Learning Basic SQL Commands

Lesson7b

Having just learned the basics of the database design process, this lesson provides a primer on the core SQL syntax, which you will use to create and manipulate your MySQL database tables. This is a very hands-on lesson, and it assumes that you are able to issue queries through the MySQL monitor on Windows or Linux/Unix. Alternatively, if you use a GUI to MySQL, this lesson assumes you know the methods for issuing queries through those interfaces. Please note, this may not be the most exciting lesson in the book, but it will show you many, many functional examples of elements you'll use throughout the rest of your work.

In this lesson, you will learn

- The basic MySQL data types
- How to use the `CREATE TABLE` command to create a table
- How to use the `INSERT` command to enter records
- How to use the `SELECT` command to retrieve records
- How to use basic functions, the `WHERE` clause, and the `GROUP BY` clause in `SELECT` expressions
- How to select from multiple tables, using `JOIN`
- How to use the `UPDATE` and `REPLACE` commands to modify existing records
- How to use the `DELETE` command to remove records
- How to use string functions built into MySQL
- How to use date and time functions built into MySQL

Learning the MySQL Data Types

Properly defining the fields in a table is important to the overall optimization of your database. You should use only the type and size of field you really need to use; don't define a field as 10 characters wide if you know you're only going to use 2 characters that's 8 extra characters the database has to account for, even if they're unused. These field types are also referred to as data types, as in the "type of data" you will be storing in those fields.

MySQL uses many different data types, broken into three categories: numeric, date and time, and string types. Pay close attention because defining the data type is more important than any other part of the table creation process.

Numeric Data Types

MySQL uses all the standard ANSI SQL numeric data types, so if you're coming to MySQL from a different database system, these definitions will look familiar to you. The following list shows the common numeric data types and their descriptions.

NOTE:

The terms signed and unsigned will be used in the list of numeric data types. If you remember your basic algebra, you'll recall that a signed integer can be a positive or negative integer, whereas an unsigned integer is always a non-negative integer.

- `INT` A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from 2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.
- `TINYINT` A very small integer that can be signed or unsigned. If signed, the allowable range is from 128 to 127. If unsigned, the allowable range is from 0 to 255. You can specify a width of up to 4 digits.
- `SMALLINT` A small integer that can be signed or unsigned. If signed, the allowable range is from 32768 to 32767. If unsigned, the allowable range is from 0 to 65535. You can specify a width of up to 5 digits.
- `MEDIUMINT` A medium-sized integer that can be signed or unsigned. If signed, the allowable range is from 8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. You can specify a width of up to 9 digits.
- `BIGINT` A large integer that can be signed or unsigned. If signed, the allowable range is from 9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. You can specify a width of up to 11 digits.
- `FLOAT(M,D)` A floating-point number that cannot be unsigned. You can define the display length `(M)` and the number of decimals `(D)`. This is not required and will default to `10,2`, where `2` is the number of decimals and `10` is the total number of digits (including decimals). Decimal precision can go to 24 places for a `FLOAT`.
- `DOUBLE(M,D)` A double precision floating-point number that cannot be unsigned. You can define the display length `(M)` and the number of decimals `(D)`. This is not required and will default to `16,4`, where `4` is the number of decimals. Decimal precision can go to 53 places for a `DOUBLE`. `REAL` is a synonym for `DOUBLE`.
- `DECIMAL(M,D)` An unpacked floating-point number that cannot be unsigned. In unpacked decimals, each decimal corresponds to one byte. Defining the display `length (M)` and the number of decimals `(D)` is required. `NUMERIC` is a synonym for `DECIMAL`.

Of all the MySQL numeric data types, you will likely use `INT` most often. You can run into problems if you define your fields to be smaller than you actually need; for example, if you define an `ID` field as an unsigned `TINYINT`, you won't be able to successfully insert that 256th record if `ID` is a primary `key` (and thus required).

Date and Time Types

MySQL has several data types available for storing dates and times, and these data types are flexible in their input. In other words, you can enter dates that are not really days, such as February 30February has only 28 or 29 days, never 30. Also, you can store dates with missing information. For example, if you know that someone was born sometime in November of 1980, you can use 1980-11-00, where "00" would have been for the day, if you knew it.

The flexibility of MySQL's date and time types also means that the responsibility for date checking falls on the application developer (that would be you). MySQL checks only two elements for validity: that the month is between 0 and 12 and the day is between 0 and 31. MySQL does not automatically verify that the 30th day of the second month (February 30th) is a valid date.

The MySQL date and time datatypes are

- `DATE` A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.
- `DATETIME` A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
- `TIMESTAMP` A timestamp between midnight, January 1, 1970 and sometime in 2037. You can define multiple lengths to the `TIMESTAMP` field, which directly correlates to what is stored in it. The default length for `TIMESTAMP` is 14, which stores YYYYMMDDHHMMSS. This looks like the previous `DATETIME` format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000. Other definitions of `TIMESTAMP` are 12 (YYMMDDHHMMSS), 8 (YYYYMMDD), and 6 (YYMMDD).
- `TIME` Stores the time in HH:MM:SS format.
- `YEAR(M)` Stores a year in 2-digit or 4-digit format. If the length is specified as 2 (for example, `YEAR(2)`), `YEAR` can be 1970 to 2069 (70 to 69). If the length is specified as 4, `YEAR` can be 1901 to 2155. The default length is 4.

You will likely use `DATETIME` or `DATE` more often than any other date- or time-related data type.

String Types

Although numeric and date types are fun, most data you'll store will be in string format. This list describes the common string datatypes in MySQL.

- `CHAR(M)` A fixed-length string between 1 and 255 characters in length (for example, `CHAR(5)`), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
- `VARCHAR(M)` A variable-length string between 1 and 255 characters in length; for example, `VARCHAR(25)`. You must define a length when creating a `VARCHAR` field.
- `BLOB` or `TEXT` A field with a maximum length of 65535 characters. `BLOB`s are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as `TEXT` also hold large amounts of data; the difference between the two is that sorts and comparisons on stored data are case sensitive on `BLOB`s and are not case sensitive in `TEXT` fields. You do not specify a length with `BLOB` or `TEXT`.
- `TINYBLOB` or `TINYTEXT` A `BLOB` or `TEXT` column with a maximum length of 255 characters. You do not specify a length with `TINYBLOB` or `TINYTEXT`.
- `MEDIUMBLOB` or `MEDIUMTEXT` A `BLOB` or `TEXT` column with a maximum length of 16777215 characters. You do not specify a length with `MEDIUMBLOB` or `MEDIUMTEXT`.
- `LONGBLOB` or `LONGTEXT` A `BLOB` or `TEXT` column with a maximum length of 4294967295 characters. You do not specify a length with `LONGBLOB` or `LONGTEXT`.
- `ENUM` An enumeration, which is a fancy term for list. When defining an `ENUM`, you are creating a list of items from which the value must be selected (or it can be `NULL`). For example, if you wanted your field to contain "A" or "B" or "C", you would define your `ENUM` as `ENUM ('A', 'B', 'C')` and only those values (or `NULL`) could ever populate that field. `ENUM`s can have 65535 different values. `ENUM`s use an index for storing items.

NOTE:

The `SET` type is similar to `ENUM` in that it is defined as a list. However, the `SET` type is stored as a full value rather than an index of a value, as with `ENUM`s.

You will probably use `VARCHAR` and `TEXT` fields more often than other field types, and `ENUM`s are useful as well.

Learning the Table Creation Syntax

The table creation command requires

- Name of the table
- Names of fields
- Definitions for each field

The generic table creation syntax is

```
CREATE TABLE table_name (column_name column_type);
```

The table name is up to you of course, but should be a name that reflects the usage of the table. For example, if you have a table that holds the inventory of a grocery store, you wouldn't name the table `s`. You would probably name it something like `grocery_inventory`. Similarly, the field names you select should be as concise as possible and relevant to the function they serve and data they hold. For example, you might call a field holding the name of an item `item_name`, not `n`.

This example creates a generic `grocery_inventory` table with fields for ID, name, description, price, and quantity:

```
mysql> CREATE TABLE grocery_inventory (
    -> id int not null primary key auto_increment,
    -> item_name varchar (50) not null,
    -> item_desc text,
    -> item_price float not null,
    -> curr_qty int not null
    -> );
Query OK, 0 rows affected (0.02 sec)
```

NOTE:

The `id` field is defined as a primary key. You will learn more about keys in later lessons, in the context of creating specific tables as parts of sample applications. By using `auto_increment` as an attribute of the field, you are telling MySQL to go ahead and add the next available number to the `id` field for you.

The MySQL server will respond with `Query OK` each time a command, regardless of type, is successful. Otherwise, an error message will be displayed.

Using the `INSERT` Command

After you have created some tables, you'll use the SQL command `INSERT` for adding new records to these tables. The basic syntax of `INSERT` is

```
INSERT INTO table_name (column list) VALUES (column values);
```

Within the parenthetical list of values, you must enclose strings within quotation marks. The SQL standard is single quotes, but MySQL enables the usage of either single or double quotes. Remember to escape the type of quotation mark used, if it's within the string itself.

NOTE:

Integers do not require quotation marks around them.

