## ALTER EVENT

```
ALTER EVENT
[DEFINER = {'user'@'host'|CURRENT_USER}]
event
ON SCHEDULE
AT timestamp [+ INTERVAL count interval] |
EVERY count interval
   [STARTS timestamp [+ INTERVAL count interval]]
   [ENDS timestamp [+ INTERVAL count interval]]
[ON COMPLETION [NOT] PRESERVE]
[ENABLE|DISABLE|DISABLE ON SLAVE]
[COMMENT 'comment']
DO statement
```

Use this statement to alter an existing scheduled MySQL event. The statement can be used to change the time when the scheduled SQL statement will execute or other aspects of its upcoming execution. The event parameter has to be the name of an event that was already scheduled but has not yet been completed, or was completed but preserved by the server. It isn't possible within MySQL to change the name of an event. Instead, use the `DROP EVENT` statement to delete an existing event and then create it again with a new name using `CREATE EVENT`. You can use the `SHOW CREATE EVENT` statement to be sure that all other parameters are the same.

To change the MySQL user and host through which MySQL executes the event, use the `DEFINER` clause. As of version 5.1.12 of MySQL, a user that has `EVENT` privilege can change an event. Unless the definer is specified with the `DEFINER` clause, the user that changes an event becomes the new definer.

To change the time and date that form the basis for running the event, use the `ON SCHEDULE AT` clause and give the new time in the `timestamp` format (*yyyy-mm-dd hh:mm:ss*). The time given can be a string, a time function, or just `CURRENT_TIMESTAMP`. You can also specify a time relative to the timestamp given by adding a plus sign followed by the keyword `INTERVAL`, the number of intervals (e.g., `1`), and then the interval increment (e.g., `HOUR`). For *interval*, use one of the allowable intervals shown in the description of the `CREATE EVENT` statement later in this chapter.

To make the event a recurring one, add the `EVERY` clause, using the same syntax and format. You can also give starting and ending times for a recurring event with the `STARTS` and `ENDS` clauses.

If an event is not yet completed, you can keep the server from dropping it by adding the `ON COMPLETION` clause with the `PRESERVE` keyword. If you already did this when you created the event, you can change your mind and set the server to `NOT PRESERVE` the event.

If you created an event that you need to temporarily disable for some reason, you can do so with this statement by using the `DISABLE` keyword. An event that has been disabled can be enabled with the `ENABLE` keyword. The `DISABLE ON SLAVE` keyword prevents the event from running on slave servers.

With the `COMMENT` clause, you can add or change a comment describing the event for future reference. The `DO` clause can include any SQL statement to be executed. A stored procedure can be used to easily execute a set of SQL statements.

Here is an example using this statement to change a periodic event:

```
ALTER EVENT students_copy
ON SCHEDULE EVERY 1 DAY
STARTS '2007-12-10 01:30:00'
ON COMPLETION PRESERVE;
```

If you look at the example for CREATE EVENT later in this chapter, you'll see that our only change is to move the time from 2:30 A.M. to 1:30 A.M. here. The starting time and date given are not only for the time we want, but since this statement is run on December 9, the date of December 10 is given. When an event's time is altered or when an event is first created, it must be for a future time. The EVERY clause is included because STARTS is part of it and not a separate clause of its own. So that the ON COMPLETION PRESERVE isn't set back to the default of ON COMPLETION NOT PRESERVE, we stipulate it again here.

## ALTER FUNCTION

```
ALTER FUNCTION stored_procedure
  [{CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA} |
    SQL SECURITY {DEFINER|INVOKER} |
    COMMENT 'string']
```

This statement changes the characteristics of an existing user-defined function. You cannot change the function itself with it. To do that, you need to delete the function with DROP FUNCTION and create a new procedure with CREATE FUNCTION. See the description of CREATE FUNCTION later in this chapter for an explanation of each characteristic.

There are three types of characteristics you can set or change with this statement: the types of interaction with the server, the user recognized for SQL security, and a comment. Each type may be given in a space-separated list, in any order. See CREATE FUNCTION later in this chapter for a discussion of the characteristics. The COMMENT clause replaces any existing comment. To clear a comment without inserting another, give two quotes with nothing between them.

This statement requires the CREATE ROUTINE privilege. The ALTER ROUTINE and EXECUTE privileges are granted to the user and host account that creates or alters a function, by default.

Here is an example using this statement, in which a function shown in the example for the CREATE FUNCTION statement is altered:

```
ALTER FUNCTION date_reformatted
SQL SECURITY INVOKER
COMMENT "Converts a string date like 'Dec. 7, 2007' to standard format.";
```

## ALTER PROCEDURE

```
ALTER PROCEDURE stored_procedure
  [{CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA}]
  [SQL SECURITY {DEFINER|INVOKER}]
  [COMMENT 'string']
```

This statement changes the characteristics of an existing stored procedure. You cannot change the procedure itself with it. To do that, you need to delete the procedure with

DROP PROCEDURE and create a new procedure with CREATE PROCEDURE. See the description of CREATE PROCEDURE later in this chapter for an explanation of each characteristic.

There are three types of characteristics that you can set or change with this statement: the types of interaction with the server, the user recognized for SQL security, and a comment. Each type may be given in a space-separated list, in any order. See CREATE PROCEDURE later in this chapter for a discussion of the characteristics. The COMMENT clause replaces any existing comment. To clear a comment without inserting another, give two quotes with nothing between them.

This statement requires the CREATE ROUTINE privilege. The ALTER ROUTINE and EXECUTE privileges are granted to the user and host account that creates or alters a stored procedure, by default.

Here is an example of this statement:

```
ALTER PROCEDURE students_copy_proc
SQL SECURITY INVOKER
COMMENT 'Copies data from students table to students_backup.
Add a comment with @ref_note.'
```

If you look at the example for CREATE PROCEDURE later in this chapter, you'll see that the example here is changing the procedure created in that example. We're only adding that the user account to be used for executing the procedure will be the invoker, and we're adding a comment about the procedure—we didn't include one when we created the procedure.

## ALTER TRIGGER

There is not an ALTER TRIGGER statement at this time. Instead, use the DROP TRIGGER statement and then CREATE TRIGGER again with the new, adjusted trigger.

## BEGIN...END

BEGIN...END

Use this combination of statements to start and end the steps that are part of a stored procedure or trigger. In essence, BEGIN marks the beginning of a compound SQL statement and END marks the end of it. Multiple SQL statements can be included between them.

Traditionally, as you know from using the *mysql* client, each SQL statement must end with a semicolon. However, semicolons must be used within CREATE PROCEDURE and CREATE TRIGGER statements to separate the internal statements that form the procedure or trigger. So as not to confuse the parser in the client and server, include a DELIMITER command to change the default delimiter to another character before entering BEGIN, and then to set it back to a semicolon again after entering END. For examples of these statements, see the CREATE PROCEDURE and CREATE TRIGGER statements later in this chapter.

## CALL

```
CALL stored_procedure[([parameter[, ...]])]
```

Use this statement to call a stored procedure. Parameters to be passed to the stored procedure may be given within the parentheses. If the keyword of INOUT is used, values may be given to the stored procedure and returned to the SQL statement that called it. For an example of this statement, see the CREATE PROCEDURE statement later in this chapter.

## CLOSE

```
CLOSE cursor
```

This statement closes a cursor that has been declared within the current routine and has been opened using the OPEN statement. See the descriptions of the DECLARE and FETCH statements later in this chapter for more information on cursors.

## CREATE EVENT

```
CREATE [DEFINER = {'user'@'host'|CURRENT_USER}] EVENT
[IF NOT EXISTS] event
ON SCHEDULE
AT timestamp [+ INTERVAL count interval] |
EVERY count interval
   [STARTS timestamp [+ INTERVAL count interval]]
   [ENDS timestamp [+ INTERVAL count interval]]
[ON COMPLETION [NOT] PRESERVE]
[ENABLE|DISABLE|DISABLE ON SLAVE]
[COMMENT 'comment']
DO statement
```

Use this statement to schedule the execution of an SQL statement at a specific time and date. Events may also be recurring. Although there are many options, the basic syntax is:

```
CREATE EVENT event ON SCHEDULE AT timestamp DO statement
```

The event name you give may be any nonreserved word and is case-insensitive. The DO clause can include any SQL statement to be executed. A stored procedure can be passed here to conveniently execute a set of SQL statements.

