Introduction to Databases and SQL

Files vs Databases

In the last chapter you learned how your PHP scripts can use external files to store and retrieve data.

Although files do a great job in many circumstances, they're pretty inflexible as data storage solution.

For example, if you need to filter or sort the data retrieved from a file, you have to write your own PHP code to do it.

Not only is this tiresome, but if you're working with large sets of data — for example, hundreds of thousands of user records — your script will probably grind to a halt. Not good if you're hoping to build a popular Web site.

Databases

Often, the most efficient alternative to text files is to use a database engine — commonly known as a **Database Management System** (DBMS) — to store, retrieve, and modify the data for you.

A good database engine serves as a smart tool between you and your data, organizing and cataloging the data for quick and easy retrieval.

You can use any database systems in your PHP applications. In this course, you'll focus on just one database engine: MySQL.

Also, the database **model** dictates how the data is stored and accessed. Many different database models are used today but here, we will focus on **relational** model, which is the model MYSQL uses.

Relational Databases

In simple terms, a relational database is any database system that allows data to be associated and grouped by common attributes. For example, a bunch of payroll records might be grouped by employee, by department, or by date.

Typically, a relational database arranges data into tables, where each table is divided into rows and columns of data.

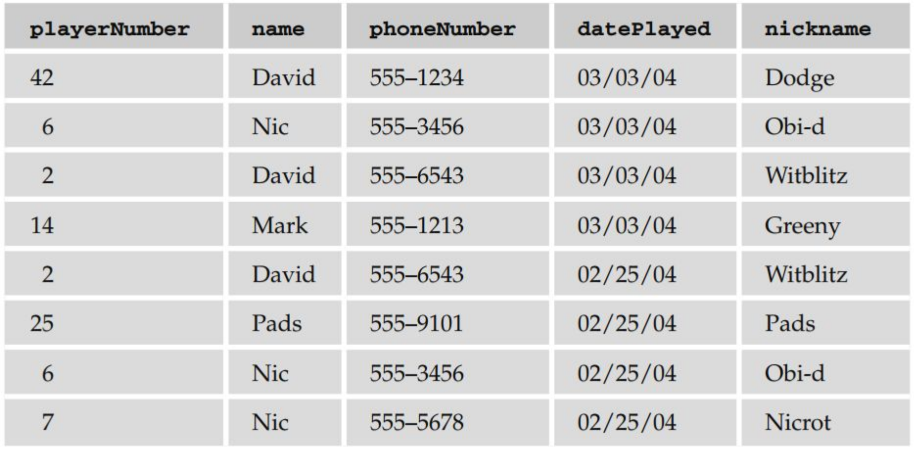
In database language, each **row** in a table represents a **data record** : a set of connected pieces of data, such as information relating to a particular person.

Likewise, each **column** represents a **field** : a specific type of data that has the same significance for each record in the table, such as " first name " or " age. "

Relational Database Example

Here's an example of a database table. Suppose that the manager of a football team sets up a database so that she can track the matches in which her players compete.

She asks each player to enter his details into the database after each match. After two matches the manager's table, called matchLog , looks like this:



playerNumber	name	phoneNumber	datePlayed	nickname
42	David	555–1234	03/03/04	Dodge
6	Nic	555–3456	03/03/04	Obi-d
2	David	555–6543	03/03/04	Witblitz
14	Mark	555–1213	03/03/04	Greeny
2	David	555–6543	02/25/04	Witblitz
25	Pads	555–9101	02/25/04	Pads
6	Nic	555–3456	02/25/04	Obi-d
7	Nic	555–5678	02/25/04	Nicrot

The Example

In this table, you can see that each row represents a particular set of information about a player who played on a certain date, and each column contains a specific type of data for each person or date.

Notice that each column has a name at the top of the table to identify it; this is known as the **field name** or **column name**

Normalization

The manager soon realizes that this matchLog table is going to be huge after everyone on the team has played an entire season's worth of games.

As you can see, the structure of the table is inefficient because each player's details (number, name, phone number, and so on) are entered every time he plays a match.

Such redundancy is undesirable in a database. For example, say that the player with the number 6 keeps dropping the ball, and his teammates decide to give him a new nickname (which won't be mentioned here). To update the table, every one of this player's records would have to be modified to reflect his new nickname.

Normalization

By normalizing our data, we can ensure that our data tables are efficient and well-designed.