Here is an example of a string where escaping is necessary:

```
O'Connor said "Boo"
```

If you enclose your strings in double quotes, the `INSERT` statement would look like this:

```
INSERT INTO table_name (column_name) VALUES ("O'Connor said
\"Boo\"");
```

If you enclose your strings in single quotes instead, the `INSERT` statement would look like this:

```
INSERT INTO table_name (column_name) VALUES ('O\'Connor said "Boo"');
```

A Closer Look at `INSERT`

Besides the table name, there are two main parts of the `INSERT` statement: the column list and the value list. Only the value list is actually required, but if you omit the column list, you must specifically provide for each column in your value listin the exact order.

Using the `grocery_inventory` table as an example, you have five fields: `id`, `item_name`, `item_desc`, `item_price`, and `curr_qty`. To insert a complete record, you could use either of these statements:

- A statement with all columns named:
- `insert into grocery_inventory (id, item_name, item_desc, item_price, curr_qty)`
- `values ('1', 'Apples', 'Beautiful, ripe apples.', '0.25', 1000);`
- A statement that uses all columns but does not explicitly name them:
- `insert into grocery_inventory values ('2', 'Bunches of Grapes',`
- `'Seedless grapes.', '2.99', 500);`

Give both of them a try and see what happens. You should get results like this:

```
mysql> insert into grocery_inventory
    -> (id, item_name, item_desc, item_price, curr_qty)
    -> values
```

```
    -> (1, 'Apples', 'Beautiful, ripe apples.', 0.25, 1000);
Query OK, 1 row affected (0.01 sec)

mysql> insert into grocery_inventory values (2, 'Bunches of Grapes',
    -> 'Seedless grapes.', 2.99, 500);
Query OK, 1 row affected (0.01 sec)
```

Now for some more interesting methods of using `INSERT`. Because `id` is an auto-incrementing integer, you don't have to put it in your value list. However, if there's a value you specifically don't want to list (such as `id`), you then must list the remaining columns in use. For example, the following statement does not list the columns and also does not give a value for `id`, and it will produce an error:

```
mysql> insert into grocery_inventory values
    -> ('Bottled Water (6-pack)', '500ml spring water.', 2.29, 250);
ERROR 1136: Column count doesn't match value count at row 1
```

Because you didn't list any columns, MySQL expects all of them to be in the value list, causing an error on the previous statement. If the goal was to let MySQL do the work for you by auto-incrementing the `id` field, you could use either of these statements:

- A statement with all columns named except `id`:
- `  insert into grocery_inventory (item_name, item_desc, item_price, curr_qty)`
- `  values ('Bottled Water (6-pack)', '500ml spring water.', '2.29', 250);`

- A statement that uses all columns, but does not explicitly name them and indicates a NULL entry for `id` (so one is filled in for you):
- `  insert into grocery_inventory values ('NULL', 'Bottled Water (6-pack)',`
- `  '500ml spring water.', 2.29, 250);`

Go ahead and pick one to use so that your `grocery_inventory` table has three records in total. It makes no different to MySQL, but as with everything based on user preferences, be consistent in your application development. Consistent structures will be easier for you to debug later, because you'll know what to expect.

Using the `SELECT` Command

`SELECT` is the SQL command used to retrieve records from your tables. This command syntax can be totally simplistic or very complicated, depending on which fields you want to select, if you want to select from multiple tables, and what conditions you plan to impose. As you become more comfortable with database programming, you will learn to enhance your `SELECT` statements, ultimately making

your database do as much work as possible and not overworking your programming language.

The most basic `SELECT` syntax looks like this:

```
SELECT expressions_and_columns FROM table_name
[WHERE some_condition_is_true]
[ORDER BY some_column [ASC | DESC]]
[LIMIT offset, rows]
```

Look at the first line:

```
SELECT expressions_and_columns FROM table_name
```

One handy expression is the `*` symbol, which stands for everything. So, to select everything (all rows, all columns) from the `grocery_inventory` table, your SQL statement would be

```
SELECT * FROM grocery_inventory;
```

Depending on how much data found in the `grocery_inventory` table, your results will vary, but the results might look something like this:

```
mysql> select * from grocery_inventory;
+---+---------------+------------------------+----------+---------+
| id| item_name     | item_desc              | item_price| curr_qty|
+---+---------------+------------------------+----------+---------+
|  1| Apples        | Beautiful, ripe apples.|      0.25|     1000|
|  2| Bunches of Grapes | Seedless grapes.   |      2.99|      500|
|  3| Bottled Water (6-pack)| 500ml spring water.|   2.29|      250|
+---+---------------+------------------------+----------+---------+
3 rows in set (0.00 sec)
```

As you can see, MySQL creates a lovely, formatted table with the names of the columns along the first row as part of the result set. If you only want to select specific columns, replace the `*` with the names of the columns, separated by commas. The following statement selects just the `id, item_name`, and `curr_qty` fields from the `grocery_inventory` table:

```
mysql> select id, item_name, curr_qty from grocery_inventory;
+----+-----------------------+----------+
| id | item_name             | curr_qty |
+----+-----------------------+----------+
|  1 | Apples                |     1000 |
|  2 | Bunches of Grapes     |      500 |
|  3 | Bottled Water (6-pack) |      250 |
+----+-----------------------+----------+
3 rows in set (0.00 sec)
```

Ordering `SELECT` Results

Results of `SELECT` queries are ordered as they were inserted into the table, and shouldn't be relied upon as a meaningful ordering system. If you want to order results a specific way, such as by date, ID, name, and so on, specify your requirements using the `ORDER BY` clause. In the following statement, results are ordered by `item_name`:

```
mysql> select id, item_name, curr_qty from grocery_inventory
    -> order by item_name;
+----+------------------------+----------+
| id | item_name              | curr_qty |
+----+------------------------+----------+
|  1 | Apples                 |     1000 |
|  3 | Bottled Water (6-pack) |      250 |
|  2 | Bunches of Grapes      |      500 |
+----+------------------------+----------+
3 rows in set (0.04 sec)
```

NOTE:

When selecting results from a table without specifying a sort order, the results may or may not be ordered by their key value. This occurs because MySQL reuses the space taken up by previously deleted rows. In other words, if you add records with ID values of 1 through 5, delete the record with ID number 4, and then add another record (ID number 6), the records might appear in the table in this order: 1, 2, 3, 6, 5.

The default sorting of `ORDER BY` results is ascending (`ASC`); strings sort from A to Z, integers start at 0, dates sort from oldest to newest. You can also specify a descending sort, using `DESC`:

```
mysql> select id, item_name, curr_qty from grocery_inventory
    -> order by item_name desc;
+----+------------------------+----------+
| id | item_name              | curr_qty |
+----+------------------------+----------+
|  2 | Bunches of Grapes      |      500 |
|  3 | Bottled Water (6-pack) |      250 |
|  1 | Apples                 |     1000 |
+----+------------------------+----------+
3 rows in set (0.00 sec)
```

You're not limited to sorting by just one field you can specify as many fields as you want, separated by a comma. The sorting priority is the order in which you list the fields.

Limiting Your Results

You can use the `LIMIT` clause to return only a certain number of records from your `SELECT` query result. There are two requirements when using the `LIMIT` clause: the

offset and the number of rows. The offset is the starting position, and the number of rows should be self-explanatory.

Suppose you had more than two or three records in the `grocery_inventory` table, and you wanted to select the ID, name, and quantity of the first three, ordered by `curr_qty`. In other words, you want to select the three items with the least inventory. The following single-parameter limit will start at the 0 position and go to the third record:

```
mysql> select id, item_name, curr_qty from grocery_inventory
    -> order by curr_qty limit 3;
+----+------------------------+----------+
| id | item_name              | curr_qty |
+----+------------------------+----------+
|  4 | Bananas                |      150 |
|  3 | Bottled Water (6-pack) |      250 |
|  2 | Bunches of Grapes      |      500 |
+----+------------------------+----------+
3 rows in set (0.00 sec)
```

The `LIMIT` clause can be quite useful in an actual application. For example, you can use the `LIMIT` clause within a series of `SELECT` statements to travel through results in steps (first three items, next three items, next three items after that):

1. `SELECT * FROM grocery_inventory ORDER BY curr_qty LIMIT 0, 3;`
2. `SELECT * FROM grocery_inventory ORDER BY curr_qty LIMIT 3, 3;`
3. `SELECT * FROM grocery_inventory ORDER BY curr_qty LIMIT 6, 3;`

If you specify an offset and number of rows in your query, and no results are found, you won't see an error just an empty result set. For example, if the `grocery_inventory` table contains only six records, a query with a `LIMIT` offset of `6` will produce no results:

```
mysql> select id, item_name, curr_qty from grocery_inventory
    -> order by curr_qty limit 6, 3;
Empty set (0.00 sec)
```

In Web-based applications, when lists of data are displayed with links such as "previous 10" and "next 10," it's a safe bet that a `LIMIT` clause is at work.