With the DEFINER clause, you can specify the MySQL user and host to be used by MySQL for the event. This means that the event may be created by a user with SUPER privileges but executed by another user account in which privileges are limited for security reasons. The IF NOT EXISTS clause may be given to prevent errors from being returned if the event has already been created.

For the required ON SCHEDULE AT clause, include a specific time and date in the timestamp format (*yyyy-mm-dd hh:mm:ss*). The time given can be a string, a time function, or just CURRENT_TIMESTAMP. You can also specify a time relative to the timestamp given by adding a plus sign followed by the keyword INTERVAL, the number of intervals (e.g., 1), and then the interval increment (e.g., HOUR). For *interval*, use one of the allowable intervals: SECOND, MINUTE, MINUTE_SECOND, HOUR, HOUR_SECOND, HOUR_MINUTE, DAY, DAY_SECOND, DAY_MINUTE, DAY_HOUR, WEEK, MONTH, QUARTER, YEAR, or YEAR_MONTH.

To make the event a recurring one, add the EVERY clause, using the same syntax and format. You can also give starting and ending times for a repeating event with the STARTS and ENDS clauses.

Once an event is completed, it will be dropped automatically. However, you can drop it manually before completion with the DROP EVENT statement. You can also keep the server from dropping an event by adding the ON COMPLETION clause with the PRESERVE keyword. The NOT PRESERVE keyword instructs the server not to retain the event when completed; this is the server's default behavior.

When creating an event, you may want to create it with the DISABLE parameter so that it won't begin to execute until you enable it. Then use the ALTER EVENT statement to enable it later. The DISABLE ON SLAVE keyword will disable the event from running on slave servers. By default, an event runs on the master and all slaves.

With the COMMENT clause, you can add a comment describing the event for future reference. This comment is displayed only when SHOW CREATE EVENT is executed for the event.

Here is an example using this statement. It schedules a procedure that is created in the example under the CREATE PROCEDURE statement later in this chapter:

```
CREATE EVENT students_copy
ON SCHEDULE EVERY 1 DAY
STARTS '2007-11-27 02:30:00'
ON COMPLETION PRESERVE
COMMENT 'Daily copy of students table to students_backup'
DO CALL students_copy_proc();
```

In this example, the event will be run once a day starting from the time given and then every day afterward at the same time (2:30 A.M.). It's set to be recurring, but in case someone ever changes that aspect of it, MySQL will preserve the event upon completion. We've added a comment to explain the purpose of the event. Use ALTER EVENT to change an event and SHOW EVENTS to get a list of events.

## CREATE FUNCTION

```
CREATE
[DEFINER = {'user'@'host'|CURRENT_USER}]
FUNCTION function ([parameter data_type[,...]])
RETURNS data_type
  [LANGUAGE SQL]
  [[NOT] DETERMINISTIC]
  [{CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA}]
  [COMMENT 'string']
  [SQL SECURITY {DEFINER|INVOKER}]
RETURN routine
```

A user-defined function is essentially a set of SQL statements that may be called as a unit, processing any data it's given in its parameters and returning a value to the caller of the function. This is similar to a stored procedure, except that a function returns a value and a stored procedure does not. A stored procedure normally places the values it generates in user variables that can then be retrieved in various ways.

The basic, minimum syntax is something like this:

```
CREATE FUNCTION function_name (parameter) RETURNS INT RETURN routine
```

The function name given can be any nonreserved name; don't use the name of a built-in function. The name is case-insensitive. Within parentheses, give a comma-separated list of the parameters. For each parameter, specify the data type to be used (`INT`, `CHAR`, etc.). The keyword `RETURNS` is followed by the data type of the value that will be returned by the function. At the end comes the keyword `RETURN` followed by the routine to perform.

You may provide special parameters to indicate the characteristics of the function. Several may be given in any order, in a space-separated list. You can specify the language used as SQL with the `LANGUAGE SQL` parameter, but this is the default and usually unnecessary.

A function that returns the same results each time for the same given parameters is considered *deterministic*. You can save processing time on the server by specifying this property through the `DETERMINISTIC` parameter. `NOT DETERMINISTIC` is the default.

The following keywords may be used to tell the server how the function will interact with it, allowing the server to optimize the function. The server does not enforce the restrictions on the function, however:

`CONTAINS SQL`
: The function executes SQL statements, but does not read from or write to a table; one example is a function that queries server status. This is the default.

`NO SQL`
: The function does not contain any SQL statements.

`READS SQL DATA`
: The function might read data from at least one table, but it doesn't write data to any tables.

`MODIFIES SQL DATA`
: The function might write data to at least one table, as well as potentially read data from tables.

With the `COMMENT` clause, you can add a comment describing the function for future reference.

This statement requires the `CREATE ROUTINE` privilege. The `ALTER ROUTINE` and `EXECUTE` privileges are granted to the user and host account that creates or alters a routine, by default. With the `DEFINER` clause, you can specify the MySQL user and host to be used by MySQL for the function. Related to this clause is `SQL SECURITY` keyword, which instructs MySQL to use either the user account of the creator (`DEFINER`) of the function or the account that's calling the function (`INVOKER`). This can help to prevent some users from accessing restricted functions.

Here is an example using this statement:

```
CREATE FUNCTION date_reformatted (new_date VARCHAR(13))
RETURNS DATE
RETURN STR_TO_DATE(REPLACE(new_date, '.', ''), '%b %d, %Y');

SELECT date_reformatted('Dec. 7, 2007')
AS proper_date;

+-------------+
| proper_date |
+-------------+
```

```
| 2007-12-07  |
+-------------+
```

This function simply uses the `STR_TO_DATE( )` function to convert a string to a particular date format (i.e., `yyyy-mm-dd`) based on a common string that users may give. It expects the data given to be no more than 13 characters long. Because some users may include a period after the abbreviated month name and some may not, the function uses the `REPLACE( )` function to remove the period. A function like this one can be used in any type of statement (e.g., an `UPDATE` statement to set a column value).

To change an existing user-defined function, use the `ALTER FUNCTION` statement. The `DROP FUNCTION` statement removes a user-defined function..You cannot change standard, built-in functions.

## CREATE PROCEDURE

```
CREATE
[DEFINER = {'user'@'host'|CURRENT_USER}]
PROCEDURE stored_procedure ([[IN|OUT|INOUT] parameter data_type[,...]])
  [LANGUAGE SQL]
  [NOT] DETERMINISTIC]
  [{CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA}]
  [COMMENT 'string']
  [SQL SECURITY {DEFINER|INVOKER}]
routine
```

A *procedure*, also known as a *stored procedure*, is a set of SQL statements stored on the server and called as a unit, processing any data it's given in its parameters. A procedure may communicate results back to the user by placing the values it generates in user variables that can then be retrieved in various ways.

The basic, minimum syntax is something like this:

```
CREATE PROCEDURE procedure_name (IN parameter INT) SQL_statements
```

The procedure name given can be any nonreserved name, and is case-insensitive. Within parentheses, give a comma-separated list of the parameters that will take data in (`IN`), return data (`OUT`), or do both (`INOUT`). For each parameter, specify the data type to be used (`INT`, `CHAR`, etc.).

You may provide special parameters to indicate the characteristics of the stored procedure. Several may be given in any order, in a space-separated list. You can specify the language used as SQL with the `LANGUAGE SQL` parameter, but this is the default and usually unnecessary.

A procedure that returns the same results each time for the same given parameters is considered *deterministic*. You can save processing time on the server by specifying this property through the `DETERMINISTIC` parameter. `NOT DETERMINISTIC` is the default.

