Discover five common database problems that occur in PHP applications -- including database schema design, database access, and the business logic code that uses the database -- as well as their solutions.

If only there were one way to use databases correctly ...

You can create database design, database access, and the PHP business logic code that sits on top of it in any number of ways, and you often end up getting it wrong. This article illustrates five common problems in database design, in the PHP code that accesses databases, and how to fix these problems when you see them.

**Problem 1: Using MySQL directly**

One common problem is older PHP code using the `mysql_` functions to access the database directly. Listing 1 shows how to access the database directly.

**Listing 1. Access/get.php**

```php
<?php
function get_user_id( $name ) {
    $db = mysql_connect( 'localhost', 'root', 'password' );
    mysql_select_db( 'users' );
    $res = mysql_query( "SELECT id FROM users WHERE login='$\name'" );
    while( $row = mysql_fetch_array( $res ) ) { $id = $row[0]; }
    return $id;
}
var_dump( get_user_id( 'jack' ) );
?>
```

Notice the use of the `mysql_connect` function to access the database. Also notice the query in which we use string concatenation to add the `$name` parameter to the query.
This technique has two good alternatives: the PEAR DB module and the PHP Data Objects (PDO) classes. Both provide abstraction from the choice of a particular database. Therefore, your code can run without too much adjustment on IBM® DB2®, MySQL, PostgreSQL, or any other database you want to connect to.

The other value in using the abstraction layers of the PEAR DB module and PDO is that you can use the `?` operator in your SQL statements. Doing so makes the SQL easier to maintain and secures your application from SQL injection attacks.

The alternative code using the PEAR DB module is shown below.

**Listing 2. Access/get_good.php**

```php
<?php
require_once("DB.php");

function get_user_id( $name )
{
    $dsn = 'mysql://root:password@localhost/users';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); } 

    $res = $db->query( 'SELECT id FROM users WHERE login=?',
    array( $name ) );
    $id = null;
    while( $res->fetchInto( $row ) ) { $id = $row[0]; }
    return $id;
}

var_dump( get_user_id( 'jack' ) );
?>
```

Notice that all direct mentions of MySQL are gone except for the database connection string in `$dsn`. In addition, we use the `$name` variable in the SQL through the `?` operator. Then, the data for the query is sent in through `array` at the end of the `query()` method.

**Problem 2: Not using auto-increment functionality**

Like most modern databases, MySQL has the ability to create auto-incrementing unique identifiers on a per-record basis. Despite that, we still see code that first runs a `SELECT` statement to find the maximum `id`, then adds one to that `id`, as well as a new record. Listing 3 shows a sample bad schema.

**Listing 3. Badid.sql**

```sql
DROP TABLE IF EXISTS users;
CREATE TABLE users ( 
    id MEDIUMINT,
    login TEXT,
    password TEXT
);

INSERT INTO users VALUES ( 1, 'jack', 'pass' );
INSERT INTO users VALUES ( 2, 'joan', 'pass' );
INSERT INTO users VALUES ( 1, 'jane', 'pass' );
```

Notice that all direct mentions of MySQL are gone except for the database connection string in `$dsn`. In addition, we use the `$name` variable in the SQL through the `?` operator. Then, the data for the query is sent in through `array` at the end of the `query()` method.
The \texttt{id} field here is specified simply as an integer. So, although it should be unique, we can add any value we like, as shown in the \texttt{INSERT} statements that follow the \texttt{CREATE} statement. Listing 4 shows the PHP code that adds users into this type of schema.

\textbf{Listing 4. Add\_user.php}

```php
<?php
require_once("DB.php");

function add_user( $name, $pass )
{
    $rows = array();
    $dsn = 'mysql://root:password@localhost/bad\_badid';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }
    $res = $db->query( "SELECT max(id) FROM users" );
    $id = null;
    while( $res->fetchInto( $row ) ) { $id = $row[0]; }
    $id += 1;
    $sth = $db->prepare( "INSERT INTO users VALUES(? ,?,?)" );
    $db->execute( $sth, array( $id, $name, $pass ) );
    return $id;
}
$id = add_user( 'jerry', 'pass' );
var\_dump( $id );
?>
```

The code in \texttt{add\_user.php} first performs a query to find the maximum value of the \texttt{id}. Then the file runs an \texttt{INSERT} statement with the \texttt{id} value plus one. This code could fail in race conditions on servers with a heavy load. Plus, it's just inefficient.

So what's the alternative? Use the auto-increment feature in MySQL to create unique IDs for each insertion automatically. The updated schema is shown below.