Using WHERE in Your Queries

You have learned numerous ways to retrieve particular columns from your tables, but not specific rows. This is when the WHERE clause comes in to play. From the example SELECT syntax, you see that WHERE is used to specify a particular condition:

```
SELECT expressions_and_columns FROM table_name
[WHERE some_condition_is_true]
```

An example would be to retrieve all the records for items with a quantity of 500:

```
mysql> select * from grocery_inventory where curr_qty = 500;
+----+-------------+-----------------------+------------+----------+
| id | item_name   | item_desc             | item_price | curr_qty |
+----+-------------+-----------------------+------------+----------+
|  2 | Bunches of Grapes | Seedless grapes. |       2.99 |      500 |
|  5 | Pears             | Anjou, nice and sweet. |  0.5 |      500 |
+----+-------------+-----------------------+------------+----------+
2 rows in set (0.00 sec)
```

As shown previously, if you use an integer as part of your WHERE clause, quotation marks are not required. Quotation marks are required around strings, and the same rules apply with regard to escaping characters, as you learned in the section on INSERT.

Using Operators in WHERE Clauses

You've used the equal sign (=) in your WHERE clauses to determine the truth of a condition that is, whether one thing is equal to another. You can use many types of operators, with comparison operators and logical operators being the most popular types.

Table 1. Basic Comparison Operators and Their Meanings

Operator	Meaning
=	Equal to
!=	Not equal to
<=	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than

There's also a handy operator called BETWEEN, which is useful with integer or data comparisons because it searches for results between a minimum and maximum value.

For example

```
mysql> select * from grocery_inventory
    -> where item_price between 1.50 and 3.00;
+----+---------------+--------------------+------------+----------+
| id | item_name      | item_desc           | item_price | curr_qty |
+----+---------------+--------------------+------------+----------+
|  2 | Bunches of Grapes | Seedless grapes. |       2.99 |      500 |
|  3 | Bottled Water (6-pack) | 500ml spring water.|2.29 |      250 |
|  4 | Bananas            | Bunches, green.     |   1.99 |      150 |
+----+---------------+--------------------+------------+----------+
3 rows in set (0.00 sec)
```

Other operators include logical operators, which enable you to use multiple comparisons within your `WHERE` clause. The basic logical operators are `AND` and `OR`. When using `AND`, all comparisons in the clause must be true to retrieve results, whereas using `OR` allows a minimum of one comparison to be true. Also, you can use the `IN` operator to specify a list of items that you want to match.

String Comparison Using `LIKE`

You were introduced to matching strings within a `WHERE` clause by using = or `!=`, but there's another useful operator for the `WHERE` clause, when comparing strings: the `LIKE` operator. This operator uses two characters as wildcards in pattern matching.

- `%`Matches multiple characters
- _Matches exactly one character

For example, if you want to find records in the `grocery_inventory` table where the first name of the item starts with the letter `A`, you would use

```
mysql> select * from grocery_inventory where item_name like 'A%';
+----+-----------+-------------------------+------------+----------+
| id | item_name | item_desc               | item_price | curr_qty |
+----+-----------+-------------------------+------------+----------+
|  1 | Apples    | Beautiful, ripe apples. |       0.25 |     1000 |
|  6 | Avocado   | Large Haas variety.     |       0.99 |      750 |
+----+-----------+-------------------------+------------+----------+
```

NOTE:

Unless performing a `LIKE` comparison on a binary string, the comparison is not case sensitive. You can force a case-sensitive comparison using the `BINARY` keyword.

Selecting from Multiple Tables

You're not limited to selecting only one table at a time. That would certainly make application programming a long and tedious task! When you select from more than one table in one `SELECT` statement, you are really joining the tables together.

Suppose you have two tables: `fruit` and `color`. You can select all rows from each of the two tables, using two separate `SELECT` statements:

```
mysql> select * from fruit;
+----+-----------+
| id | fruitname |
+----+-----------+
|  1 | apple     |
|  2 | orange    |
|  3 | grape     |
|  4 | banana    |
+----+-----------+
4 rows in set (0.00 sec)

mysql> select * from color;
+----+-----------+
| id | colorname |
+----+-----------+
|  1 | red       |
|  2 | orange    |
|  3 | purple    |
|  4 | yellow    |
+----+-----------+
4 rows in set (0.00 sec)
```

When you want to select from both tables at once, there are a few differences in the syntax of the `SELECT` statement. First, you must ensure that all the tables you're using in your query appear in the `FROM` clause of the `SELECT` statement. Using the `fruit` and `color` example, if you simply want to select all columns and rows from both tables, you might think you would use the following `SELECT` statement:

```
mysql> select * from fruit, color;
+----+-----------+----+-----------+
| id | fruitname | id | colorname |
+----+-----------+----+-----------+
|  1 | apple     |  1 | red       |
|  2 | orange    |  1 | red       |
|  3 | grape     |  1 | red       |
|  4 | banana    |  1 | red       |
|  1 | apple     |  2 | orange    |
|  2 | orange    |  2 | orange    |
|  3 | grape     |  2 | orange    |
|  4 | banana    |  2 | orange    |
|  1 | apple     |  3 | purple    |
|  2 | orange    |  3 | purple    |
|  3 | grape     |  3 | purple    |
|  4 | banana    |  3 | purple    |
|  1 | apple     |  4 | yellow    |
|  2 | orange    |  4 | yellow    |
|  3 | grape     |  4 | yellow    |
|  4 | banana    |  4 | yellow    |
+----+-----------+----+-----------+
16 rows in set (0.00 sec)
```

Sixteen rows of repeated information are probably not what you were looking for!

What this query did is literally join a row in the `color` table to each row in the `fruit` table. Because there are four records in the `fruit` table and four entries in the `color` table, that's 16 records returned to you.

When you select from multiple tables, you must build proper `WHERE` clauses to ensure you really `get` what you want. In the case of the `fruit` and `color` tables, what you really want is to see the `fruitname` and `colorname` records from these two tables where the IDs of each match up. This brings us to the next nuance of the query how to indicate exactly which field you want when the fields are named the same in both tables!

Simply, you append the table name to the field name, like this:

```
tablename.fieldname
```

So, the query for selecting `fruitname` and `colorname` from both tables where the IDs match would be

```
mysql> select fruitname, colorname from fruit, color where fruit.id =
color.id;
+-----------+-----------+
| fruitname | colorname |
+-----------+-----------+
| apple     | red       |
| orange    | orange    |
| grape     | purple    |
| banana    | yellow    |
+-----------+-----------+
4 rows in set (0.00 sec)
```

However, if you attempt to select a column that appears in both tables with the same name, you will get an ambiguity error:

```
mysql> select id, fruitname, colorname from fruit, color
    -> where fruit.id = color.id;
ERROR 1052: Column: 'id' in field list is ambiguous
```

If you mean to select the ID from the fruit table, you would use

```
mysql> select fruit.id, fruitname, colorname from fruit,
    -> color where fruit.id = color.id;
+------+-----------+-----------+
| id   | fruitname | colorname |
+------+-----------+-----------+
|    1 | apple     | red       |
|    2 | orange    | orange    |
|    3 | grape     | purple    |
|    4 | banana    | yellow    |
+------+-----------+-----------+
4 rows in set (0.00 sec)
```

This was a basic example of joining two tables together for use in a single `SELECT` query. The `JOIN` keyword is an actual part of SQL, which enables you to build more complex queries.

Using `JOIN`

Several types of `JOIN`s can be used in MySQL, all of which refer to the order in which the tables are put together and the results are displayed. The type of `JOIN` used with the `fruit` and `color` tables is called an `INNER JOIN`, although it wasn't written explicitly as such. To rewrite the SQL statement using the proper `INNER JOIN` syntax, you would use

```
mysql> select fruitname, colorname from fruit
    -> inner join color on fruit.id = color.id;
+-----------+-----------+
| fruitname | colorname |
+-----------+-----------+
| apple     | red       |
| orange    | orange    |
| grape     | purple    |
| banana    | yellow    |
+-----------+-----------+
4 rows in set (0.00 sec)
```

The `ON` clause replaces the `WHERE` clause you've seen before; in this instance it tells MySQL to join together the rows in the tables where the IDs match each other. When joining tables using ON clauses, you can use any conditions that you would use in a `WHERE` clause, including all the various logical and arithmetic operators.

Another common type of `JOIN` is the `LEFT JOIN`. When joining two tables with `LEFT JOIN`, all rows from the first table will be returned, no matter whether there are matches in the second table or not. Suppose you have two tables in an address book, one called `master_name`, containing basic records, and one called `email`, containing email records. Any records in the `email` table would be tied to a particular ID of a record in the `master_name` table. For example, take a look at these two tables:

```
mysql> select name_id, firstname, lastname from master_name;
+---------+-----------+----------+
| name_id | firstname | lastname |
+---------+-----------+----------+
|       1 | John      | Smith    |
|       2 | Jane      | Smith    |
|       3 | Jimbo     | Jones    |
|       4 | Andy      | Smith    |
|       7 | Chris     | Jones    |
|      45 | Anna      | Bell     |
|      44 | Jimmy     | Carr     |
|      43 | Albert    | Smith    |
|      42 | John      | Doe      |
+---------+-----------+----------+
9 rows in set (0.00 sec)
```

```
mysql> select name_id, email from email;
+---------+------------------+
| name_id | email            |
+---------+------------------+
|      42 | jdoe@yahoo.com   |
|      45 | annabell@aol.com |
+---------+------------------+
2 rows in set (0.00 sec)
```