The following keywords may be used to tell the server how the procedure will interact with it, allowing the server to optimize the procedure. The server does not enforce the restrictions on the procedure, however:

`CONTAINS SQL`
: The procedure executes SQL statements, but does not read from or write to a table; one example is a procedure that queries server status. This is the default.

NO SQL
: The procedure does not contain any SQL statements.

READS SQL DATA
: The procedure might read data from at least one table, but it doesn't write data to any tables.

MODIFIES SQL DATA
: The procedure might write data to at least one table, as well as potentially read data from tables.

With the COMMENT clause, you can add a comment describing the procedure for future reference.

This statement requires the CREATE ROUTINE privilege. The ALTER ROUTINE and EXECUTE privileges are granted to the user and host account that creates or alters a routine, by default. With the DEFINER clause, you can specify the MySQL user and host to be used by MySQL for the procedure. Related to this clause is the SQL SECURITY keyword, which instructs MySQL to use either the user account of the creator (DEFINER) of the procedure or the account that's executing the procedure (INVOKER). This can help prevent some users from accessing restricted procedures.

In the following example, we create a simple procedure that copies all of the data from the students table to a backup table with the same schema. The table also includes an extra column in which the user can add a comment or reference note:

```
DELIMITER |

CREATE PROCEDURE students_copy_proc (IN ref_note VARCHAR(255))
BEGIN
REPLACE INTO students_backup
SELECT *, ref_note FROM students;
END|

DELIMITER ;

SET @ref_note = '2008 Spring Roster';

CALL students_copy_proc(@ref_note);
```

The first statement changes the terminating character for an SQL statement from its default, a semicolon, to a vertical bar. See the BEGIN...END statement earlier in this chapter for the reasons this is necessary.

Inside the procedure, the REPLACE statement selects all columns from students along with the value of the ref_note variable. Thus, every row of students is inserted, along with the value of the variable, into a new row in students_backup.

After the procedure is defined and the delimiter is changed back to a semicolon, the example sets a variable called ref_note that contains a note the user wants added to each row of data in the new table. This variable is passed to the CALL statement that runs the procedure.

To change an existing stored procedure, use the ALTER PROCEDURE statement. The DROP PROCEDURE statement removes a procedure.

## CREATE TRIGGER

```
CREATE
[DEFINER = {'user'@'host'|CURRENT_USER}]
TRIGGER trigger {AFTER|BEFORE}
{DELETE|INSERT|UPDATE}
ON table FOR EACH ROW statement
```

- Only one of each trigger timing and trigger event combination is allowed for each table. For example, a table cannot have two `BEFORE INSERT` triggers, but it can have a `BEFORE INSERT` and an `AFTER INSERT` trigger.
- To specify that the trigger be executed immediately before the associated user statement, use the parameter `BEFORE`; to indicate that the trigger should be executed immediately afterward, use `AFTER`.
- At this time, only three types of SQL statements can cause the server to execute a trigger: insertions, deletions, and updates. Specifying `INSERT`, however, applies the trigger to `INSERT` statements, `LOAD DATA` statements, and `REPLACE` statements—all statements that are designed to insert data into a table. Similarly, specifying `DELETE` includes both `DELETE` and `REPLACE` statements because `REPLACE` potentially deletes rows as well as inserting them.
- Triggers are actions to be taken when a user requests a change to data. Each trigger is associated with a particular table and includes definitions related to *timing* and *event*. A trigger timing indicates when a trigger is to be performed (i.e., `BEFORE` or `AFTER`). A trigger event is the action that causes the trigger to be executed (i.e., a `DELETE`, `INSERT`, or `UPDATE` on a specified table).
- After specifying the trigger event, give the keyword `ON` followed by the table name. This is followed by `FOR EACH ROW` and the SQL statement to be executed when the trigger event occurs. Multiple SQL statements to execute may be given in the form of a compound statement using `BEGIN...END`, which is described earlier in this chapter.
- There is no `ALTER TRIGGER` statement at this time. Instead, use the `DROP TRIGGER` statement and then reissue `CREATE TRIGGER` with the new trigger.
- To show how a trigger may be created, suppose that for a college database, whenever a student record is deleted from the `students` table, we want to write the data to another table to preserve that information. Here is an example of how that might be done with a trigger:

```
DELIMITER |

CREATE TRIGGER students_deletion
BEFORE DELETE
ON students FOR EACH ROW

BEGIN
INSERT INTO students_deleted
(student_id, name_first, name_last)
VALUES(OLD.student_id, OLD.name_first, OLD.name_last);
END|

DELIMITER ;
```

The first statement changes the terminating character for an SQL statement from its default, a semicolon, to a vertical bar. See the BEGIN...END statement earlier in this chapter for the reasons this is necessary.

Next, we create a trigger to stipulate that, before making a deletion in the students table, the server must perform the compound SQL statement given. The statements between BEGIN and END will write the data to be deleted to another table with the same schema.

To capture that data and pass it to the INSERT statement, we use the OLD table alias provided by MySQL coupled with the column names of the table where the row is to be deleted. OLD refers to the table in the trigger's ON clause, before any changes are made by the trigger or the statement causing the trigger. To save space, in this example we're capturing the data from only three of the columns. OLD.* is not allowed, so we have to specify each column. To specify the columns after they are inserted or updated, use NEW as the table alias.

The statement to be executed by the trigger in the previous example is a compound statement. It starts with BEGIN and ends with END and is followed by the vertical bar (|) that we specified as the delimiter. The delimiter is then reset in the last line back to a semicolon.

## DECLARE

```
DECLARE variable data_type [DEFAULT value]

DECLARE condition CONDITION FOR
{SQLSTATE [VALUE] value | error_code]

DECLARE cursor CURSOR FOR SELECT...

DECLARE {CONTINUE|EXIT|UNDO} HANDLER FOR
  {[SQLSTATE [VALUE] value]
   [SQLWARNING]
   [NOT FOUND]
   [SQLEXCEPTION]
   [error_code]
   [condition]}
SQL_statement
```

This statement declares local variables and other items related to routines. It must be used within a BEGIN...END compound statement of a routine, after BEGIN and before any other SQL statements. There are four basic uses for DECLARE: to declare local variables, conditions, cursors, and handlers. Within a BEGIN...END block, variables and conditions must be declared before cursors and handlers, and cursors must be declared before handlers.

The first syntax shows how to declare variables. It includes the data type and, optionally, default values. A variable declared with this statement is available only within the routine in which it is declared. If the default is a string, place it within quotes. If no default is declared, NULL is the default value.

A condition is generally either an SQLSTATE value or a MySQL error code number. The second syntax is used for declaring a condition and associating it with an SQLSTATE or

an error code. When declaring a condition based on an SQLSTATE, give the SQLSTATE VALUE clause followed by the state. Otherwise, give the error code number.

The third syntax declares a cursor, which represents—within a procedure—a results set that is retrieved one row at a time. Give a unique, nonreserved word for the cursor's name. This is followed by CURSOR FOR and then a SELECT statement. It must not have an INTO clause. To call or open a cursor, use the OPEN statement within the same routine in which the declaration was made. To retrieve data from a cursor, which is done one row at a time, use the FETCH statement. When finished, use the CLOSE statement to close an open cursor.

The last syntax for this statement declares a handler. With a handler, you can specify an SQL statement to be executed given a specific condition that occurs within a routine. Three types of handlers are allowed: CONTINUE, EXIT, and UNDO. Use CONTINUE to indicate that the routine is to continue after the SQL statement given is executed. The EXIT parameter indicates that the BEGIN...END compound statement that contains the declaration should be exited when the condition given is met. UNDO is meant to instruct MySQL to undo the compound statement for which it is given. However, this parameter is not yet supported by MySQL.