\textbf{Listing 5. Goodid.php}

```sql
DROP TABLE IF EXISTS users;

CREATE TABLE users (  
    id MEDIUMINT NOT NULL AUTO\_INCREMENT,  
    login TEXT NOT NULL,  
    password TEXT NOT NULL,  
    PRIMARY KEY( id )  
);  

INSERT INTO users VALUES( null, 'jack', 'pass' );  
INSERT INTO users VALUES( null, 'joan', 'pass' );  
INSERT INTO users VALUES( null, 'jane', 'pass' );
```

We added the \texttt{NOT NULL} flag to indicate that the fields must not be null. We also added the \texttt{AUTO\_INCREMENT} flag to indicate that the field is auto-incrementing, as well as the \texttt{PRIMARY KEY} flag to indicate which field is an \texttt{id}. These changes speed things up a bit. Listing 6 shows the updated PHP code that inserts users into the table.
Listing 6. Add_user_good.php

```php
<?php
require_once("DB.php");

function add_user( $name, $pass )
{
    $dsn = 'mysql://root:password@localhost/good_genid';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }

    $sth = $db->prepare( "INSERT INTO users VALUES(null,?,?)" );
    $db->execute( $sth, array( $name, $pass ) );

    $res = $db->query( "SELECT last_insert_id()" );
    $id = null;
    while( $res->fetchInto( $row ) ) { $id = $row[0]; }
    return $id;
}
$id = add_user( 'jerry', 'pass' );
var_dump( $id );
?>
```

Instead of getting the maximum id value, I now just use the `INSERT` statement to insert the data, then use a `SELECT` statement to retrieve the id of the last inserted record. This code is a lot simpler and more efficient than the original version and its related schema.

An alternative to using MySQL's auto-increment functionality is to use the `nextId()` method in the PEAR DB system. In the case of MySQL, this creates a new sequence table and manages that using an elaborate locking mechanism. The advantage of using this method is that it will work across different database systems.

Either way, you should use a system that manages incrementing unique IDs for you and not rely on a system where you query first, then increment the value yourself, and add the record. The latter approach is susceptible to race conditions on high-volume sites.

Problem 3: Using multiple databases

Once in a while, we see an application in which each table is in a separate database. There are reasons for doing that in extraordinarily large databases, but for an average application, you don't need this level of segmentation. In addition, even though it's possible to perform relation queries across multiple databases, I strongly recommend against it. The syntax is more complex. Backup and restore is not easily managed. The syntax may or may not work between different database engines. And it's difficult to follow the relational structure when the tables are split over multiple databases.

So, what would multiple databases look like? To begin, you need some data. Listing 7 shows this data divided into four files.

Listing 7. The database files

```sql
Files.sql:
CREATE TABLE files (  
```
id MEDIUMINT,
user_id MEDIUMINT,
name TEXT,
path TEXT
);

Load_files.sql:
INSERT INTO files VALUES ( 1, 1, 'test1.jpg', 'files/test1.jpg' );
INSERT INTO files VALUES ( 2, 1, 'test2.jpg', 'files/test2.jpg' );

Users.sql:
DROP TABLE IF EXISTS users;
CREATE TABLE users (id MEDIUMINT,
login TEXT,
password TEXT
);

Load_users.sql:
INSERT INTO users VALUES ( 1, 'jack', 'pass' );
INSERT INTO users VALUES ( 2, 'jon', 'pass' );

In the multiple-database version of these files, you would load SQL statements into one database, then load the users SQL statements into another database. The PHP code to query the database for the files associated with a particular user is shown below.

Listing 8. Getfiles.php

<?php
require_once("DB.php");

function get_user( $name )
{
    $dsn = 'mysql://root:password@localhost/bad_multi1';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }
    $res = $db->query( "SELECT id FROM users WHERE login=?", array( $name ) );
    $uid = null;
    while( $res->fetchInto( $row ) ) { $uid = $row[0]; }
    return $uid;
}

function get_files( $name )
{
    $uid = get_user( $name );
    $rows = array();
    $dsn = 'mysql://root:password@localhost/bad_multi2';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }
    $res = $db->query( "SELECT * FROM files WHERE user_id=?", array( $uid ) );
    while( $res->fetchInto( $row ) ) { $rows[] = $row; }
    return $rows;
}

$files = get_files( 'jack' );
var_dump( $files );
The `get_user` function connects to the database that contains the users table and retrieves the ID for a given user. The `get_files` function connects to the files table and retrieves the file rows associated with the given user.

A better way to do all this is to load the data into one database, then perform a query, such as that shown below.