Using `LEFT JOIN` on these two tables, you can see that if a value from the `email` table doesn't exist, a `NULL` value will appear in place of an email address:

```
mysql> select firstname, lastname, email fom master_name
    -> left join email on master_name.name_id = email.name_id;
+-----------+----------+------------------+
| firstname | lastname | email            |
+-----------+----------+------------------+
| John      | Smith    | NULL             |
| Jane      | Smith    | NULL             |
| Jimbo     | Jones    | NULL             |
| Andy      | Smith    | NULL             |
| Chris     | Jones    | NULL             |
| Anna      | Bell     | annabell@aol.com |
| Jimmy     | Carr     | NULL             |
| Albert    | Smith    | NULL             |
| John      | Doe      | jdoe@yahoo.com   |
+-----------+----------+------------------+
9 rows in set (0.01 sec)
```

A `RIGHT JOIN` works like `LEFT JOIN`, but with the table order reversed. In other words, when using a `RIGHT JOIN`, all rows from the second table will be returned, no matter whether there are matches in the first table or not. However, in the case of the `master_name` and `email` tables, there are only two rows in the `email` table, whereas there are nine rows in the `master_name` table. This means that only two of the nine rows will be returned:

```
mysql> select firstname, lastname, email from master_name
    -> right join email on master_name.name_id = email.name_id;
+-----------+----------+------------------+
| firstname | lastname | email            |
+-----------+----------+------------------+
| John      | Doe      | jdoe@yahoo.com   |
| Anna      | Bell     | annabell@aol.com |
+-----------+----------+------------------+
2 rows in set (0.00 sec)
```

Several different types of `JOIN`s are available in MySQL, and you've learned about the most common types. To learn more about `JOIN`s such as `CROSS JOIN`, `STRAIGHT JOIN`, and `NATURAL JOIN`, please visit the MySQL manual at http://dev.mysql.com/doc/J/O/JOIN.html.

Using the `UPDATE` Command to Modify Records

`UPDATE` is the SQL command used to modify the contents of one or more columns in an existing record or set of records. The most basic `UPDATE` syntax looks like this:

```
UPDATE table_name
SET column1='new value',
column2='new value2'
[WHERE some_condition_is_true]
```

The guidelines for updating a record are similar to those used when inserting a record: The data you're entering must be appropriate to the data type of the field, and you must enclose your strings in single or double quotes, escaping where necessary.

For example, assume you have a table called `fruit` containing an ID, a `fruit` name, and the status of the fruit (`ripe` or `rotten`):

```
mysql> SELECT * FROM fruit;
+----+------------+--------+
| id | fruit_name | status |
+----+------------+--------+
|  1 | apple      | ripe   |
|  2 | pear       | rotten |
|  3 | banana     | ripe   |
|  4 | grape      | rotten |
+----+------------+--------+
4 rows in set (0.00 sec)
```

To update the status of the fruit to `ripe`, use

```
mysql> update fruit set status = 'ripe';
Query OK, 2 rows affected (0.00 sec)
Rows matched: 4 Changed: 2 Warnings: 0
```

Take a look at the result of the query. It was successful, as you can tell from the `Query OK` message. Also note that only two rows were affected if you try to set the value of a column to the value it already is, the update won't occur for that column.

The second line of the response shows that four rows were matched, and only two were changed. If you're wondering what matched, the answer is simple: Because you did not specify a particular condition for matching, the match would be `all rows`.

You must be very careful and use a condition when updating a table, unless you really intend to change all the columns for all records to the same value. For the sake of argument, assume that "grape" is spelled incorrectly in the table, and you want to use `UPDATE` to correct this mistake. This query would have horrible results:

```
mysql> update fruit set fruit_name = 'grape';
Query OK, 4 rows affected (0.00 sec)
Rows matched: 4 Changed: 4 Warnings: 0
```

When you read the result, you should be filled with dread: 4 of 4 records were changed, meaning your `fruit` table now looks like this:

```
mysql> SELECT * FROM fruit;
+----+------------+--------+
| id | fruit_name | status |
+----+------------+--------+
|  1 | grape      | ripe   |
|  2 | grape      | ripe   |
|  3 | grape      | ripe   |
|  4 | grape      | ripe   |
+----+------------+--------+
4 rows in set (0.00 sec)
```

All your fruit records are now grapes. While attempting to correct the spelling of one field, all fields were changed because no condition was specified! When doling out `UPDATE` privileges to your users, think about the responsibility you're giving to someone one wrong move and your entire table could be grapes.

Conditional `UPDATE`s

Making a conditional `UPDATE` means that you are using `WHERE` clauses to match specific records. Using a `WHERE` clause in an `UPDATE` statement is just like using a `WHERE` clause in a `SELECT` statement. All the same comparison and logical operators can be used, such as `equal to, greater than, OR, AND`.

Assume your `fruit` table has not been completely filled with grapes, but instead contains four records, one with a spelling mistake (`grappe` instead of `grape`). The `UPDATE` statement to fix the spelling mistake would be

```
mysql> update fruit set fruit_name = 'grape' where fruit_name =
'grappe';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

In this case, only one row was matched and one row was changed. Your `fruit` table should be intact, and all fruit names should be spelled properly:

```
mysql> select * from fruit;
+----+------------+--------+
| id | fruit_name | status |
+----+------------+--------+
|  1 | apple      | ripe   |
|  2 | pear       | ripe   |
|  3 | banana     | ripe   |
|  4 | grape      | ripe   |
+----+------------+--------+
4 rows in set (0.00 sec)
```

Using Existing Column Values with `UPDATE`

Another feature of `UPDATE` is the capability to use the current value in the record as the base value. For example, go back to the `grocery_inventory` table used earlier in this lesson:

```
mysql> select * from grocery_inventory;
+----+-------------+-----------------------+------------+----------+
| id | item_name   | item_desc             | item_price | curr_qty |
+----+-------------+-----------------------+------------+----------+
|  1 | Apples      | Beautiful, ripe apples.|       0.25 |     1000 |
|  2 | Bunches of Grapes | Seedless grapes. |       2.99 |      500 |
|  3 | Bottled Water (6-pack)| 500ml spring water. |2.29 |      250 |
|  4 | Bananas     | Bunches, green.       |       1.99 |      150 |
|  5 | Pears       | Anjou, nice and sweet. |        0.5 |      500 |
|  6 | Avocado     | Large Haas variety.   |       0.99 |      750 |
+----+-------------+-----------------------+------------+----------+
6 rows in set (0.00 sec)
```

When someone purchases an apple, the inventory table should be updated accordingly. However, you won't know exactly what number to enter in the `curr_qty` column, just that you sold one. In this case, use the current value of the column and subtract one:

```
mysql> update grocery_inventory set curr_qty = curr_qty - 1 where id
= 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

This should give you a new value of `999` in the `curr_qty` column, and indeed it does:

```
mysql> select * from grocery_inventory;
+----+------------+-----------------------+------------+----------+
| id | item_name  | item_desc             | item_price | curr_qty |
+----+------------+-----------------------+------------+----------+
|  1 | Apples     | Beautiful, ripe apples.|       0.25 |      999 |
|  2 | Bunches of Grapes | Seedless grapes.|       2.99 |      500 |
|  3 | Bottled Water (6-pack)| 500ml spring water.|2.29 |      250 |
|  4 | Bananas    | Bunches, green.       |       1.99 |      150 |
|  5 | Pears      | Anjou, nice and sweet. |        0.5 |      500 |
|  6 | Avocado    | Large Haas variety.   |       0.99 |      750 |
+----+------------+-----------------------+------------+----------+
```

6 rows in set (0.00 sec)

Using the `REPLACE` Command

Another method for modifying records is to use the `REPLACE` command, which is remarkably similar to the `INSERT` command.

```
REPLACE INTO table_name (column list) VALUES (column values);
```

The `REPLACE` statement works like this: If the record you are inserting into the table contains a primary key value that matches a record already in the table, the record in the table will be deleted and the new record inserted in its place.

NOTE:

The `REPLACE` command is a MySQL-specific extension to ANSI SQL. This command mimics the action of a `DELETE` and re-`INSERT` of a particular record. In other words, you get two commands for the price of one.

Using the `grocery_inventory` table, the following command will replace the entry for Apples:

```
mysql> replace into grocery_inventory values
    -> (1, 'Granny Smith Apples', 'Sweet!', '0.50', 1000);
Query OK, 2 rows affected (0.00 sec)
```

In the query result, notice that the result states, `2 rows affected`. In this case because `id` is a primary key that had a matching value in the `grocery_inventory` table, the original row was deleted and the new row inserted`2 rows affected`.

Select the records to verify that the entry is correct, which it is

```
mysql> select * from grocery_inventory;
+----+--------------+-----------------------+------------+-----------+
| id | item_name    | item_desc             | item_price | curr_qty |
+----+--------------+-----------------------+------------+-----------+
|  1 | Granny Smith Apples   | Sweet!       |        0.5 |     1000 |
|  2 | Bunches of Grapes | Seedless grapes. |       2.99 |      500 |
|  3 | Bottled Water (6-pack)| 500ml spring water.| 2.29 |      250 |
|  4 | Bananas               | Bunches, green.|     1.99 |      150 |
|  5 | Pears        | Anjou, nice and sweet.|        0.5 |      500 |
|  6 | Avocado      | Large Haas variety.   |       0.99 |      750 |
+----+--------------+-----------------------+------------+-----------+
6 rows in set (0.00 sec)
```

If you use a `REPLACE` statement, and the value of the primary key in the new record does not match a value for a primary key already in the table, the record would simply be inserted and only one row would be affected.

