Creating Tables...contd.

Storing Data

Now that you have a basic understanding of databases, SQL, and MySQL, this topic begins the process of taking that knowledge deeper. The focus in this chapter, as the title states, is real-world database design. Like the work we did last week, much of the effort this week requires paper and pen—and serious thinking about what your applications will need to do.

Data stored in a relational database is organized into **tables**. A database table organizes data in a grid-like fashion, where each entry forms a row and each column identifies a specific value in the entry. To illustrate this, here's a table showing the number of medals won by each of the top five medal-winning countries that participated in the 2014 Winter Olympic Games. Each row lists the country's name, how many gold medals, silver medals, and bronze medals were won, and the total number of medals won.

Country	Gold	Silver	Bronze	Total
Russia	13	11	9	33
United States	9	7	12	28
Norway	11	5	10	26
Canada	10	10	5	25
Netherlands	8	7	9	24

A table like the one above is "physical" in that we can see it printed in a book or drawn on a whiteboard. It's limited only by the amount of physical space available. On the other hand, a database table is an intangible structure stored somewhere on a hard drive or in computer memory. We can only imagine it or make drawings to represent it. A database table is interpreted by a computer process (such as MySQL), and the limitations of the interpreting process impose restrictions on the table. The number of columns, the number of rows, and even what the individual values in a row can be, all depend upon what the computer system and database server can handle. But despite these limitations, a database table is very flexible. We can define relationships between tables, combine multiple tables together, sort rows and view specific entries, remove rows, and easily perform various calculations on the data.

We'll look at the **CREATE TABLE** statement—which defines new database tables—and discuss some important details surrounding table creation: MySQL's supported data types, naming restrictions, and storage engines. We'll also see how to add rows to a table with the INSERT statement, and finish by discussing transactions.

Creating Tables

Tables are created using the CREATE TABLE statement. In its simplest form, the statement provides the name of the table we we want to create and a list of column names and their data types. Not surprisingly, a CREATE TABLE statement can be very very complex depending on the requirements driving the design of the table. We can specify one or more attributes as part of a column's definition; such attributes can limit the range of values the column can store or specify a default value when one isn't provided by the user. Defining any logical relationships that exist between the table and another, and which storage engine MySQL should use to manage the table, is also common. You can see how detailed the statement can be if you look at the syntax and options for CREATE TABLE in the **MySQL documentation. (http://dev.mysql.com/doc/refman/5.6/en/create-table.html)**

Let's look at a pair of relatively simple CREATE TABLE statements. (I'll highlight some common points that add complexity, but I won't get too crazy, I promise.) With the "company" database created in the previous page as your active database, issue the statements below. MySQL should respond "Query OK" after each one.

```
CREATE TABLE employees (
employee_id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
last_name VARCHAR(30) NOT NULL,
first_name VARCHAR(30) NOT NULL,
email VARCHAR(100) NOT NULL,
hire_date DATE NOT NULL,
notes MEDIUMTEXT,

PRIMARY KEY (employee_id),
INDEX (last_name),
UNIQUE (email) );
```

```
CREATE TABLE address (
employee_id INTEGER UNSIGNED NOT NULL, address VARCHAR(50) NOT NULL,
city VARCHAR(30) NOT NULL,
state CHAR(2) NOT NULL,
postcode CHAR(5) NOT NULL,
FOREIGN KEY (employee_id)

REFERENCES employee (employee_id) );
```

The first statement creates a table named employee, designed to store basic information about a company's employees—their name, email address, date of hire, and perhaps any notes the Human Resources director might provide. The formatting is just to keep things readable for ourselves; it makes no difference to MySQL whether we write a statement entirely on one line or across several lines with indentation. The spacing in a statement is also generally irrelevant.

LOCAL BIAS:

The address table has a North American bias. An address in the United States or Mexico fits perfectly, and a Canadian address can store the two-letter province or territory abbreviation in the state column. But an address in the Netherlands, for example, needs space for a 6-character postal code. Feel free to adapt the definition to your own locale.