**Listing 9. Getfiles_good.php**

```php
<?php
require_once("DB.php");

function get_files( $name )
{
    $rows = array();

    $dsn = 'mysql://root:password@localhost/good_multi';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); };

    $res = $db->query("SELECT files.* FROM users, files WHERE
        users.login=? AND users.id=files.user_id",
        array( $name ) );
    while( $res->fetchInto( $row ) ) { $rows[] = $row; }

    return $rows;
}

$files = get_files( 'jack' );

var_dump( $files );
?>
```

This code is not only shorter but it's also easier to understand and more efficient. Instead of performing two queries, we're performing one.

While this problem sounds a bit far-fetched, we've seen it in practice enough times to know that all tables should be in the same database unless there's a pressing reason otherwise.

**Problem 4: Not using relations**

Relational databases aren't like programming languages. They don't have array types. Instead, they use relations among tables to create a one-to-many structure between objects, which has the same effect as an array. One problem I've seen with applications is when engineers attempt to use a database as though it were a programming language, creating arrays by using text strings with comma-separated identifiers. Look at the schema below.
Listing 10. Bad.sql

DROP TABLE IF EXISTS files;
CREATE TABLE files (  
id MEDIUMINT,  
  name TEXT,  
  path TEXT
);

DROP TABLE IF EXISTS users;
CREATE TABLE users (  
id MEDIUMINT,  
  login TEXT,  
  password TEXT,  
  files TEXT
);

INSERT INTO files VALUES ( 1, 'test1.jpg', 'media/test1.jpg' );
INSERT INTO files VALUES ( 2, 'test1.jpg', 'media/test1.jpg' );
INSERT INTO users VALUES ( 1, 'jack', 'pass', '1,2' );

One user in the system can have multiple files. In a programming language, you would use an array to represent the files associated with a user. In this example, the programmer chose to create a **files** field containing a list of file ids separated by commas. To get a list of all the files for a particular user, the programmer must first read the row from the users table, then parse the file's text and run an individual SELECT statement for each file. This code is shown below.

Listing 11. Get.php

```php
<?php
require_once("DB.php");

function get_files( $name )  
{  
    $dsn = 'mysql://root:password@localhost/bad_norel';
    $db = DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }  

    $res = $db->query( "SELECT files FROM users WHERE login=?",  
                        array( $name ) );
    $files = null;
    while( $res->fetchInto( $row ) ) { $files = $row[0]; }  

    $rows = array();

    foreach( split( ',',$files ) as $file )  
    {  
        $res = $db->query( "SELECT * FROM files WHERE id=?",  
                           array( $file ) );
        while( $res->fetchInto( $row ) ) { $rows[] = $row; }
    }

    return $rows;
}

.files = get_files( 'jack' );

var_dump( $files );
?>
```
This technique is slow, difficult to maintain, and doesn't make good use of the database. The only solution is to re-architect the schema to turn it back into a traditional relational form, as shown below.

### Listing 12. Good.sql

```sql
DROP TABLE IF EXISTS files;
CREATE TABLE files (  
id MEDIUMINT,  
user_id MEDIUMINT,  
name TEXT,  
path TEXT
);

DROP TABLE IF EXISTS users;
CREATE TABLE users (  
id MEDIUMINT,  
login TEXT,  
password TEXT
);

INSERT INTO users VALUES ( 1, 'jack', 'pass' );
INSERT INTO files VALUES ( 1, 1, 'test1.jpg', 'media/test1.jpg' );
INSERT INTO files VALUES ( 2, 1, 'test1.jpg', 'media/test1.jpg' );
```

Here, each file is related to the user through the `user_id` function in the file table. This probably seems backwards to anyone looking at this as an array. Certainly, arrays don’t reference their containing objects -- in fact, just the opposite. But in a relational database, this is how things work and why queries are so much faster and easier. Listing 13 shows the corresponding PHP code.

### Listing 13. Get_good.php

```php
<?php
require_once("DB.php");

function get_files( $name )
{
    $dsn = 'mysql://root:password@localhost/good_rel';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }

    $rows = array();
    $res = $db->query(  
        "SELECT files.* FROM users,files WHERE users.login=?  
        AND users.id=files.user_id",
        array( $name ) );
    while( $res->fetchInto( $row ) ) { $rows[] = $row; }
    return $rows;
}

$files = get_files( 'jack' );
var_dump( $files );
?>
```

Here, we make one query to the database to get all the rows. The code isn’t complex, and it uses the database as it was intended.
Problem 5: The n+1 pattern

I can't tell you how many times we've seen large applications in which the code first retrieves a list of entities -- say, customers -- then comes back and retrieves them one by one to get the details for each entity. We call it the \( n+1 \) pattern because that's how many queries will be performed -- one query to retrieve the list of all the entities, then one query for each of the \( n \) entities. This isn't a problem when \( n=10 \), but what about when \( n=100 \) or \( n=1000 \)? Then the inefficiency really kicks in. Listing 14 shows an example of such a schema.