Using the DELETE Command

The basic DELETE syntax is

```
DELETE FROM table_name
[WHERE some_condition_is_true]
[LIMIT rows]
```

Notice there is no column specification used in the DELETE commandwhen you use DELETE, the entire record is removed. You might recall the fiasco earlier in this lesson, regarding grapes in the fruit table, when updating a table without specifying a condition caused all records to be updated. You must be similarly careful when using DELETE.

Assuming the structure and data in a table called fruit:

```
mysql> select * from fruit;
+----+------------+--------+
| id | fruit_name | status |
+----+------------+--------+
|  1 | apple      | ripe   |
|  2 | pear       | rotten |
|  3 | banana     | ripe   |
|  4 | grape      | rotten |
+----+------------+--------+
4 rows in set (0.00 sec)
```

This statement will remove all records in the table:

```
mysql> delete from fruit;
Query OK, 0 rows affected (0.00 sec)
```

You can always verify the deletion by attempting to SELECT data from the table. If you issued this command after removing all the records:

```
mysql> select * from fruit;
Empty set (0.00 sec)
```

You would see that all your fruit is gone.

Conditional DELETE

A conditional DELETE statement, just like a conditional SELECT or UPDATE statement, means you are using WHERE clauses to match specific records. You have the full range of comparison and logical operators available to you, so you can pick and choose which records you want to delete.

A prime example would be to remove all records for rotten fruit from the fruit table:

```
mysql> delete from fruit where status = 'rotten';
Query OK, 2 rows affected (0.00 sec)
```

Two records were deleted, and only ripe fruit remains:

```
mysql> select * from fruit;
+----+------------+--------+
| id | fruit_name | status |
+----+------------+--------+
|  1 | apple      | ripe   |
|  3 | banana     | ripe   |
+----+------------+--------+
2 rows in set (0.00 sec)
```

Since MySQL 4.0.x, users can also use `ORDER BY` clauses in your `DELETE` statements. Take a look at the basic `DELETE` syntax with the `ORDER BY` clause added to its structure:

```
DELETE FROM table_name
[WHERE some_condition_is_true]
[ORDER BY some_column [ASC | DESC]]
[LIMIT rows]
```

At first glance, you might wonder, "Why does it matter in what order I delete records?" The `ORDER BY` clause isn't for the deletion order, it's for the sorting order of records.

In this example, a table called `access_log` shows access time and username:

```
mysql> select * from access_log;
+----+---------------------+----------+
| id | date_accessed       | username |
+----+---------------------+----------+
|  1 | 2001-11-06 06:09:13 | johndoe  |
|  2 | 2001-11-06 06:09:22 | janedoe  |
|  3 | 2001-11-06 06:09:39 | jsmith   |
|  4 | 2001-11-06 06:09:44 | mikew    |
+----+---------------------+----------+
4 rows in set (0.00 sec)
```

To remove the oldest record, first use `ORDER BY` to sort the results appropriately, and then use `LIMIT` to remove just one record:

```
mysql> delete from access_log order by date_accessed desc limit 1;
Query OK, 1 row affected (0.01 sec)
```

Select the record from `access_log` and verify that only three records exist:

```
mysql> select * from access_log;
+----+---------------------+----------+
| id | date_accessed       | username |
+----+---------------------+----------+
|  2 | 2001-11-06 06:09:22 | janedoe  |
|  3 | 2001-11-06 06:09:39 | jsmith   |
|  4 | 2001-11-06 06:09:44 | mikew    |
+----+---------------------+----------+
```

3 rows in set (0.00 sec)

Frequently Used String Functions in MySQL

MySQL's built-in string-related functions can be used several ways. You can use functions in `SELECT` statements without specifying a table to retrieve a result of the function. Or you can use functions to enhance your `SELECT` results by concatenating two fields to form a new string.

Length and Concatenation Functions

The group of length and concatenation functions focuses on the length of strings and concatenating strings together. Length-related functions include `LENGTH()`, `OCTET_LENGTH()`, `CHAR_LENGTH()`, and `CHARACTER_LENGTH()`, which do virtually the same thing: count characters in a string.

```
mysql> select length('This is cool!');
+-------------------------+
| LENGTH('This is cool!') |
+-------------------------+
|                      13 |
+-------------------------+
1 row in set (0.00 sec)
```

The fun begins with the `CONCAT()` function, which is used to concatenate two or more strings:

```
mysql> select concat('My', 'S', 'QL');
+-------------------------+
| CONCAT('My', 'S', 'QL') |
+-------------------------+
| MySQL                   |
+-------------------------+
1 row in set (0.00 sec)
```

Imagine using this function with a table containing names, split into `firstname` and `lastname` fields. Instead of using two strings, use two field names to concatenate the `firstname` and the `lastname` fields. By concatenating the fields, you reduce the lines of code necessary to achieve the same result in your application:

```
mysql> select concat(firstname, lastname) from table_name;
+-----------------------------+
| CONCAT(firstname, lastname) |
+-----------------------------+
| JohnSmith                   |
| JaneSmith                   |
| JimboJones                  |
| AndySmith                   |
| ChrisJones                  |
| AnnaBell                    |
| JimmyCarr                   |
| AlbertSmith                 |
| JohnDoe                     |
+-----------------------------+
9 rows in set (0.00 sec)
```

NOTE:

If you're using a field name and not a string in a function, don't enclose the field name within quotation marks. If you do, MySQL will interpret the string literally. In the `CONCAT()` example, you would get the following result:

```
mysql> select concat('firstname', 'lastname') FROM table_name;
+---------------------------------+
| CONCAT('firstname', 'lastname') |
+---------------------------------+
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
| firstnamelastname               |
+---------------------------------+
9 rows in set (0.00 sec)
```

The `CONCAT()` function would be useful if there were some sort of separator between the names, and that's where the next function comes in: `CONCAT_WS()`.

As you may have figured out, `CONTACT_WS()` stands for concatenate with separator. The separator can be anything you choose, but the following example uses white space:

```
mysql> select concat_ws(' ', firstname, lastname) FROM table_name;
+-------------------------------------+
| CONCAT_WS(' ', firstname, lastname) |
+-------------------------------------+
| John Smith                          |
| Jane Smith                          |
| Jimbo Jones                         |
| Andy Smith                          |
| Chris Jones                         |
| Anna Bell                           |
| Jimmy Carr                          |
| Albert Smith                        |
| John Doe                            |
+-------------------------------------+
9 rows in set (0.00 sec)
```

If you want to shorten the width of your result table, you can use `AS` to name the custom result field:

```
mysql> select concat_ws(' ', firstname, lastname) AS fullname FROM
table_name;
+--------------+
| fullname     |
+--------------+
| John Smith   |
| Jane Smith   |
| Jimbo Jones  |
| Andy Smith   |
| Chris Jones  |
| Anna Bell    |
| Jimmy Carr   |
| Albert Smith |
| John Doe     |
+--------------+
9 rows in set (0.00 sec)
```

Trimming and Padding Functions

MySQL provides several functions for adding and removing extra characters (including whitespace) from strings. The `RTRIM()` and `LTRIM()` functions remove whitespace from either the right or left side of a string:

```
mysql> select rtrim('stringstring ');
+--------------------------+
| RTRIM('stringstring   ') |
+--------------------------+
| stringstring             |
+--------------------------+
1 row in set (0.00 sec)

mysql> select ltrim(' stringstring');
+-------------------------+
| LTRIM(' stringstring')  |
+-------------------------+
| stringstring            |
+-------------------------+
1 row in set (0.00 sec)
```

You may have padded strings to trim if the string is coming out of a fixed-width field, and either doesn't need to carry along the additional padding or is being inserted into a `varchar` or other non fixed-width field. If your strings are padded with a character besides white space, use the `trIM()` function to name the characters you want to remove. For example, to remove the leading `X` characters from the string `XXXneedleXXX`, use

```
mysql> select trim(leading 'X' from 'XXXneedleXXX');
+---------------------------------------+
| TRIM(LEADING 'X' from 'XXXneedleXXX') |
+---------------------------------------+
| needleXXX                             |
+---------------------------------------+
1 row in set (0.00 sec)
```

Use `TRAILING` to remove the characters from the end of the string:

```
mysql> select trim(trailing 'X' from 'XXXneedleXXX');
+----------------------------------------+
| TRIM(TRAILING 'X' from 'XXXneedleXXX') |
+----------------------------------------+
| XXXneedle                              |
+----------------------------------------+
1 row in set (0.00 sec)
```

If neither `LEADING` nor `TRAILING` is indicated, both are assumed:

```
mysql> select trim('X' from 'XXXneedleXXX');
+-------------------------------+
| TRIM('X' from 'XXXneedleXXX') |
+-------------------------------+
| needle                        |
+-------------------------------+
1 row in set (0.00 sec)
```