The handler's FOR clause may contain multiple conditions in a comma-separated list. There are several related to the SQLSTATE: you can specify a single SQLSTATE code number, or you can list SQLWARNING to declare any SQLSTATE code starting with 01, NOT FOUND for any SQLSTATE code starting with 02, or SQLEXCEPTION for all states that don't start with 01 or 02. Another condition you can give is a MySQL error code number. You can also specify the name of a condition you previously created with its own DECLARE statement.

## DELIMITER

```
DELIMITER character
```

This statement changes the delimiter (terminating character) of SQL statements from the default of a semicolon to another character. This is useful when creating a stored procedure or trigger, so that MySQL does not confuse a semicolon contained in the procedure or trigger as the end of the CREATE PROCEDURE or CREATE TRIGGER statement. This statement is also used to restore the default delimiter. Don't use the backslash as the delimiter, as that is used to escape special characters. Examples of this statement appear in the CREATE PROCEDURE and CREATE TRIGGER statements earlier in this chapter.

## DROP EVENT

```
DROP EVENT [IF EXISTS] event
```

This statement deletes an event. The IF EXISTS keyword prevents error messages when the event doesn't exist. Instead, a note will be generated, which can be displayed afterward by executing the SHOW WARNINGS statement. As of version 5.1.12 of MySQL, this statement requires the EVENT privilege.

## DROP FUNCTION

```
DROP FUNCTION [IF EXISTS] function
```

Use this statement to delete a user-defined function. The IF EXISTS keyword prevents error messages when the function doesn't exist. Instead, a note will be generated, which can be displayed afterward by executing the SHOW WARNINGS statement. This statement requires the ALTER ROUTINE privilege for the function given, which is automatically granted to the creator of the function.

## DROP PREPARE

```
{DROP|DEALLOCATE} PREPARE statement_name
```

This statement deletes a prepared statement. The syntax of DROP PREPARE and DEALLOCATE PREPARE are synonymous. For an example, see the PREPARE statement later in this chapter.

## DROP PROCEDURE

```
DROP PROCEDURE [IF EXISTS] procedure
```

This statement deletes a stored procedure. The IF EXISTS keyword prevents error messages when the stored procedure doesn't exist. Instead, a note will be generated, which can be displayed afterward by executing the SHOW WARNINGS statement. This statement requires the ALTER ROUTINE privilege for the stored procedure given, which is automatically granted to the creator of the stored procedure.

## DROP TRIGGER

```
DROP TRIGGER [IF EXISTS] [database.]trigger
```

This statement deletes a trigger. The IF EXISTS keyword prevents error messages when the trigger doesn't exist. Instead, a note will be generated, which can be displayed afterward by executing the SHOW WARNINGS statement. You may specify the database or schema with which the trigger is associated. If not given, the current default database is assumed. As of version 5.1.6 of MySQL, this statement requires the TRIGGER privilege for the table related to the trigger given. Previously, it required SUPER privilege. When upgrading from version 5.0.10 or earlier of MySQL, be sure to drop all triggers because there's a problem with using or dropping triggers from earlier versions.

## EXECUTE

```
EXECUTE statement_name [USING @variable[, ...] ...]
```

This statement executes a user-defined prepared statement. If the prepared statement contains placeholders so that you can pass parameters to it, these parameters must be given in the form of user-defined variables. Multiple variables may be given in a comma-separated list. You can use the SET statement to set the value of a variable. See the PREPARE statement later in this chapter for an example of the EXECUTE statement's use.

## FETCH

```
FETCH cursor INTO variable[, ...]
```

A cursor is similar to a table or a view: it represents, within a procedure, a results set that is retrieved one row at a time using this statement. You first establish a cursor with the DECLARE statement. Then you use the OPEN statement to initialize the cursor. The FETCH statement retrieves the next row of the cursor and places the data retrieved into one or more variables. There should be the same number of variables as there are columns in the underlying SELECT statement of the cursor. Variables are given in a comma-separated list. Each execution of FETCH advances the pointer for the cursor by one row. Once all rows have been fetched, an SQLSTATE of 02000 is returned. You can tie a condition to this state through a DECLARE statement and end fetches based on the condition. Use the CLOSE statement to close a cursor.

## OPEN

```
OPEN cursor
```

This statement opens a cursor that has been declared within the current routine. Data selected with the cursor is accessed with the FETCH statement. The cursor is closed with the CLOSE statement. See the descriptions of the DECLARE and FETCH statements earlier in this chapter for more information on cursors.

## PREPARE

```
PREPARE statement_name FROM statement
```

This statement creates a prepared statement. A prepared statement is used to cache an SQL statement, so as to save processing time during multiple executions of the statement. This can potentially improve performance. Prepared statements are local to the user and session; they're not global. The name given can be any nonreserved name and is case-insensitive. The statement given within quotes can be any type of SQL statement.

If you want to include a value that will be changed when the statement is executed, give a question mark as a placeholder within *statement*. When the prepared statement is executed later with the EXECUTE statement, the placeholders will be replaced with the values given. The values must be user variables (set with the SET statement) and must be passed to the EXECUTE statement in the order that the placeholders appear in the prepared statement. Here is a simple example using these statements:

```
PREPARE state_tally
FROM 'SELECT COUNT(*)
      FROM students
      WHERE home_city = ?';

SET @city = 'New Orleans';
EXECUTE state_tally USING @city;

SET @city = 'Boston';
EXECUTE state_tally USING @city;
```

In this example, the query within the prepared statement will return a count of the number of students from the city given. By setting the value of the user-defined variable

@city to another city, we can execute the prepared statement state_tally again without having to reenter the PREPARE statement. The results will probably be different, of course. To remove a prepared statement from the cache, use the DROP PREPARE statement.

## SHOW CREATE EVENT

SHOW CREATE EVENT *event*

This statement displays an SQL statement that can be used to create an event like the one given. It's mostly useful for displaying any comments associated with the event because they're not included in the results of the SHOW EVENTS statement.

Here is an example showing an event that was created with the CREATE EVENT statement earlier in this chapter:

```
SHOW CREATE EVENT students_copy \G

*************************** 1. row ***************************
       Event: students_copy
    sql_mode:
Create Event: CREATE EVENT `students_copy` ON SCHEDULE
EVERY 1 DAY ON COMPLETION PRESERVE ENABLE
COMMENT 'Daily copy of students table to students_backup'
DO CALL students_copy_proc()
```

## SHOW CREATE FUNCTION

SHOW CREATE FUNCTION *function*

This statement displays an SQL statement that can be used to create a function like the one given. It's useful for displaying the SQL statements that are performed by the function.

Here is an example of a function that was created with the CREATE FUNCTION statement earlier in this chapter:

```
SHOW CREATE FUNCTION date_reformatted \G

*************************** 1. row ***************************
       Function: date_reformatted
       sql_mode:
Create Function: CREATE DEFINER=`root`@`localhost`
FUNCTION `date_reformatted`(new_date VARCHAR(12))
RETURNS date
SQL SECURITY INVOKER
COMMENT 'Converts a string date like ''Dec. 7, 2007'' to standard format.'
RETURN STR_TO_DATE(REPLACE(new_date, '.', ''), '%b %d, %Y')
```

## SHOW CREATE PROCEDURE

SHOW CREATE PROCEDURE *procedure*

This statement displays an SQL statement that can be used to create a stored procedure like the one given. It's useful for displaying the SQL statements that are performed by the stored procedure.