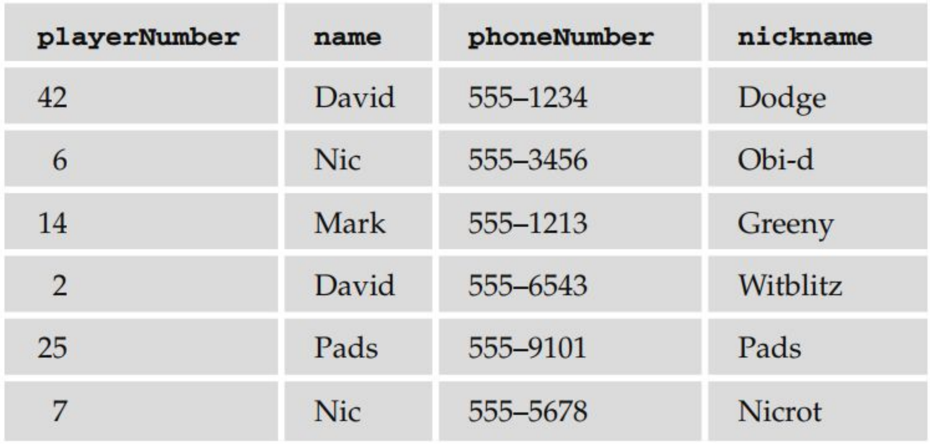
This chapter doesn't go into detail about normalization, which is quite a complex topic. However, the basic idea is to break up your data into several related tables, so as to minimize the number of times you have to repeat the same data.

The matchLog table contains a lot of repeating data. You can see that most of the repeating data is connected with individual players. For example, the player with the nickname “ Witblitz ” is mentioned twice in the table, and each time he ’ s mentioned, all of his information (his player number, name, and phone number) is also included.

Normalization

Therefore, it makes sense to pull the player details out into a separate **players** table, as follows:

You can see that each player has just one record in this table. The playerNumber field is the field that uniquely identifies each player (for example, there are two Davids, but they have different playerNumber fields). The playerNumber field is said to be the **table's primary key** .

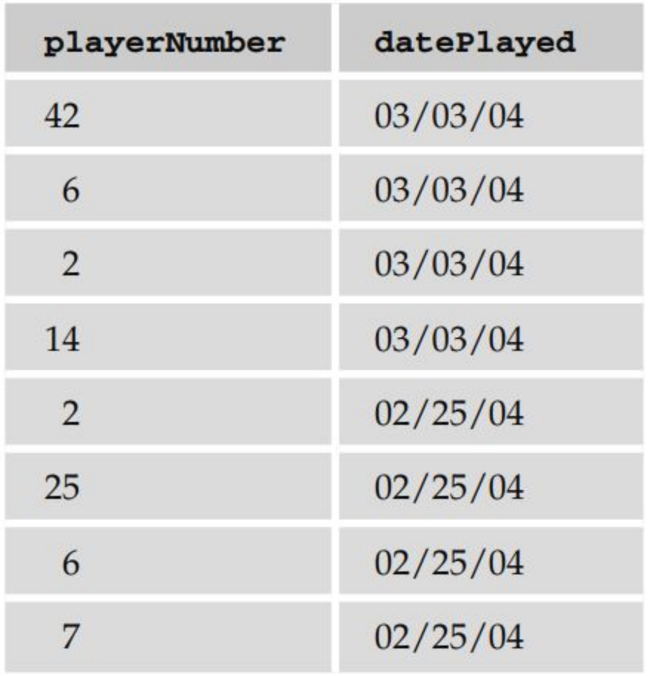


playerNumber	name	phoneNumber	nickname
42	David	555–1234	Dodge
6	Nic	555–3456	Obi-d
14	Mark	555–1213	Greeny
2	David	555–6543	Witblitz
25	Pads	555–9101	Pads
7	Nic	555–5678	Nicrot

Normalization

Now that the player fields have been pulled out into the players table, the original matchLog table contains just one field — datePlayed — representing the date that a particular player participated in a match.

Here comes the clever bit. We add the playerNumber column back into the matchLog table



playerNumber	datePlayed
42	03/03/04
6	03/03/04
2	03/03/04
14	03/03/04
2	02/25/04
25	02/25/04
6	02/25/04
7	02/25/04

Normalization

Now, by linking the values of the playerNumber fields in both the player and matchLog tables, you can associate each player with the date (or dates) he played.

The two tables are said to be **joined** by the playerNumber field. The playerNumber field in the matchLog table is known as a **foreign key** , because it references the **primary key** in the players table, and you can't have a playerNumber value in the matchLog table that isn't also in the players table.

Because the only repeating player information remaining in the matchLog table is the playerNumber field, you've saved some storage space when compared to the original table. Furthermore, it's now easy to change the nickname of a player, because you only have to change it in one place: a single row in the players table.

Talking to Databases with SQL

You're probably wondering how to actually retrieve information from these two tables, such as the nicknames of the players who played on March 3, 2004.