Just like `RTRIM()` and `LTRIM()` remove padding characters, `RPAD()` and `LPAD()` add characters to a string. For example, you may want to add specific identification characters to a string that is part of an order number, in a database used for sales. When you use the padding functions, the required elements are the string, the target length, and the padding character. For example, pad the string `needle` with the `X` character until the string is 10 characters long:

```
mysql> select rpad('needle', 10, 'X');
+--------------------------+
| RPAD('needle', 10, 'X') |
+--------------------------+
| needleXXXX               |
+--------------------------+
1 row in set (0.00 sec)

mysql> select lpad('needle', 10, 'X');
+--------------------------+
| LPAD('needle', 10, 'X') |
+--------------------------+
| XXXXneedle               |
+--------------------------+
1 row in set (0.00 sec)
```

Location and Position Functions

The group of location and position functions is useful for finding parts of strings within other strings. The `LOCATE()` function returns the position of the first occurrence of a given substring within the target string. For example, you can look for a needle in a haystack:

```
mysql> select locate('needle', 'haystackneedlehaystack');
+-------------------------------------------+
| LOCATE('needle', 'haystackneedlehaystack') |
+-------------------------------------------+
|                                         9 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

The substring `needle` begins at position 9 in the target string. If the substring cannot be found in the target string, MySQL returns 0 as a result.

NOTE:

Unlike position counting within most programming languages, which starts at 0, position counting using MySQL starts at 1.

An extension of the `LOCATE()` function is to use a third argument for starting position. If you start looking for `needle` in `haystack` before position 9, you'll receive a result. Otherwise, because `needle` starts at position 9, you'll receive a 0 result if you specify a greater starting position:

```
mysql> select locate('needle', 'haystackneedlehaystack',6);
+-----------------------------------------------+
| LOCATE('needle', 'haystackneedlehaystack',9) |
+-----------------------------------------------+
|                                             9 |
+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> select locate('needle', 'haystackneedlehaystack',12);
+------------------------------------------------+
| LOCATE('needle', 'haystackneedlehaystack',12) |
+------------------------------------------------+
|                                              0 |
+------------------------------------------------+
1 row in set (0.00 sec)
```

Substring Functions

If your goal is to extract a substring from a target string, several functions fit the bill. Given a string, starting position, and length, you can use the `SUBSTRING()` function. This example gets three characters from the string `MySQL`, starting at position 2:

```
mysql> select substring("MySQL", 2, 3);
+--------------------------+
| SUBSTRING("MySQL", 2, 3) |
+--------------------------+
| ySQ                      |
+--------------------------+
1 row in set (0.00 sec)
```

If you just want a few characters from the left or right ends of a string, use the `LEFT()` and `RIGHT()` functions:

```
mysql> select left("MySQL", 2);
+------------------+
| LEFT("MySQL", 2) |
+------------------+
| My               |
+------------------+
1 row in set (0.00 sec)

mysql> select right("MySQL", 3);
+-------------------+
| RIGHT("MySQL", 3) |
+-------------------+
| SQL               |
+-------------------+
1 row in set (0.00 sec)
```

One of the many common uses of substring functions is to extract parts of order numbers, to find out who placed the order. In some applications, the system is designed to automatically generate an order number, containing a date, customer identification, and other information. If this order number always follows a particular

pattern, such as `XXXX-YYYY-ZZ`, you can use substring functions to extract the individual parts of the whole. For example, if `ZZ` always represents the state to which the order was shipped, you can use the `RIGHT()` function to extract these characters and report the number of orders shipped to a particular state.

String Modification Functions

Your programming language of choice likely has functions to modify the appearance of strings, but if you can perform the task as part of the SQL statement, all the better.

The MySQL `LCASE()` and `UCASE()` functions transform a string into lowercase or uppercase:

```
mysql> select lcase('MYSQL');
+----------------+
| LCASE('MYSQL') |
+----------------+
| mysql          |
+----------------+
1 row in set (0.00 sec)

mysql> select ucase('mysql');
+----------------+
| UCASE('mysql') |
+----------------+
| MYSQL          |
+----------------+
1 row in set (0.00 sec)
```

Remember, if you use the functions with field names, don't use quotation marks:

```
mysql> select ucase(lastname) from table_name;
+-----------------+
| UCASE(lastname) |
+-----------------+
| BELL            |
| CARR            |
| DOE             |
| JONES           |
| JONES           |
| SMITH           |
| SMITH           |
| SMITH           |
| SMITH           |
+-----------------+
9 rows in set (0.00 sec)
```

Another fun string-manipulation function is `REPEAT()`, which does just what it sounds like repeats a string for a given number of times:

```
mysql> select repeat("bowwow", 4);
+--------------------------+
| REPEAT("bowwow", 4)      |
+--------------------------+
```

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

The `REPLACE()` function replaces all occurrences of a given string with another string:

```
mysql> select replace('bowwowbowwowbowwowbowwow', 'wow', 'WOW');
+-------------------------------------------------+
| REPLACE('bowwowbowwowbowwowbowwow', 'wow', 'WOW') |
+-------------------------------------------------+
| bowWOWbowWOWbowWOWbowWOW                        |
+-------------------------------------------------+
1 row in set (0.00 sec)
```

Using Date and Time Functions in MySQL

MySQL's built-in date-related functions can be used in `SELECT` statements, with or without specifying a table, to retrieve a result of the function. Or you can use the functions with any type of date field: `date`, `datetime`, `timestamp`, and `year`. Depending on the type of field in use, the results of the date-related functions are more or less useful.

Working with Days

The `DAYOFWEEK()` and `WEEKDAY()` functions do similar things with slightly different results. Both functions are used to find the weekday index of a date, but the difference lies in the starting day and position.

If you use `DAYOFWEEK()`, the first day of the week is Sunday, at position 1, and the last day of the week is Saturday, at position 7. For example

```
mysql> select dayofweek('2004-08-23');
+--------------------------+
| dayofweek('2004-08-23') |
+--------------------------+
|                        2 |
+--------------------------+
1 row in set (0.00 sec)
```

The result shows that August 23, 2004 was weekday index 2, or Monday. Using the same date with `WEEKDAY()` gives you a different result with the same meaning:

```
mysql> select weekday('2004-08-23');
+-----------------------+
| WEEKDAY('2004-08-23') |
+-----------------------+
|                     0 |
+-----------------------+
1 row in set (0.00 sec)
```

The result shows that August 23, 2004 was weekday index 0. Because `WEEKDAY()` uses Monday as the first day of the week at position 0 and Sunday as the last day at position 6, 0 is accurate: Monday.

The `DAYOFMONTH()` and `DAYOFYEAR()` functions are more straightforward, with only one result and a range that starts at 1 and ends at 31 for `DAYOFMONTH()` and 366 for `DAYOFYEAR()`. Some examples follow:

```
mysql> select dayofmonth('2004-08-23');
+--------------------------+
| DAYOFMONTH('2004-08-23') |
+--------------------------+
|                       23 |
+--------------------------+
1 row in set (0.00 sec)

mysql> select dayofyear('2004-08-23');
+-------------------------+
| DAYOFYEAR('2004-08-23') |
+-------------------------+
|                     236 |
+-------------------------+
1 row in set (0.00 sec)
```

It might seem odd to have a function that returns the day of the month on a particular date because the day is right there in the string. But think about using these types of functions in `WHERE` clauses to perform comparisons on records. If you have a table that holds online orders with a field containing the date the order was placed, you can quickly get a count of the orders placed on any given day of the week, or see how many orders were placed during the first half of the month versus the second half.

The following two queries show how many orders were placed during the first three days of the week (throughout all months) and then the remaining days of the week:

```
mysql> select count(id) from orders where dayofweek(date_ordered) <
4;
+-----------+
| COUNT(id) |
+-----------+
|         3 |
+-----------+
1 row in set (0.00 sec)

mysql> select count(id) from orders where dayofweek(date_ordered) >
3;
+-----------+
| COUNT(id) |
+-----------+
|         5 |
+-----------+
1 row in set (0.00 sec)
```

Using `DAYOFMONTH()`, the following examples show the number of orders placed

during the first half of any month versus the second half:

```
mysql> select count(id) from orders where dayofmonth(date_ordered) <
16;
+-----------+
| COUNT(id) |
+-----------+
|         6 |
+-----------+
1 row in set (0.00 sec)

mysql> select count(id) from orders where dayofmonth(date_ordered) >
15;
+-----------+
| COUNT(id) |
+-----------+
|         2 |
+-----------+
1 row in set (0.00 sec)
```

You can use the `DAYNAME()` function to add more life to your results because it returns the name of the weekday for any given date:

```
mysql> select dayname(date_ordered) from orders;
+-----------------------+
| DAYNAME(date_ordered) |
+-----------------------+
| Thursday              |
| Monday                |
| Thursday              |
| Thursday              |
| Wednesday             |
| Thursday              |
| Sunday                |
| Sunday                |
+-----------------------+
8 rows in set (0.00 sec)
```

Functions aren't limited to `WHERE` clausesyou can use them in `ORDER BY` clauses as well:

```
mysql> select dayname(date_ordered) from orders
    -> order by dayofweek(date_ordered);
+-----------------------+
| DAYNAME(date_ordered) |
+-----------------------+
| Sunday                |
| Sunday                |
| Monday                |
| Wednesday             |
| Thursday              |
| Thursday              |
| Thursday              |
| Thursday              |
+-----------------------+
8 rows in set (0.00 sec)
```

Working with Months and Years

Days of the week aren't the only parts of the calendar, and MySQL has functions specifically for months and years as well. Just like the `DAYOFWEEK()` and `DAYNAME()` functions, `MONTH()` and `MONTHNAME()` return the number of the month in a year and the name of the month for a given date. For example