Here is an example of a procedure that was created with the CREATE PROCEDURE statement earlier in this chapter:

```
SHOW CREATE PROCEDURE students_copy_proc \G

*************************** 1. row ***************************
       Procedure: students_copy_proc
        sql_mode:
Create Procedure: CREATE DEFINER=`root`@`localhost`
PROCEDURE `students_copy_proc`(IN ref_note VARCHAR(255))
BEGIN
REPLACE INTO students_backup
SELECT *, ref_note FROM students;
END
```

## SHOW EVENTS

```
SHOW EVENTS [FROM database] [LIKE 'pattern'|WHERE expression]
```

This statement displays a list of scheduled events on the server. The results can also include events that have been completed but were preserved. The database to which events are related may be given in the FROM clause; the default is the current database. The LIKE or WHERE clauses can be used to list events based on a particular naming pattern. With the WHERE clause, you can use the names of fields in the results to create an expression that sets a condition determining the results returned. An example of this follows. See CREATE EVENT earlier in this chapter for more information on events:

```
 SHOW EVENTS FROM college
WHERE Definer='russell@localhost' \G

*************************** 1. row ***************************
           Db: college
         Name: students_copy
      Definer: russell@localhost
         Type: RECURRING
   Execute at: NULL
Interval value: 1
Interval field: DAY
       Starts: 2007-11-27 02:30:00
         Ends: NULL
       Status: ENABLED
```

## SHOW FUNCTION CODE

```
SHOW FUNCTION CODE function
```

This statement displays the internal code of a function. It requires that the MySQL server be built with debugging. This statement was introduced in version 5.1.3 of MySQL.

## SHOW FUNCTION STATUS

```
SHOW FUNCTION STATUS [LIKE 'pattern'|WHERE expression]
```

This statement displays information on user-defined functions. The LIKE or WHERE clauses can be used to list functions based on a particular naming pattern. With the WHERE clause, you can use the names of fields in the results to create an expression that sets a condition determining the results returned. Here is an example using this statement:

```
SHOW FUNCTION STATUS
WHERE Name='date_reformatted' \G

*************************** 1. row ***************************
           Db: college
         Name: date_reformatted
         Type: FUNCTION
      Definer: root@localhost
     Modified: 2007-11-27 11:55:00
      Created: 2007-11-27 11:47:37
Security_type: INVOKER
      Comment: Converts a string date like 'Dec. 7, 2007' to standard format.
```

## SHOW PROCEDURE CODE

```
SHOW PROCEDURE CODE stored_procedure
```

This statement displays the internal code of a stored procedure. It requires that the MySQL server be built with debugging. This statement was introduced in version 5.1.3 of MySQL.

## SHOW PROCEDURE STATUS

```
SHOW PROCEDURE STATUS [LIKE 'pattern'|WHERE expression]
```

This statement displays information on stored procedures. The LIKE or WHERE clauses can be used to list stored procedures based on a particular naming pattern. With the WHERE clause, you can use the names of fields in the results to create an expression that sets a condition determining the results returned. Here is an example using this statement:

```
SHOW PROCEDURE STATUS
WHERE Name='students_copy_proc' \G

*************************** 1. row ***************************
           Db: college
         Name: students_copy_proc
         Type: PROCEDURE
      Definer: russell@localhost
     Modified: 2007-11-27 09:27:42
      Created: 2007-11-27 09:27:42
Security_type: DEFINER
      Comment:
```

Note that for the WHERE clause we use the field name to get the specific stored procedure.

## SHOW TRIGGERS

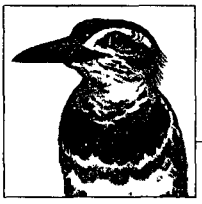
```
SHOW TRIGGERS STATUS [FROM database]
[LIKE 'pattern'|WHERE expression]
```

This statement displays a list of triggers on the server. The database to which triggers are related may be given in the FROM clause; the default is the current database. The LIKE or WHERE clauses can be used to list triggers based on a particular naming pattern. The LIKE clause includes the name of the table with which the trigger is associated or a pattern for the table name that includes wildcards (%). With the WHERE clause, you can use the names of fields in the results to create an expression that sets a condition determining the results returned. Here is an example using this statement:

```
SHOW TRIGGERS LIKE 'students' \G

*************************** 1. row ***************************
  Trigger: students_deletion
    Event: DELETE
    Table: students
Statement: BEGIN
INSERT INTO students_deleted
(student_id, name_first, name_last)
VALUES(OLD.student_id, OLD.name_first, OLD.name_last);
END
   Timing: BEFORE
  Created: NULL
 sql_mode:
  Definer: root@localhost
```

See CREATE TRIGGER earlier in this chapter for more information on triggers and to see how the trigger shown was created.



# 10

# Aggregate Clauses, Aggregate Functions, and Subqueries

MySQL has many built-in functions that you can use in SQL statements for performing calculations on combinations of values in databases; these are called *aggregate functions*. They include such types of basic statistical analysis as counting rows, determining the average of a given column's value, finding the standard deviation, and so forth. The first section of this chapter describes MySQL aggregate functions and includes examples of most of them. The second section provides a tutorial about subqueries. It includes several examples of subqueries in addition to the ones shown in the first section and in various examples throughout this book. Subqueries are included in this chapter because they are often used with `GROUP BY` and aggregate functions and because they're another method for grouping selected data.

The following functions are covered in this chapter:

`AVG( )`, `BIT_AND( )`, `BIT_OR( )`, `COUNT( )`, `GROUP_CONCAT( )`, `MAX( )`, `MIN( )`, `STD( )`, `STDDEV( )`, `STDDEV_POP( )`, `STDDEV_SAMP( )`, `SUM( )`, `VAR_POP( )`, `VAR_SAMP( )`, `VARIANCE( )`.

## Aggregate Functions in Alphabetical Order

This section describes each aggregate function. Many of the examples use a subquery. For detailed information about subqueries, see the "Subqueries" section later in this chapter.

A few general aspects of aggregate functions include:

- Aggregate functions return NULL when they encounter an error.
- Most uses for aggregate functions include a `GROUP BY` clause, which is specified in each description. If an aggregate function is used without a `GROUP BY` clause it operates on all rows.

## AVG( )

AVG([DISTINCT] *column*)

This function returns the average or mean of a set of numbers given as the argument. It returns NULL if unsuccessful. The DISTINCT keyword causes the function to count only unique values in the calculation; duplicate values will not factor into the averaging.

When returning multiple rows, you generally want to use this function with the GROUP BY clause that groups the values for each unique item, so that you can get the average for that item. This will be clearer with an example:

```
SELECT sales_rep_id,
CONCAT(name_first, SPACE(1), name_last) AS rep_name,
AVG(sale_amount) AS avg_sales
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

This SQL statement returns the average amount of sales in the sales table made by each sales representative. It will total all values found for the sale_amount column, for each unique value for sales_rep_id, and divide by the number of rows found for each of those unique values. If you would like to include sales representatives who made no sales in the results, you'll need to change the JOIN to a RIGHT JOIN:

```
SELECT sales_rep_id,
CONCAT(name_first, SPACE(1), name_last) AS rep_name,
FORMAT(AVG(sale_amount), 2) AS avg_sales
FROM sales
RIGHT JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

Sales representatives who made no sales will show up with NULL in the avg_sales column. This version of the statement also includes an enhancement: it rounds the results for avg_sales to two decimal places by adding the FORMAT( ) function.

If we only want the average sales for the current month, we could add a WHERE clause. However, that would negate the effect of the RIGHT JOIN: sales people without orders for the month wouldn't appear in the list. To include them, first we need to run a subquery that extracts the sales data that meets the conditions of the WHERE clause, and then we need to join the subquery's results to another subquery containing a tidy list of the names of sales reps:

```
SELECT sales_rep_id, rep_name,
IFNULL(avg_sales, 'none') as avg_sales_month
FROM
  (SELECT sales_rep_id,
   FORMAT(AVG(sale_amount), 2) AS avg_sales
   FROM sales
   JOIN sales_reps USING(sales_rep_id)
   WHERE DATE_FORMAT(date_of_sale, '%Y%m') =
         DATE_FORMAT(CURDATE(), '%Y%m')
   GROUP BY sales_rep_id) AS active_reps
RIGHT JOIN
  (SELECT sales_rep_id,
   CONCAT(name_first, SPACE(1), name_last) AS rep_name
   FROM sales_reps) AS all_reps
```

```
USING(sales_rep_id)
GROUP BY sales_rep_id;
```

In the first subquery here, we are determining the average sales for each sales rep that had sales for the current month. In the second subquery, we're putting together a list of names of all sales reps, regardless of sales. In the main query, using the `sales_rep_id` column as the joining point of the two results sets derived from the subqueries, we are creating a results set that will show the average sales for the month for each rep that had some sales, or (using `IFNULL( )`) the word *none* for those who had none.