This is where SQL comes in.

SQL, the Structured Query Language, is a simple, standardized language for communicating with relational databases.

SQL lets you do practically any database-related task, including creating databases and tables, as well as saving, retrieving, deleting, and updating data in databases.

MySQL Data Types

When you create a database table — which you do later in this lecture — the type and size of each field must be defined.

A field is similar to a PHP variable except that you can store only the specified type and size of data in a given field.

For example, you can't insert characters into an integer field.

MySQL supports three main groups of data types

1. numeric
2. date/time
3. string

Numeric Data Types

You can store numbers in MySQL in many ways, but here we will only discuss two of them:

Numeric Data Type	Description	Allowed Range of Values
INT	Normal - sized integer	– 2147483648 to 2147483647, or 0 to 4294967295 if UNSIGNED
BIT	0 or 1	0 or 1

You can add the attribute UNSIGNED after a numeric data type when defining a field. An unsigned data type can only hold positive numbers.

Date and Time Data Types

As with numbers, you can choose from a range of different data types to store dates and times. Here we only discuss two:

Numeric Data Type	**Description**	**Allowed Range of Values**
DATE	Date	1 Jan 1000 to 31 Dec 9999
TIME	Time	– 838:59:59 to 838:59:59

String Data Types

MySQL lets you store strings of data in many different ways. Again we will only focus on two of those:

Numeric Data Type	Description	Allowed Range of Values
VARCHAR(n)	Variable-length string of up to n characters	0 – 65535 characters
TEXT	Normal-sized text field	0 – 65535 characters

SQL Statements

To actually work with databases and tables, you use SQL statements. Common statements include:

- SELECT — Retrieves data from one or more tables
- INSERT — Inserts data into a table
- UPDATE — Updates data in a table
- DELETE — Deletes data from a table
- CREATE — Creates a database, table or index
- ALTER — Modifies the structure of a table
- DROP — Wipes out a database or table

A Sample SQL Statement

```
SELECT lastName, firstName
FROM users
WHERE firstName = ‘John’;
```

Take a closer look at the FROM and WHERE clauses in the query. The query returns ***any*** record from the users table where the value of the firstName field is “ John".

Assuming there actually is a table called users in the database, the query's output which is called the ***result set*** might look like this:

```
Simpleton John
Smith John
Thomas John
```

Setting Up MySQL

The MySQL database system comes with a number of different programs. The two important ones that you learn about here are:

The MySQL server: This is the database engine itself. The program is usually called mysqld or similar

The MySQL command-line tool: You can use this tool to talk directly to the MySQL server so that you can create databases and tables, and add, view, and delete data. It's handy for setting up your databases and also for troubleshooting. The program name is simply mysql

Starting the MySQL Server

If you installed WampServer on Windows, or MAMP on Mac OS X, then the MySQL server and command-line tool should already be installed on your computer.

In fact, MySQL server may already be running, but if it's not, here's how to start it:

WampServer on Windows: Examine the WampServer icon in your taskbar. If the icon is black and white, your Apache and MySQL servers should be running correctly. If the icon is part yellow or part red, then one or both of the servers aren't running. Click the icon to display the WampServer menu, then choose the Start All Services or Restart All Services option to start both the Apache and MySQL servers

Starting the MySQL Server

MAMP on Mac OS X: Open the MAMP folder inside your Applications folder in Finder, then double-click the MAMP icon to launch the application. If the MySQL server has a red light to the left of it, click the Start Servers button to start up both the Apache and MySQL servers. Both lights should now be green

Creating a New Database

To create a new database, all you have to do is use the CREATE DATABASE command.

Type the following to create a new database called *mydatabase* :

```
CREATE DATABASE mydatabase;
```

Creating a New Table

As you know, tables are where you actually store your data.

To start with, you'll create a very simple table, *fruit* , containing three fields: id (the primary key), name (the name of the fruit), and color (the fruit's color).

```
CREATE TABLE fruit (
  id INT UNSIGNED AUTO_INCREMENT,
  name VARCHAR(30),
  color VARCHAR(30),
  PRIMARY KEY (id)
);
```

Creating a New Table

You've created a table with the following three fields:

id is the primary key. It uniquely identifies each row of the table. You created the id field as INT UNSIGNED , which means it can hold integer values up to 4,294,967,295. You also specified the keyword AUTO_INCREMENT . This ensures that, whenever a new row is added to the table, the id field automatically gets a new unique value (starting with 1). This means you don't have to specify this field's value when inserting data

name will store the name of each fruit. It's created as VARCHAR(30) , which means it can hold strings of up to 30 characters in length.

color was created in the same way as name , and will be used to store the color of each fruit.