```
mysql> select month('2004-08-23'), monthname('2004-08-23');
+---------------------+-------------------------+
| month('2004-08-23') | monthname('2004-08-23') |
+---------------------+-------------------------+
|                   8 | August                  |
+---------------------+-------------------------+
1 row in set (0.00 sec)
```

Using `MONTHNAME()` on the `orders` table shows the proper results but a lot of repeated data:

```
mysql> select monthname(date_ordered) from orders;
+-------------------------+
| MONTHNAME(date_ordered) |
+-------------------------+
| November                |
| November                |
| November                |
| November                |
| November                |
| November                |
| November                |
| October                 |
+-------------------------+
8 rows in set (0.00 sec)
```

You can use `DISTINCT` to get nonrepetitive results:

```
mysql> select distinct monthname(date_ordered) from orders;
+-------------------------+
| MONTHNAME(date_ordered) |
+-------------------------+
| November                |
| October                 |
+-------------------------+
2 rows in set (0.00 sec)
```

For work with years, the `YEAR()` function will return the year of a given date:

```
mysql> select distinct year(date_ordered) from orders;
+--------------------+
| YEAR(date_ordered) |
+--------------------+
|               2003 |
+--------------------+
```

```
1 row in set (0.00 sec)
```

Working with Weeks

Weeks can be tricky things there can be 53 weeks in a year if Sunday is the first day of the week and December hasn't ended. For example, December 30th of 2001 was a Sunday:

```
mysql> select dayname('2001-12-30');
+-----------------------+
| DAYNAME('2001-12-30') |
+-----------------------+
| Sunday                |
+-----------------------+
1 row in set (0.00 sec)
```

That fact made December 30 of 2001 part of the 53rd week of the year:

```
mysql> select week('2001-12-30');
+--------------------+
| WEEK('2001-12-30') |
+--------------------+
|                 53 |
+--------------------+
1 row in set (0.00 sec)
```

The 53rd week contains December 30 and 31, and is only two days long; the first week of 2002 begins with January 1.

If you want your weeks to start on Mondays but still want to find the week of the year, the optional second argument enables you to change the start day. A `1` indicates a week that starts on Monday. In the following examples, a Monday start day makes December 30 part of the 52nd week of 2001, but December 31 is still part of the 53rd week of 2001.

```
mysql> select week('2001-12-30',1);
+----------------------+
| WEEK('2001-12-30',1) |
+----------------------+
|                   52 |
+----------------------+
1 row in set (0.00 sec)

mysql> select week('2001-12-31',1);
+----------------------+
| WEEK('2001-12-31',1) |
+----------------------+
|                   53 |
+----------------------+
1 row in set (0.00 sec)
```

Working with Hours, Minutes, and Seconds

If you're using a date that includes the exact time, such as `datetime` or `timestamp`, or even just a `time` field, there are functions to find the hours, minutes, and seconds from that string. Not surprisingly, these functions are called `HOUR()`, `MINUTE()`, and `SECOND()`. `HOUR()` returns the hour in a given time, which is between 0 and 23. The range for `MINUTE()` and `SECOND()` is 0 to 59.

Here are some examples:

```
mysql> select hour('2004-08-25 07:27:49') as hour,minute('2004-08-25
07:27:49')
    -> as minute,second('2004-08-25 07:27:49') as second;
+------+--------+--------+
| hour | minute | second |
+------+--------+--------+
|    7 |     27 |     49 |
+------+--------+--------+
1 row in set (0.00 sec)
```

That's a lot of queries to get at one time from a `datetime` fieldyou can put the hour and minute together and even use `CONCAT_WS()` to put the `:` between the results and get a representation of the time:

```
mysql> select concat_ws(':',hour('2004-08-25 07:27:49'),
    -> minute('2004-08-25 07:27:49')) as sample_time;
+-------------+
| sample_time |
+-------------+
| 7:27        |
+-------------+
1 row in set (0.00 sec)
```

If you use field names instead of strings, remember not to use quotation marks. Here's an example that uses the `dateadded` field from the `sometable` table:

```
mysql> select concat_ws(':',hour(dateadded), minute(dateadded))
    -> as sample_time from sometable;
+-------------+
| sample_time |
+-------------+
| 13:11       |
| 13:11       |
| 13:11       |
| 13:11       |
| 14:16       |
| 10:12       |
| 10:12       |
| 10:12       |
| 10:12       |
+-------------+
9 rows in set (0.00 sec)
```

This is cheating because it's not the actual time it's just two numbers stuck together to

look like a time. If you used the concatenation trick on a time such as `02:02`, the result would be `2:2`, as shown here:

```
mysql> select concat_ws(':',hour('02:02'), minute('02:02')) as
sample_time;
+-------------+
| sample_time |
+-------------+
| 2:2         |
+-------------+
1 row in set (0.00 sec)
```

This result is obviously not the intended result. In the next section, you learn how to use the `DATE_FORMAT()` function to properly format dates and times.

Formatting Dates and Times with MySQL

The `DATE_FORMAT()` function formats a `date`, `datetime`, or `timestamp` field into a string by using options that tell it exactly how to display the results. The syntax of `DATE_FORMAT()` is

```
DATE_FORMAT(date,format)
```

There are many formatting options, as shown in Table 2.

Table 2. `DATE_FORMAT()` Format String Options

Option	Result
`%M`	Month name (January through December)
`%b`	Abbreviated month name (Jan through Dec)
`%m`	Month, padded digits (01 through 12)
`%c`	Month (1 through 12)
`%W`	Weekday name (Sunday through Saturday)
`%a`	Abbreviated weekday name (Sun through Sat)
`%D`	Day of the month using the English suffix, such as first, second, third, and so on
`%d`	Day of the month, padded digits (00 through 31)
`%e`	Day of the month (0 through 31)
`%j`	Day of the year, padded digits (001 through 366)
`%Y`	Year, four digits
`%y`	Year, two digits
`%X`	Four-digit year for the week where Sunday is the first day; used with `%V`
`%x`	Four-digit year for the week where Monday is the first day; used with `%v`
`%w`	Day of the week (0=Sunday...6=Saturday)

%U	Week (0 through 53) where Sunday is the first day of the week
%u	Week (0 through 53) where Monday is the first day of the week
%V	Week (1 through 53) where Sunday is the first day of the week; used with %X
%v	Week (1 through 53) where Monday is the first day of the week; used with %x
%H	Hour, padded digits (00 through 23)
%k	Hour (0 through 23)
%h	Hour, padded digits (01 through 12)
%l	Hour (1 through 12)
%i	Minutes, padded digits (00 through 59)
%S	Seconds, padded digits (00 through 59)
%s	Seconds, padded digits (00 through 59)
%r	Time, 12-hour clock (hh:mm:ss [AP]M)
%T	Time, 24-hour clock (hh:mm:ss)
%p	AM or PM

NOTE:

Any other characters used in the `DATE_FORMAT()` option string appear literally.

To display the `02:02` result that we rigged in the previous section, you'd use the `%h` and `%i` options to return the hour and minute from the date with a `:` between the two options. For example

```
mysql> select date_format('2004-08-25 02:02:00', '%h:%i') as
sample_time;
+-------------+
| sample_time |
+-------------+
| 02:02       |
+-------------+
1 row in set (0.00 sec)
```

The following are just a few more examples of the `DATE_FORMAT()` function in use, but this function is best understood by practicing it yourself.

```
mysql> select date_format('2004-08-23', '%W, %M %D, %Y') as
sample_time;
+------------------------------+
| sample_time                  |
+------------------------------+
| Monday, August 23rd, 2004    |
+------------------------------+
1 row in set (0.00 sec)

mysql> select date_format(now(),'%W the %D of %M, %Y around %l
o\'clock %p')
    -> as sample_time;
+-----------------------------------------------------------+
| sample_time                                               |
+-----------------------------------------------------------+
| Monday the 23rd of August, 2004 around 8 o'clock AM       |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

If you're working specifically with time fields, the `TIME_FORMAT()` function works just like the `DATE_FORMAT()` function. Only the format options for hours, minutes, and seconds are allowed:

```
mysql> select time_format('02:02:00', '%h:%i') as sample_time;
+-------------+
| sample_time |
+-------------+
| 02:02       |
+-------------+
1 row in set (0.00 sec)
```

Performing Date Arithmetic with MySQL

MySQL has several functions to help perform date arithmetic, and this is one of the areas where it might be quicker to allow MySQL to do the math than your PHP script. The `DATE_ADD()` and `DATE_SUB()` functions return a result given a starting date and an interval. The syntax for both functions is

```
DATE_ADD(date,INTERVAL value type)

DATE_SUB(date,INTERVAL value type)
```

Table 3 shows the possible types and their expected value format.

