEFAULT 0;
    ->   SELECT s.CurrSeats INTO cs
    ->     FROM stats AS s
    ->     WHERE s.FlightID = NEW.FlightID
    ->     AND s.FlightDate = NEW.FlightDate
    ->     AND s.ClassID = NEW.ClassID;
    ->   IF cs <= 0 THEN
    ->     SET @trigger_error = 'No seats available';
    ->     CALL i_dont_exist();
    ->   END IF;
    -> END//
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TRIGGER pax_bd
    -> BEFORE DELETE ON pax
    -> FOR EACH ROW
    -> BEGIN
    ->   DECLARE cs, ms INT DEFAULT 0;
    ->   SELECT s.CurrSeats INTO cs
    ->     FROM stats AS s
    ->     WHERE s.FlightID = OLD.FlightID
    ->     AND s.FlightDate = OLD.FlightDate
    ->     AND s.ClassID = OLD.ClassID;
    ->   SELECT fc.MaxSeats INTO ms
    ->     FROM flightclass AS fc
    ->     WHERE fc.FlightID = OLD.FlightID
    ->     AND fc.ClassID = OLD.ClassID;
    ->   IF cs >= ms THEN
    ->     SET @trigger_error = 'Cannot increase seat count';
    ->     CALL i_dont_exist();
    ->   END IF;
    -> END//
Query OK, 0 rows affected (0.01 sec)
```

In this case, whenever one of the range constraints is violated and the trigger aborts, a message indicating the cause of the error will be placed in the `@trigger_error` session variable. This suggestion (which must be again credited to the MySQL forum, which developed the workaround in the first place) allows applications to access a human-readable error message and display it to the user.

Understanding Scheduled Events

The triggers discussed in the previous section are written for, and activated by, a particular type of event, such as a new record insertion or modification. However, MySQL 5.1 also supports a slightly different approach to database automation in the form of scheduled events.

Scheduled events, as the name suggests, are triggered at particular times. They provide a framework to perform one or more SQL operations on a time-based schedule. Scheduled events, like triggers, are always associated with a particular table, and can be set to execute either once or repeatedly at predefined intervals. This can come in handy for tasks that need to take place periodically, such as log rotation, statistics generation, or counter updates.

A Simple Scheduled Event

To understand how scheduled events work, let's consider a simple example: archiving old passenger data. Let's suppose that a database administrator wishes to automatically move all passenger records for flights that are 30 days old out of the *pax* table and into a different archive table. A scheduled event makes this easy to do:

```
mysql> CREATE TABLE paxarchive LIKE pax;
Query OK, 0 rows affected (0.03 sec)
mysql> ALTER TABLE paxarchive ENGINE=ARCHIVE;
Query OK, 0 rows affected (0.12 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DELIMITER //
mysql> CREATE EVENT pax_day
    -> ON SCHEDULE EVERY 1 DAY
    -> STARTS '2009-01-14 22:45:00' ENABLE
    -> DO
    ->   BEGIN
    ->     INSERT INTO paxarchive
    ->       SELECT * FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->     DELETE FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->   END//
Query OK, 0 rows affected (0.01 sec)
```

To define a scheduled event, MySQL offers the `CREATE EVENT` command. This command must be followed by the event name, the event schedule, an active/inactive flag, and the main body, which contains the SQL statements to be executed when the event fires.

These components are illustrated in the previous example, which creates a scheduled event named *paxarchive.* The `ON SCHEDULE EVERY 1 DAY` clause in the event definition ensures that it is activated daily, while the `STARTS` clause specifies the event's start date and time. The `ENABLE` keyword tells the system that this is an active event, while the `DO` clause contains the main body of the trigger; this can contain either a single SQL statement or (as in the previous example) multiple SQL statements enclosed within a `BEGIN...END` block.

Defining an event is not, however, sufficient to have it fire automatically. By default, MySQL's event scheduling engine is deactivated and must be activated with the following command:

```
mysql> SET GLOBAL event_scheduler = ON;
Query OK, 0 rows affected (0.38 sec)
```

This command starts the global event scheduling daemon, which periodically checks for scheduled events and runs them at the appropriate time.

As a result of these actions, MySQL will, on a daily basis, copy all passenger records that relate to flights 30 days in the past to the *paxarchive* table and then delete the same records from the *pax* table.

NOTE *To create a scheduled event, a user must have the* `EVENT` *privilege. To turn the global event scheduler on or off, a user must have the* `SUPER` *privilege. Privileges are discussed in greater detail in Chapter 11.*

To modify a scheduled event, use the `ALTER EVENT` command and provide new parameters for the event. Here's an example, which alters the previous event to run every two hours instead:

```
mysql> DELIMITER //
mysql> ALTER EVENT pax_day
    -> ON SCHEDULE EVERY 2 HOUR
    -> STARTS '2009-01-14 22:45:00' ENABLE
    -> DO
    ->   BEGIN
    ->     INSERT INTO paxarchive
    ->       SELECT * FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->     DELETE FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->   END//
Query OK, 0 rows affected (0.24 sec)
```

Here's another example, which disables a specified event (disabled events will not fire at all):

```
mysql> ALTER EVENT pax_day DISABLE;
Query OK, 0 rows affected (0.00 sec)
```

By default, once an event has completed, it is automatically removed from the event queue by the event scheduler. However, you can manually remove it at any time; use the `DROP EVENT` command with the event name as argument:

```
mysql> DROP EVENT pax_day;
Query OK, 0 rows affected (0.03 sec)
```

TIP *To prevent an event from being automatically removed from the event queue once it is completed (for audit or other reasons), attach an* `ON COMPLETION PRESERVE` *clause to the* `CREATE EVENT` *command.*

Alternatively, to turn off all scheduled events, turn off the global scheduler, as shown:

```
mysql> SET GLOBAL event_scheduler = OFF;
Query OK, 0 rows affected (0.38 sec)
```

To view the body of a specific event, use the `SHOW CREATE EVENT` command with the event name as argument. Here's an example:

```
mysql> SHOW CREATE EVENT pax_day\G
*************************** 1. row ***************************
               Event: pax_day
            sql_mode: STRICT_TRANS_TABLES
           time_zone: SYSTEM
        Create Event: CREATE EVENT `pax_day`
ON SCHEDULE EVERY 1 DAY
STARTS '2009-01-14 22:45:00' ON COMPLETION NOT PRESERVE
ENABLE DO
  BEGIN
    INSERT INTO paxarchive
      SELECT * FROM pax
      WHERE FlightDate <=
      DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    DELETE FROM pax
      WHERE FlightDate <=
      DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
  END
character_set_client: latin1
collation_connection: latin1_swedish_ci
  Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

To view a list of all events scheduled on the server, use the SHOW EVENTS command, as shown:

```
mysql> SHOW EVENTS\G
*************************** 1. row ***************************
                  Db: db1
                Name: pax_day
             Definer: root@localhost
           Time zone: SYSTEM
                Type: RECURRING
          Execute at: NULL
      Interval value: 1
      Interval field: DAY
              Starts: 2009-01-14 22:45:00
                Ends: NULL
              Status: ENABLED
          Originator: 0
character_set_client: latin1
collation_connection: latin1_swedish_ci
  Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

Event Security

The CREATE EVENT command supports a DEFINER clause, which specifies the user account whose privileges should be considered when executing the event code. For the event to execute successfully, this user should have all the privileges necessary to perform the statements listed in the event body. By default, MySQL sets the DEFINER value to the user who created the trigger.

Here's an example:

```
mysql> DELIMITER //
mysql> CREATE DEFINER = 'jack@example.com'
    -> EVENT pax_day
    -> ON SCHEDULE EVERY 1 DAY
    -> STARTS '2009-01-14 22:45:00' ENABLE
    -> DO
    ->   BEGIN
    ->     INSERT INTO paxarchive
    ->       SELECT * FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->     DELETE FROM pax
    ->       WHERE FlightDate <=
    ->       DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
    ->   END//
Query OK, 0 rows affected (0.01 sec)
```

Recurring Events

Let's take a closer look at recurring events. As the previous section illustrated, a recurring event contains the EVERY clause in the event definition; this clause tells MySQL that the event is one that repeats "every XX time units." The EVERY clause also contains the *repeat interval*—typically, this consists of a number and a keyword representing the time unit. Valid time units include YEAR, QUARTER, MONTH, DAY, HOUR, MINUTE, WEEK, and SECOND.

Here's an example, which checks the percentage of seats that have been booked for each flight every hour and logs flights that are more than 80 percent full:

```
mysql> DELIMITER //
mysql> CREATE EVENT util_hour
    -> ON SCHEDULE EVERY 1 HOUR ENABLE
    -> DO
    -> BEGIN
    ->  DECLARE fid INT;
    ->  DECLARE fdate DATE;
    ->  DECLARE str TEXT DEFAULT '';
    ->  DECLARE util FLOAT;
    ->  DECLARE c CURSOR FOR
    ->   SELECT s.FlightID, s.FlightDate, 1-(SUM(s.CurrSeats) /
    ->       (SELECT SUM(fc.MaxSeats)
    ->       FROM flightclass AS fc
    ->       WHERE fc.FlightID = s.FlightID
    ->       GROUP BY FlightID))
    ->     AS u FROM stats AS s
    ->     GROUP BY s.FlightID, s.FlightDate
    ->     HAVING u > 0.80;
    ->   OPEN c;
    ->   l: LOOP
    ->     FETCH c INTO fid,fdate,util;
    ->     SET str = CONCAT('Flight # ', fid, ' on ',
    ->       fdate, ": ", ROUND(util*100), '%');
    ->     INSERT INTO log (ByUser, Note, EventTime)
    ->       VALUES (CURRENT_USER(), str, NOW());
    ->   END LOOP l;
    ->   CLOSE c;
    -> END//
Query OK, 0 rows affected (0.00 sec)
```

CAUTION *Open-ended recurring events that write new data to a table and have no defined end time (like the previous example) are dangerous, because they could cause the target table to grow in size quite quickly, with no end in sight. Avoid using these as much as possible (the previous example is only illustrative and should not be used in a production environment), and if you must do so, always specify an end time and as many additional constraints as possible to limit the event's action.*

You can also configure the event to fire only within a certain time period by specifying optional STARTS and ENDS clauses, which contain the starting and ending times for the event. Here's a revision of the previous example, which configures the event to fire only during a particular month:

```
mysql> DELIMITER //
mysql> CREATE EVENT util_hour
    -> ON SCHEDULE EVERY 1 HOUR
    -> STARTS '2009-04-01 00:00:01'
    -> ENDS '2009-04-30 23:59:01'
    -> ENABLE
    -> DO
    -> BEGIN
    ->  DECLARE fid INT;
    ->  DECLARE fdate DATE;
    ->  DECLARE str TEXT DEFAULT '';
    ->  DECLARE util FLOAT;
    ->  DECLARE c CURSOR FOR
    ->   SELECT s.FlightID, s.FlightDate, 1-(SUM(s.CurrSeats) /
    ->       (SELECT SUM(fc.MaxSeats)
    ->       FROM flightclass AS fc
    ->       WHERE fc.FlightID = s.FlightID
    ->       GROUP BY FlightID))
    ->     AS u FROM stats AS s
    ->     GROUP BY s.FlightID, s.FlightDate
    ->     HAVING u > 0.80;
    ->   OPEN c;
    ->   l: LOOP
    ->     FETCH c INTO fid,fdate,util;
    ->     SET str = CONCAT('Flight # ', fid, ' on ',
    ->       fdate, ": ", ROUND(util*100), '%');
    ->     INSERT INTO log (ByUser, Note, EventTime)
    ->       VALUES (CURRENT_USER(), str, NOW());
    ->   END LOOP l;
    ->   CLOSE c;
    -> END//
Query OK, 0 rows affected (0.01 sec)
```

One-Off Events

Although MySQL's event scheduler is great for setting up recurring events, it also supports events that only fire once, at a predefined time and date. To set up such an event, replace the EVERY clause in the CREATE EVENT statement with an AT clause that contains the date and time at which the event should fire. Here's an example, which sets up an event to fire at 1:25 A.M. on April 1, 2009:

```
mysql> CREATE EVENT log_onetime
    -> ON SCHEDULE AT '2009-04-01 01:25' ENABLE
    -> DO
    -> INSERT INTO log (ByUser, Note, EventTime)
    -> VALUES (CURRENT_USER(), 'Updating all accounts', NOW());
Query OK, 0 rows affected (0.50 sec)
```

TIP *To force an event to fire at the instant it is created, use the NOW() function in the AT clause instead of a timestamp.*

Summary

This chapter focused on database automation, explaining how database triggers and scheduled events can be used to easily perform operations that would otherwise need separate application-level workflows and/or integration with scheduling agents such as *cron*. Utilizing simple applications, it showed you how to construct various types of triggers, schedule events for either one-time or repeated execution, and build in complex programming logic using the conditional tests, loops, and cursors discussed in the previous chapter.

To learn more about the topics discussed in this chapter, consider visiting the following links:

- Triggers, at http://dev.mysql.com/doc/refman/5.1/en/create-trigger.html and http://forge.mysql.com/wiki/Triggers
- Scheduled events, at http://dev.mysql.com/doc/refman/5.1/en/events-overview.html
- Key limitations on triggers and scheduled events, at http://dev.mysql.com/doc/refman/5.1/en/stored-program-restrictions.html
- A MySQL forum discussion of raising errors inside triggers, at http://forums.mysql.com/read.php?99,55108,55108#msg-55108 and http://rpbouman.blogspot.com/2005/11/using-udf-to-raise-errors-from-inside.html