Adding Data to a Table

To add a new row to a table, you use the SQL INSERT statement. In its basic form, an INSERT statement looks like this:

```
INSERT INTO table VALUES ( value1 , value2 , ... );
```

This inserts values into each of the fields of the table, in the order that the fields were created.

Alternatively, you can create a row with only some fields populated. The remaining fields will contain NULL (if allowed), or in the case of special fields such as an AUTO_INCREMENT field, the field value will be calculated automatically.

To insert a row of partial data, use:

```
INSERT INTO table ( field1 , field2 , ... ) VALUES ( value1, value2 , ... );
```

Adding Data to a Table

Now try adding some fruit to your table. you can add three rows to the *fruit* table by inserting data into just the *name* and *color* fields (the *id* field will be filled automatically):

```
INSERT INTO fruit ( name, color ) VALUES ( ‘banana’, ‘yellow’ );

INSERT INTO fruit ( name, color ) VALUES ( ‘tangerine’, ‘orange’ );

INSERT INTO fruit ( name, color ) VALUES ( ‘plum’, ‘purple’ );
```

Reading Data From a Table

To read data in SQL, you create a query using the SELECT statement.

name of the table you want to read data from

```
SELECT field1, field2, ... FROM table WHERE condition;
```

name of the fields you want to read from the table e.g age

You can filter the results using the condition. e.g. age = 15

Reading Data From a Table

If you want to read all fields from a table, you can use * instead of including all field names.

```
SELECT * FROM fruit;
```

This SQL statement returns all records in the fruit table.

id	name	color
1	banana	yellow
2	tangerine	orange
3	plum	purple

Reading Data From a Table

Or we may use the following statement if we only want to see the name and color of all fruits.

```
SELECT name, color FROM fruit;
```

name	**color**
banana	yellow
tangerine	orange
plum	purple

Reading Data From a Table

Now if we want to retrieve a specific set of records with a given condition we can add WHERE conditions to our query:

```
SELECT * FROM fruit where name = 'banana';
```

This query will return all records in the fruit table with the name 'banana'

id	name	color
1	banana	yellow

Reading Data From a Table

Similarly we can specify conditions on other fields:

```
SELECT * FROM fruit where id >= 2;
```

This query will return all records in the fruit table where id >= 2

id	name	color
2	tangerine	orange
3	plum	purple

Reading Data From a Table

We can also combine multiple conditions by using AND/OR operators:

```
SELECT * FROM fruit where id >= 2 AND name = 'plum';
```

Here we will get a record with id >= 2 and the name = 'plum'

id	**name**	**color**
3	plum	purple

Updating Data in a Table

You change existing data in a table with the UPDATE statement.

As with the SELECT statement, you can (and usually will) add a WHERE clause to specify exactly which rows you want to update.

If you leave out the WHERE clause, the entire table gets updated.

```
UPDATE table SET field1 = value1, field2 = value2, ... WHERE condition;
```

name of the table you are updating

pairs of field name and values e.g. age =15 will set the age to 15 for all records that the condition is true

You can limit the set of records that you want to update by using a condition. e.g. name = 'sally'

Updating Data in a Table

Here's how to use UPDATE to change values in your fruit table:

```
UPDATE fruit SET name = 'grapefruit', color = 'yellow' WHERE id = 2;
```

This SQL statement changes the name and color field values for the record with id = 2. If we read data in the table after this statement, the table looks like this:

```
SELECT * FROM fruit;
```

id	**name**	**color**
1	banana	yellow
2	grapefruit	yellow
3	plum	purple

Deleting Data from a Table

Deleting works in a similar way to updating.

To delete rows, you use the DELETE statement. If you add a WHERE clause, you can choose which row or rows to delete; otherwise all the data in the table are deleted (though the table itself remains)

```
DELETE FROM table WHERE condition;
```

name of the table you want to delete data from

You can limit the set of records that you want to delete by using a condition. e.g. name = 'sally'

Deleting Data from a Table

Let's delete all records with id = 2 in our fruit table:

```
DELETE FROM fruit where id = 2;
```

This will delete one record from our fruit table. We can use SELECT to read data in our table :

```
SELECT * FROM fruit;
```

id	**name**	**color**
1	banana	yellow
3	plum	purple

Deleting Tables

To delete a table entirely, use the DROP TABLE statement

```
DROP TABLE table;
```

This is the name of the table you want to delete entirely

Note that this will delete the table and all records in it. This operation is not reversible.

Deleting Databases

To delete a database entirely, use the DROP DATABASE statement

```
DROP DATABASE db;
```

This is the name of the database you want to delete entirely

Note that this will delete the database and all tables and all records in it. This operation is not reversible.