## BIT_AND( )

`BIT_AND(`*expression*`)`

This function returns the bitwise `AND` for all bits for the expression given. Use this in conjunction with the `GROUP BY` clause. The function has a 64-bit precision. If there are no matching rows, before version 4.0.17 of MySQL, –1 is returned. Newer versions return 18446744073709551615, which is the value of 1 for all bits of an unsigned `BIGINT` column.

## BIT_OR( )

`BIT_OR(`*expression*`)`

This function returns the bitwise `OR` for all bits for the expression given. It calculates with a 64-bit precision (`BIGINT`). It returns 0 if no matching rows are found. Use it in conjunction with the `GROUP BY` clause.

## BIT_XOR( )

`BIT_XOR(`*expression*`)`

This function returns the bitwise `XOR` (exclusive `OR`) for all bits for the expression given. It calculates with a 64-bit precision (`BIGINT`). It returns 0 if no matching rows are found. Use it in conjunction with the `GROUP BY` clause. This function is available as of version 4.1.1 of MySQL.

## COUNT( )

`COUNT([DISTINCT]` *expression*`)`

This function returns the number of rows retrieved in the `SELECT` statement for the given column. By default, rows in which the column is NULL are not counted. If the wildcard * is used as the argument, the function counts all rows, including those with NULL values. If you want only a count of the number of rows in the table, you don't need `GROUP BY`, and you can still include a `WHERE` to count only rows meeting specific criteria. If you want a count of the number of rows for each value of a column, you will need to use the `GROUP BY` clause. As an alternative to using `GROUP BY`, you can add the `DISTINCT` keyword to get a count of unique non-NULL values found for the given column. When you use `DISTINCT`, you cannot include any other columns in the `SELECT` statement. You can, however, include multiple columns or expressions within the function. Here is an example:

```
SELECT branch_name,
COUNT(sales_rep_id) AS number_of_reps
FROM sales_reps
JOIN branch_offices USING(branch_id)
GROUP BY branch_id;
```

This example joins the sales_reps and branch_offices tables together using the branch_id contained in both tables. We then use the COUNT( ) function to count the number of sales reps found for each branch (determined by the GROUP BY clause).

## GROUP_CONCAT( )

```
GROUP_CONCAT([DISTINCT] expression[, ...]
   [ORDER BY {unsigned_integer|column|expression}
   [ASC|DESC] [,column...]]
   [SEPARATOR character])
```

This function returns non-NULL values of a group concatenated by a GROUP BY clause, separated by commas. The parameters for this function are included in the parentheses, separated by spaces, not commas. The function returns NULL if the group doesn't contain non-NULL values.

Duplicates are omitted with the DISTINCT keyword. The ORDER BY clause instructs the function to sort values before concatenating them. Ordering may be based on an unsigned integer value, a column, or an expression. The sort order can be set to ascending with the ASC keyword (default), or to descending with DESC. To use a different separator from a comma, use the SEPARATOR keyword followed by the preferred separator.

The value of the system variable group_concat_max_len limits the number of elements returned. Its default is 1024. Use the SET statement to change the value. This function is available as of version 4.1 of MySQL.

As an example of this function, suppose that we wanted to know how many customers order a particular item. We could enter an SQL statement like this:

```
SELECT item_nbr AS Item,
GROUP_CONCAT(quantity) AS Quantities
FROM orders
WHERE item_nbr = 100
GROUP BY item_nbr;

+------+------------+
| Item | Quantities |
+------+------------+
|  100 | 7,12,4,8,4 |
+------+------------+
```

Notice that the quantities aren't sorted—it's the item numbers that are sorted by the GROUP BY clause. To sort the quantities within each field and to use a different separator, we would enter something like the following instead:

```
SELECT item_nbr AS Item,
GROUP_CONCAT(DISTINCT quantity
  ORDER BY quantity ASC
  SEPARATOR '|')
AS Quantities
```

```
FROM orders
WHERE item_nbr = 100
GROUP BY item_nbr;

+------+------------+
| Item | Quantities |
+------+------------+
|  100 | 4|7|8|12   |
+------+------------+
```

Because the results previously contained a duplicate value (4), we're eliminating duplicates here by including the DISTINCT keyword.

## MAX( )

MAX(*expression*)

This function returns the highest number in the values for a given column. It's normally used in conjunction with a GROUP BY clause specifying a unique column, so that values are compared for each unique item separately.

As an example of this function, suppose that we wanted to know the maximum sale for each sales person for the month. We could enter the following SQL statement:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
MAX(sale_amount) AS biggest_sale
FROM sales
JOIN sales_reps USING(sales_rep_id)
WHERE DATE_FORMAT(date_of_sale, '%Y%m') =
      DATE_FORMAT(CURDATE(), '%Y%m')
GROUP BY sales_rep_id DESC;
```

We've given sale_amount as the column for which we want the largest value returned for each sales rep. The WHERE clause indicates that we want only sales for the current month. Notice that the GROUP BY clause includes the DESC keyword. This will order the rows in descending order for the values of the biggest_sale field: the biggest sale at the top, the smallest at the bottom.

Here's an example of another handy but less obvious use of this function: suppose we have a table in which client profiles are kept by the sales people. When a sales rep changes a client profile through a web interface, instead of updating the existing row, the program we wrote creates a new entry. We use this method to prevent sales people from inadvertently overwriting data and to keep previous client profiles in case someone wants to refer to them later. When the client profile is viewed through the web interface, we want only the latest profile to appear. Retrieving the latest row becomes a bit cumbersome, but we can do this with MAX( ) and a subquery as follows:

```
SELECT client_name, profile,
MAX(entry_date) AS last_entry
FROM
  (SELECT client_id, entry_date, profile
   FROM client_profiles
   ORDER BY client_id, entry_date DESC) AS profiles
JOIN clients USING(client_id)
GROUP BY client_id;
```

In the subquery, we retrieve a list of profiles with the date each has in its entry in the table `client_profiles`; the results contain the duplicate entries for clients. In the main query, using `MAX( )`, we get the maximum (latest) date for each client. The associated `profile` is included in the columns selected by the main query. We join the results of the subquery to the `clients` table to extract the client's name.

The subquery is necessary so that we get the latest date instead of the oldest. The problem is that the `GROUP BY` clause orders the fields based on the given column. Without the subquery, the `GROUP BY` clause would use the value for the `entry_date` of the first row it finds, which will be the earliest date, not the latest. So we order the data in the subquery with the latest entry for each client first. `GROUP BY` then takes the first entry of the subquery results, which will be the latest entry.

## MIN( )

`MIN(`*expression*`)`

This function returns the lowest number in the values for a given column. It's normally used in conjunction with a `GROUP BY` clause specifying a unique column, so that values are compared for each unique item separately. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
MIN(sale_amount) AS smallest_sale,
MAX(sale_amount) AS biggest_sale
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

In this example, we retrieve the smallest sale and largest sale made by each sales representative. We use `JOIN` to join the two tables to get the sales rep's name. Because `MAX( )` is very similar, see the examples in its description earlier in this chapter for *additional* ways to use `MIN( )`.

## STD( )

`STD(`*expression*`)`

This function returns the population standard deviation of the given column. This function is an alias for `STDDEV( )`; see the description of that function for an example of its use.

## STDDEV( )

`STDDEV(`*expression*`)`

This function returns the population standard deviation of the given column. It's normally used in conjunction with a `GROUP BY` clause specifying a unique column, so that values are compared for each unique item separately. It returns NULL if no matching rows are found. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
SUM(sale_amount) AS total_sales,
COUNT(sale_amount) AS total_tickets,
AVG(sale_amount) AS avg_sale_per_ticket,
STDDEV(sale_amount) AS standard_deviation
```

```
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

This statement employs several aggregate functions. We use SUM( ) to get the total sales for each sales rep, COUNT( ) to retrieve the number of orders for the each, AVG( ) to determine the average sale, and STDDEV( ) to find out how much each sale made by each sales rep tends to vary from each one's average sale. Incidentally, statistical functions return several decimal places. To return only two decimal places, you can wrap each function in FORMAT( ).

## STDDEV_POP( )

STDDEV_POP(*expression*)

This function returns the population standard deviation of the given column. It was added in version 5.0.3 of MySQL for compliance with SQL standards. This function is an alias for STDDEV( ); see the description of that function earlier in this chapter for an example of its use.

## STDDEV_SAMP( )

STDDEV_SAMP(*expression*)

This function returns the sample standard deviation of the given column. It's normally used in conjunction with a GROUP BY clause specifying a unique column, so that values are compared for each unique item separately. It returns NULL if no matching rows are found. It was added in version 5.0.3 of MySQL for compliance with SQL standards. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
AVG(sale_amount) AS avg_sale_per_ticket,
STDDEV_POP(sale_amount) AS population_std_dev,
STDDEV_SAMP(sale_amount) AS sample_std_dev
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

This SQL statement uses several aggregate functions: AVG( ) to determine the average sale for each sales rep; STDDEV_POP( ) to determine how much each sale made by each sales rep tends to vary from each rep's average sale; and STDDEV_SAMP( ) to determine the standard deviation from the average based on a sample of the data.

## SUM( )

SUM([DISTINCT] *expression*)

This function returns the sum of the values for the given column or expression. It's normally used in conjunction with a GROUP BY clause specifying a unique column, so that values are compared for each unique item separately. It returns NULL if no matching rows are found. The parameter DISTINCT may be given within the parentheses of the function to add only unique values found for a given column. This parameter was added in version 5.1 of MySQL. Here is an example:

```
SELECT sales_rep_id,
SUM(sale_amount) AS total_sales
FROM sales
WHERE DATE_FORMAT(date_of_sale, '%Y%m') =
      DATE_FORMAT(SUBDATE(CURDATE(), INTERVAL 1 MONTH), '%Y%m')
GROUP BY sales_rep_id;
```

This statement queries the sales table to retrieve only sales made during the last month. From these results, SUM( ) returns the total sale amounts aggregated by the sales_rep_id (see "Grouping SELECT results" under the SELECT statement in Chapter 6).

## VAR_POP( )

VAR_POP(*expression*)

This function returns the variance of a given column, based on the rows selected as a population. It's synonymous with VARIANCE and was added in version 5.0.3 of MySQL for compliance with SQL standards. See the description of VAR_SAMP( ) for an example of this function's use.

## VAR_SAMP( )

VAR_SAMP(*expression*)

This function returns the variance of a given column, based on the rows selected as a sample of a given population. It's normally used in conjunction with a GROUP BY clause specifying a unique column, so that values are compared for each unique item separately. To determine the variance based on the entire population rather than a sample, use VAR_POP( ). Both of these functions were added in version 5.0.3 of MySQL for compliance with SQL standards. Here is an example of both:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
AVG(sale_amount) AS avg_sale,
STDDEV_POP(sale_amount) AS population_std_dev,
STDDEV_SAMP(sale_amount) AS sample_std_dev,
VAR_POP(sale_amount) AS population_variance,
VAR_SAMP(sale_amount) AS sample_variance
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

This SQL statement uses several aggregate functions: AVG( ) to determine the average sale for each sales rep; STDDEV_POP( ) to determine how much each sale made by each sales rep tends to vary from each rep's average sale; and STDDEV_SAMP( ) to determine the standard deviation from the average based on a sample of the data. It also includes VAR_POP( ) to show the variances based on the population, and VAR_SAMP( ) to return the variance based on the sample data.

## VARIANCE( )

VARIANCE(*expression*)

The *variance* is determined by taking the difference between each given value and the average of all values given. Each of those differences is then squared, and the results are

totaled. The average of that total is then determined to get the variance. This function returns the variance of a given column, based on the rows selected as a population. It's normally used in conjunction with a GROUP BY clause specifying a unique column, so that values are compared for each unique item separately. This function is available as of version 4.1 of MySQL. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS rep_name,
AVG(sale_amount) AS avg_sale,
STDDEV_POP(sale_amount) AS standard_deviation,
VARIANCE(sale_amount) AS variance
FROM sales
JOIN sales_reps USING(sales_rep_id)
GROUP BY sales_rep_id;
```

This SQL statement uses a few aggregate functions: AVG( ) to determine the average sale for each sales rep; STDDEV_POP( ) to determine how much each sale made by each sales rep tends to vary from each rep's average sale; and VARIANCE( ) to show the variances based on the population. To comply with SQL standards, VAR_POP( ) could have been used instead of VARIANCE( ).

# Subqueries

A subquery is a SELECT statement nested within another SQL statement. This feature became available as of version 4.1 of MySQL. Although the same results can be accomplished by using the JOIN clause or UNION, depending on the situation, subqueries are a cleaner approach that is sometimes easier to read. They make a complex query more modular, which makes it easier to create and to troubleshoot. Here is a simple example of a subquery:

```
SELECT *
FROM
  (SELECT col1, col2
   FROM table1
   WHERE col_id = 1000) AS derived1
ORDER BY col2;
```

In this example, the subquery or *inner* query is a SELECT statement specifying two column names. The other query is called the *main* or *outer* query. It doesn't have to be a SELECT. It can be an INSERT, a DELETE, a DO, an UPDATE, or even a SET statement. The outer query generally can't select data or modify data from the same table as an inner query, but this doesn't apply if the subquery is part of a FROM clause. A subquery can return a value (a scalar), a field, multiple fields containing values, or a full results set that serves as a derived table.

You can encounter performance problems with subqueries if they are not well constructed. One problem occurs when a subquery is placed within an IN( ) clause as part of a WHERE clause. It's generally better to use the = operator for each value, along with AND for each parameter/value pair.

When you see a performance problem with a subquery, try reconstructing the SQL statement with JOIN and compare the differences using the BENCHMARK( ) function. If the performance is better without a subquery, don't give up on subqueries. Only

in some situations is performance poorer. For those situations where there is a performance drain, MySQL AB is working on improving MySQL subqueries. So performance problems you experience now may be resolved in future versions. You may just need to upgrade to the current release or watch for improvements in future releases.

## Single Field Subqueries

The most basic subquery is one that returns a scalar or single value. This type of subquery is particularly useful in a WHERE clause in conjunction with an = operator, or in other instances where a single value from an expression is permitted.

As an example of this situation, suppose that at our fictitious college one of the music teachers, Sonia Oram, has called us saying that she wants a list of students for one of her classes so that she can call them to invite them to a concert. She wants the names and telephone numbers for only the students in her first period Monday morning class.

The way most databases store this data, the course number would be a unique key and would make it easy to retrieve the other data without a subquery. But Sonia doesn't know the course number, so we enter an SQL statement like this:

```
SELECT CONCAT(name_first, ' ', name_last) AS student,
phone_home, phone_dorm
FROM students
JOIN course_rosters USING (student_id)
WHERE course_id =
  (SELECT course_id
   FROM course_schedule
   JOIN teachers USING (teacher_id)
   WHERE semester_code = '2007AU'
   AND class_time = 'monday_01'
   AND name_first = 'Sonia'
   AND name_last = 'Oram');
```

Notice in the subquery that we're joining the course_schedule table with teachers so we can give the teacher's first and last name in the WHERE clause of the subquery. We're also indicating in the WHERE clause a specific semester (Autumn 2007) and time slot (Monday, first period). The results of these specifics should be one course identification number because a teacher won't teach more than one class during a particular class period. That single course number will be used by the WHERE clause of the main query to return the list of students on the class roster for the course, along with their telephone numbers.