Table 3. Values and Types in Date Arithmetic

Value	Type
Number of seconds	SECOND
Number of minutes	MINUTE
Number of hours	HOUR
Number of days	DAY
Number of months	MONTH
Number of years	YEAR
"minutes:seconds"	MINUTE_SECOND
"hours:minutes"	HOUR_MINUTE
"days hours"	DAY_HOUR
"years-months"	YEAR_MONTH
"hours:minutes:seconds"	HOUR_SECOND
"days hours:minutes"	DAY_MINUTE
"days hours:minutes:seconds"	DAY_SECOND

For example, to find the date of the current day plus 21 days, use the following:

```
mysql> select date_add(now(), interval 21 day);
+----------------------------------+
| date_add(now(), interval 21 day) |
+----------------------------------+
| 2004-09-17 13:07:34              |
+----------------------------------+
1 row in set (0.00 sec)
```

To subtract 21 days, use

```
mysql> select date_sub(now(), interval 21 day);
+----------------------------------+
| date_sub(now(), interval 21 day) |
+----------------------------------+
| 2004-08-06 13:07:49              |
+----------------------------------+
1 row in set (0.00 sec)
```

Use the expression as it's shown in Table 3, despite what might be a natural tendency to use `DAYS` instead of `DAY`. Using `DAYS` results in an error:

```
mysql> select date_add(now(), interval 21 days);
ERROR 1064: You have an error in your SQL syntax near 'days)' at line
1
```

If you're using `DATE_ADD()` or `DATE_SUB()` with a `date` value instead of a `datetime` value, the result will be shown as a `date` value unless you use expressions related to hours, minutes, and seconds. In that case, your result will be a `datetime` result.

For example, the result of the first query remains a `date` field, whereas the second becomes a `datetime`:

```
mysql> select date_add("2001-12-31", interval 1 day);
+----------------------------------------+
| DATE_ADD("2001-12-31", INTERVAL 1 DAY) |
+----------------------------------------+
| 2002-01-01                             |
+----------------------------------------+
1 row in set (0.00 sec)

mysql> select date_add("2001-12-31", interval 12 hour);
+-----------------------------------------+
| DATE_ADD("2001-12-31", INTERVAL 12 HOUR) |
+-----------------------------------------+
| 2001-12-31 12:00:00                      |
+-----------------------------------------+
1 row in set (0.00 sec)
```

Beginning with MySQL version 3.23, you can also perform date arithmetic using the `+` and `-` operators instead of `DATE_ADD()` and `DATE_SUB()` functions:

```
mysql> select "2001-12-31" + interval 1 day;
+-------------------------------+
| "2001-12-31" + INTERVAL 1 DAY |
+-------------------------------+
| 2002-01-01                    |
+-------------------------------+
1 row in set (0.00 sec)

mysql> select "2001-12-31" - interval 14 hour;
+---------------------------------+
| "2001-12-31" - INTERVAL 14 HOUR |
+---------------------------------+
| 2001-12-30 10:00:00             |
+---------------------------------+
1 row in set (0.00 sec)
```

Special Functions and Conversion Features

The MySQL `NOW()` function returns a current `datetime` result, and is useful for timestamping login or access times, as well as numerous other tasks. MySQL has a few other functions that perform similarly.

The `CURDATE()` and `CURRENT_DATE()` functions are synonymous, and each returns the current date in `YYYY-MM-DD` format:

```
mysql> select curdate(), current_date();
+------------+----------------+
| curdate()  | current_date() |
+------------+----------------+
| 2004-08-25 | 2004-08-25     |
+------------+----------------+
1 row in set (0.01 sec)
```

Similarly, the `CURTIME()` and `CURRENT_TIME()` functions return the current time in `HH:MM:SS` format:

```
mysql> select curtime(), current_time();
+-----------+----------------+
| curtime() | current_time() |
+-----------+----------------+
| 09:14:26  | 09:14:26       |
+-----------+----------------+
1 row in set (0.00 sec)
```

The `NOW()`, `SYSDATE()`, and `CURRENT_TIMESTAMP()` functions return values in full `datetime` format (`YYYY-MM-DD HH:MM:SS`):

```
mysql> select now(), sysdate(), current_timestamp();
+---------------------+---------------------+---------------------+
| now()               | sysdate()           | current_timestamp() |
+---------------------+---------------------+---------------------+
| 2004-08-25 09:14:50 | 2004-08-25 09:14:50 | 2004-08-25 09:14:50 |
+---------------------+---------------------+---------------------+
1 row in set (0.00 sec)
```

The `UNIX_TIMESTAMP()` function returns the current date inor converts a given date toUnix timestamp format. Unix timestamp format is in seconds since the epoch, or seconds since midnight, January 1, 1970. For example

```
mysql> select unix_timestamp();
+------------------+
| UNIX_TIMESTAMP() |
+------------------+
|       1093637311|
+------------------+
1 row in set (0.00 sec)
```

```
mysql> select unix_timestamp('1973-12-30');
+------------------------------+
| UNIX_TIMESTAMP('1973-12-30') |
+------------------------------+
|                    126086400 |
+------------------------------+
1 row in set (0.00 sec)
```

The `FROM_UNIXTIME()` function performs a conversion of a Unix timestamp to a full `datetime` format when used without any options:

```
mysql> select from_unixtime(' 1093637311');
+------------------------------+
| from_unixtime('1061767446') |
+------------------------------+
| 2004-08-27 13:08:31         |
+------------------------------+
1 row in set (0.00 sec)
```

You can use the format options from the `DATE_FORMAT()` functions to display a timestamp in a more appealing manner:

```
mysql> select from_unixtime(unix_timestamp(), '%D %M %Y at
%h:%i:%s');
+-----------------------------------------------------------+
| from_unixtime(unix_timestamp(), '%D %M %Y at %h:%i:%s') |
+-----------------------------------------------------------+
| 27th August 2004 at 01:09:20                              |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

If you're working with a number of seconds and want to convert the seconds to a time-formatted result, you can use `SEC_TO_TIME()` and `TIME_TO_SEC()` to convert values back and forth.

For example, 1440 seconds is equal to 24 minutes and vice versa:

```
mysql> select sec_to_time('1440'), time_to_sec('00:24:00');
+---------------------+-------------------------+
| SEC_TO_TIME('1440') | TIME_TO_SEC('00:24:00') |
+---------------------+-------------------------+
| 00:24:00            |                    1440 |
+---------------------+-------------------------+
1 row in set (0.01 sec)
```

Q&A

Q What characters can I use to name my tables and fields, and what is the character limit?

A The maximum length of database, table, or field names is 64 characters. Any character that you can use in a directory name or file name, you can use in database and table namesexcept / and .. These limitations are in place because MySQL creates directories and files in your file system, which correspond to database and table names. There are no character limitations (besides length) in field names.

Q Can I use multiple functions in one statement, such as making a concatenated string all uppercase?

A Surejust be mindful of your opening and closing parentheses. This example shows how to uppercase the concatenated first and last names from the master name table:

```
mysql> SELECT UCASE(CONCAT_WS(' ', firstname,
lastname)) FROM table_name;
+-----------------------------------------+
| UCASE(CONCAT_WS(' ', firstname, lastname)) |
+-----------------------------------------+
| JOHN SMITH                              |
| JANE SMITH                              |
| JIMBO JONES                             |
| ANDY SMITH                              |
| CHRIS JONES                             |
| ANNA BELL                               |
| JIMMY CARR                              |
| ALBERT SMITH                            |
| JOHN DOE                                |
+-----------------------------------------+
9 rows in set (0.00 sec)
```

If you want to uppercase just the last name, use

```
mysql> SELECT CONCAT_WS(' ', firstname,
UCASE(lastname)) FROM master_name;
+-----------------------------------------+
| CONCAT_WS(' ', firstname, UCASE(lastname)) |
+-----------------------------------------+
| John SMITH                              |
| Jane SMITH                              |
| Jimbo JONES                             |
| Andy SMITH                              |
| Chris JONES                             |
| Anna BELL                               |
| Jimmy CARR                              |
| Albert SMITH                            |
| John DOE                                |
+-----------------------------------------+
9 rows in set (0.00 sec)
```

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and begin learning how to put your knowledge into practice.

Quiz

1. The integer 56678685 could be which data type(s)?
2. How would you define a field that could only contain the following strings: apple, pear, banana, cherry?
3. What would be the `LIMIT` clauses for selecting the first 25 records of a table? Then the next 25?
4. How would you formulate a string comparison using `LIKE` to match first names of "John" or "Joseph"?
5. How would you explicitly refer to a field called `id` in a table called `table1`?
6. Write a SQL statement that joins two tables, orders, and `items_ordered`, with a primary key in each of `order_id`. From the `orders` table, select the following fields: `order_name` and `order_date`. From the `items_ordered` table, select the `item_description` field.
7. Write a SQL query to find the starting position of a substring `"grape"` in a string `"applepearbananagrape"`.
8. Write a query that selects the last five characters from the string `"applepearbananagrape"`.

Answers

1. `MEDIUMINT, INT, or BIGINT.`
2. `ENUM ('apple', 'pear', 'banana', 'cherry') or SET ('apple', 'pear', 'banana', 'cherry')`
3. `LIMIT 0, 25 and LIMIT 25, 25`
4. `LIKE 'Jo%'`
5. Use `table1.id` instead of `id` in your query.
6.
```
SELECT orders.order_name, orders.order_date,
items_ordered.item_description FROM orders LEFT JOIN
items_ordered
ON orders.order_id = items_ordered.id;
```

7. `SELECT LOCATE('grape', 'applepearbananagrape');`
8. `SELECT RIGHT("applepearbananagrape", 5);`

Activity

Take the time to create some sample tables, and practice using basic `INSERT` and `SELECT` commands.