Names chosen for a table and its columns can be anything we like so long as they adhere to the following restrictions:

- The name uses basic Latin letters (A–Z, both uppercase and lowercase), the dollar sign ($), underscore (_), or Unicode characters U+0080–U+FFFF.
- The null character 0x00, Unicode characters U+10000 and higher, and characters that are prohibited in file names like slash (/), backslash (\), and period are not allowed in a name.
- The name must be quoted if it contains characters outside of the above. MySQL uses back ticks by default for this (`…`) although it can be configured to use single quotes ('…') as well. I recommend sticking with the default.
- The name must be quoted if it's a MySQL reserved keyword. A list of reserved words can be found in the online documentation (https://dev.mysql.com/doc/refman/5.7/en/keywords.html).

The employee_id column is designated as the table's primary key. A **primary key** is a column in which all the values are distinct and can be used to uniquely identify each row in the table. In more complex table definitions, we may define a primary key from multiple columns together but using a single INTEGER type column is the most common practice. Only one primary key can be defined per table (hence the name primary key).

The employee_id column also has the **AUTO_INCREMENT** attribute. Whenever we add a row that doesn't provide a value for this column, MySQL will automatically use the next highest sequential integer as its value. Suppose we have several rows in the employee table and the largest employee_id value among them is 42. If we add a new row without an employee_id value, MySQL will use 43 for the missing value. If we then add another row without the value, MySQL will use 44, and so on. Only one column in the table can be designated an auto-increment column, and the column must also be a primary key.

Behind the scenes, MySQL maintains various data structures to track data and relationships. The INDEX defined on last_name lets MySQL know that we might use its value in our selection criteria later when we retrieve rows—for example, if we wanted to search for employees named Smith or Jones. MySQL will create and manage a special index structure with the values in the column to make its search more efficient. Don't go overboard adding indexes though. It takes time for MySQL to maintain them so row retrieval may be faster, but adding/updating rows will be slower.

The term **constraint** describes a special condition imposed on a column or table that must always be adhered to. Most of the column definitions have NOT NULL, a constraint that prohibits storing NULL values in the column. NULL is a special value that represents the absence of a value. Essentially, NOT NULL means the column must hold a value. MySQL treats NULL differently from an empty value, such as an empty text string.

The UNIQUE constraint defined on the email column ensures all the email addresses stored in the table are different. UNIQUE and PRIMARY KEY are similar, but there are important differences between them. Because the values in a primary key column must be able to unambiguously identify each row, its uniqueness is inherent. We don't explicitly specify UNIQUE with PRIMARY KEY. And while only one primary key can be defined per table, we can provide any number of UNIQUE constraints. A UNIQUE column may also contain NULL values, something PRIMARY KEY doesn't allow.

The **FOREIGN KEY** constraint in the address table's CREATE TABLE statement references the employee table, thus defining a relationship between the two tables. This relationship means that a row in the address table is logically related to whatever row in the employee table that has the same value in its employee_id column. Take, for instance, a row in the address table with an employee_id value of 42. That row may be associated with the row in the employee table whose employee_id value is also 42. In other words, an address with employee_id 42 is linked to employee 42's employee record. A FOREIGN KEY column doesn't need to have the same name as its partner column in the other table, but the two must share the same data type and NULL constraint.

We can issue DESCRIBE or SHOW CREATE TABLE statements to verify a table was created or view the definition of an existing table. The DESCRIBE statement returns the list of the table's column names and their data types, and SHOW CREATE TABLE returns a statement that can be used later to re-create the table.

```
DESCRIBE employee;
```

```
SHOW CREATE TABLE employee;
```

Pick A Convention: A convention I've adopted is to type MySQL keywords in uppercase and my own identifiers in lowercase. MySQL doesn't treat keywords and column names in a case-sensitive manner, but table names might be case-sensitive depending on the file system storing your tables' files. It's best to pick a convention—whatever it may be—and stick with it.

So far, we've discussed the column attributes and table constraints that appear in the example, but we haven't discussed the **data types**. The next part may be a little dry, but it covers some important information. Each type requires a different amount of storage on disk and in memory, so we always want to specify the minimum viable type for a column. The amount of wasted space from assigning a data type that's larger than necessary might be negligible at first because there's only a handful of rows, but it can add up quickly as more and more data is added to the table.