Listing 14. Schema.sql

```
DROP TABLE IF EXISTS authors;
CREATE TABLE authors (  
id MEDIUMINT NOT NULL AUTO_INCREMENT,  
name TEXT NOT NULL,  
PRIMARY KEY ( id )  
);

DROP TABLE IF EXISTS books;
CREATE TABLE books (  
id MEDIUMINT NOT NULL AUTO_INCREMENT,  
author_id MEDIUMINT NOT NULL,  
name TEXT NOT NULL,  
PRIMARY KEY ( id )  
);

INSERT INTO authors VALUES ( null, 'Jack Herrington' );
INSERT INTO authors VALUES ( null, 'Dave Thomas' );

INSERT INTO books VALUES ( null, 1, 'Code Generation in Action' );
INSERT INTO books VALUES ( null, 1, 'Podcasting Hacks' );
INSERT INTO books VALUES ( null, 1, 'PHP Hacks' );
INSERT INTO books VALUES ( null, 2, 'Pragmatic Programmer' );
INSERT INTO books VALUES ( null, 2, 'Ruby on Rails' );
INSERT INTO books VALUES ( null, 2, 'Programming Ruby' );
```

This schema is solid. There's nothing wrong here. The problem is in the code that accesses the database to find all the books for a given author, as shown below.

Listing 15. Get.php

```
<?php
require_once('DB.php');

$dsn = 'mysql://root:password@localhost/good_books';
$db =& DB::Connect( $dsn, array() );
if (PEAR::isError($db)) { die($db->getMessage()); }

function get_author_id( $name )
{
    global $db;

    $res = $db->query( "SELECT id FROM authors WHERE name=?",  
                        array( $name ) );
    $id = null;
    while( $res->fetchInto( $row ) ) { $id = $row[0]; }
    return $id;
}

function get_books( $id )
{
    global $db;
```
Five common PHP database problems

If you look at the code at the bottom, you are likely to think to yourself, "Hey, this is really clean." First, get the author id, then get a list of the books, then get information about each book. Sure, it's clean -- but is it efficient? No. Look at how many queries we had to perform to retrieve only the books by Jack Herrington. One to get an id, another to get the list of books, then one for each book. Five queries for three books!

The solution is to have one function that performs one bulk query, as shown below.

**Listing 16. Get_good.php**

```php
<?php
require_once('DB.php');
$dsn = 'mysql://root:password@localhost/good_books';
$db = DB::Connect($dsn, array());
if (PEAR::isError($db)) { die($db->getMessage()); }

function get_books($name)
{
    global $db;
    
    $res = $db->query("SELECT * FROM books WHERE authors.name=?", array($name));
    $rows = array();
    while($res->fetchInto($row)) { $rows[] = $row; }
    return $rows;
}

$books = get_books( 'Jack Herrington' );
var_dump($books);
?>
```

Now retrieving the list requires a fast, single query. It means that I will likely have to have several of these types of methods with different parameters, but there really is no choice. If you want to
have a PHP application that scales, you must make efficient use of the database, and that means smarter queries.

The problem with this example is that it's a bit too clear-cut. Typically, these types of $n+1$ or $n^2$ problems are much more subtle. And they only appear when the database administrator runs a query profiler on your system when it has performance problems.

**Conclusion**

Databases are powerful tools, and -- like all powerful tools -- they can be abused if you don't know how to use them properly. The trick behind identifying and solving these problems is to better understand the underlying technology. For too long, I've heard business logic coders lament that they don't want to have to understand the database or the SQL code. They wrap the database in objects and wonder why the performance is so poor.

They fail to realize that understanding the SQL is fundamental to turning the database from a difficult necessity into a powerful ally. If you use databases on a daily basis, but SQL isn't your strong suit, read *The Art of SQL*. It's a well-written, practical guide to getting the most out of a database.
Related topics

- *The Art of SQL*, by Stephane Faroult and Peter Robson, is a must-read book for programmers who use databases in their applications.
- **PHP.net** is the starting point for all things PHP.
- The **PEAR DB documentation** is an excellent resource.
- The **PDO Functions documentation** can bring you up to speed on PHP Data Objects (PDO).
- **MySQL.org** has excellent documentation with examples that show how to make better use of a database.
- Visit IBM developerWorks' **PHP project resources** to learn more about PHP.
- Innovate your next open source development project with **IBM trial software**, available for download or on DVD.

© Copyright IBM Corporation 2006

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)