If by chance more than one value is returned by the subquery in the previous example, MySQL will return an error:

```
ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
SQLSTATE = 21000
Message = "Subquery returns more than 1 row"
```

Despite our supposition, it is possible that a teacher might teach more than one class at a time: perhaps the teacher is teaching one course in violin and another in viola,

but each class had so few students that the department head put them together. In such a situation, the teacher would want the data for both course numbers. To use multiple fields derived from a subquery in a WHERE clause like this, we would have to use something other than the = operator, such as IN. For this kind of situation, see the next section on "Multiple Fields Subqueries."

## Multiple Fields Subqueries

In the previous section, we discussed instances where one scalar value was obtained from a subquery in a WHERE clause. However, there are times when you may want to match multiple values. For those situations you will need to use the subquery in conjunction with an operator or a clause: ALL, ANY, EXISTS, IN, or SOME.

As an example of a multiple fields subquery—and specifically of a subquery using IN (or using ANY or SOME)—let's adapt the example from the previous section to a situation where the teacher wants the contact information for students in all of her classes. To do this, we can enter the following SQL statement:

```
SELECT CONCAT(name_first, ' ', name_last) AS student,
phone_home, phone_dorm
FROM students
JOIN course_rosters USING (student_id)
WHERE course_id IN
  (SELECT course_id
   FROM course_schedule
   JOIN teachers USING (teacher_id)
   WHERE semester_code = '2007AU'
   AND name_first = 'Sonia'
   AND name_last = 'Oram');
```

In this example, notice that the subquery is contained within the parentheses of the IN clause. Subqueries are executed first, so the results will be available before the WHERE clause is executed. Although a comma-separated list isn't returned, MySQL still accepts the results so that they may be used by the outer query. The criteria of the WHERE clause here does not specify a specific time slot as the earlier example did, so multiple values are much more likely to be returned.

Instead of IN, you can use ANY or SOME to obtain the same results by the same methods. (ANY and SOME are synonymous.) These two keywords must be preceded by a comparison operator (e.g., =, <, >). For example, we could replace the IN in the SQL previous statement with = ANY or with = SOME and the same results will be returned. IN can be preceded with NOT for negative comparisons: NOT IN(...). This is the same as != ANY (...) and != SOME (...).

Let's look at another subquery returning multiple values but using the ALL operator. The ALL operator must be preceded by a comparison operator (e.g., =, <, >). As an example of this usage, suppose one of the piano teachers provides weekend seminars for students. Suppose also that he heard a few students are enrolled in all of the seminars he has scheduled for the semester and he wants a list of their names and telephone numbers in advance. We should be able to get that data by entering an

SQL statement like the following (though currently it doesn't work, for reasons to be explained shortly):

```
SELECT DISTINCT student_id,
CONCAT(name_first, ' ', name_last) AS student
FROM students
JOIN seminar_rosters USING (student_id)
WHERE seminar_id = ALL
  (SELECT seminar_id
   FROM seminar_schedule
   JOIN teachers ON (instructor_id = teacher_id)
   WHERE semester_code = '2007AU'
   AND name_first = 'Sam'
   AND name_last = 'Oram');
```

In this example, a couple of the tables have different column names for the ID we want, and we have to join one of them with `ON` instead of `USING`, but that has nothing to do with the subquery. What's significant is that this subquery returns a list of seminar identification numbers and is used in the `WHERE` clause of the main query with `= ALL`. Unfortunately, although this statement is constructed correctly, it doesn't work with MySQL at the time of this writing and just returns an empty set. However, it should work in future releases of MySQL, so I've included it for future reference. For now, we would have to reorganize the SQL statement like so:

```
SELECT student_id, student
FROM
  (SELECT student_id, COUNT(*)
     AS nbr_seminars_registered,
   CONCAT(name_first, ' ', name_last)
     AS student
   FROM students
   JOIN seminar_rosters USING (student_id)
   WHERE seminar_id IN
     (SELECT seminar_id
      FROM seminar_schedule
      JOIN teachers
      ON (instructor_id = teacher_id)
      WHERE semester_code = '2007AU'
      AND name_first = 'Sam'
      AND name_last = 'Oram')
   GROUP BY student_id) AS students_registered
WHERE nbr_seminars_registered =
   (SELECT COUNT(*) AS nbr_seminars
   FROM seminar_schedule
   JOIN teachers
   ON (instructor_id = teacher_id)
   WHERE semester_code = '2007AU'
   AND name_first = 'Sam'
   AND name_last = 'Oram');
```

This is much more involved, but it does work with the latest release of MySQL.

The first subquery is used to get the student's name. This subquery's `WHERE` clause uses another subquery to retrieve the list of seminars taught by the professor for the semester, to determine the results set from which the main query will draw its

ultimate data. The third subquery counts the number of seminars that the same professor is teaching for the semester. This single value is used with the WHERE clause of the main query. In essence, we're determining the number of seminars the professor is teaching and which students are registered for all of them.

The last possible method for using multiple fields in a subquery uses EXISTS. With EXISTS, in order for it to return meaningful or desired results, you need to stipulate in the WHERE clauses of the subquery a point in which it is joined to the outer query. Using the example from the previous section involving the teacher Sonia Oram, let's suppose that we want to retrieve a list of courses that she teaches:

```
SELECT DISTINCT course_id, course_name
FROM courses
WHERE EXISTS
  (SELECT course_id
   FROM course_schedule
   JOIN teachers USING (teacher_id)
   WHERE semester_code = '2007AU'
   AND name_first = 'Sonia'
   AND name_last = 'Oram'
   AND courses.course_id = course_schedule.course_id);
```

As you can see here, we've added EXISTS to the WHERE clause with the subquery in parentheses, similar to using IN. The significant difference is that we added courses.course_id = course_schedule.course_id to the end. Without it, a list of all courses would be returned regardless of the criteria of the WHERE clause in the subquery. Incidentally, if we specified NOT EXISTS instead, we would get all courses *except* for the ones taught by the teacher given.

## Results Set Subqueries

A subquery can be used to generate a results set, which is a table from which an outer query can select data. That is, a subquery can be used in a FROM clause as if it were another table in a database. It is a *derived table*. Along these lines, each derived table must be named. This is done with AS following the parentheses containing the subquery. A subquery contained in a FROM clause generally cannot be a correlated subquery—that is, it cannot reference the same table as the outer query. The exception is if it's constructed with a JOIN.

In the following example, let's consider the subquery separately as though it were a plain query and not a subquery. It will generate a results set containing the student's ID and the student's average exam score for a specific course taught during a specific semester. The query uses AVG( ), which requires a GROUP BY clause. The problem with GROUP BY is that it will order data only by the columns by which it's given to group data. In this case, it will order the data by student_id and not list the results by any other, more useful column. If we want to order the data so that the highest student average is first, descending in order to the lowest student average, we have to turn our query into a subquery and have the outer query re-sort the results:

```
SELECT CONCAT(name_first, ' ', name_last) AS student,
student_id, avg_grade
```

```
FROM students
JOIN
  (SELECT student_id,
   AVG(exam_grade) AS avg_grade
   FROM exams
   WHERE semester_code = '2007AU'
   AND course_id = 1489
   GROUP BY student_id) AS grade_averages
USING(student_id)
ORDER BY avg_grade DESC;
```

- The results set (the derived table generated by the subquery in the `FROM` clause) is named `grade_averages`. Notice that although the column `student_id` exists in the derived table, in the table from which it gets its data (i.e., `exams`) and in the primary table used in the main query (i.e., `students`), there is no ambiguity. No error is generated. However, if we wanted to specify that the data be taken from the derived table, we could put `grade_averages.student_id` in the `SELECT` of the outer query.
- This subquery is a correlated subquery, which is generally not permitted in a `FROM` clause. It's allowed in this example because we are using a `JOIN` to join the results set to the table referenced in the outer query.