DB2

IBM

DB2 Version 9
for Linux, UNIX, and Windows

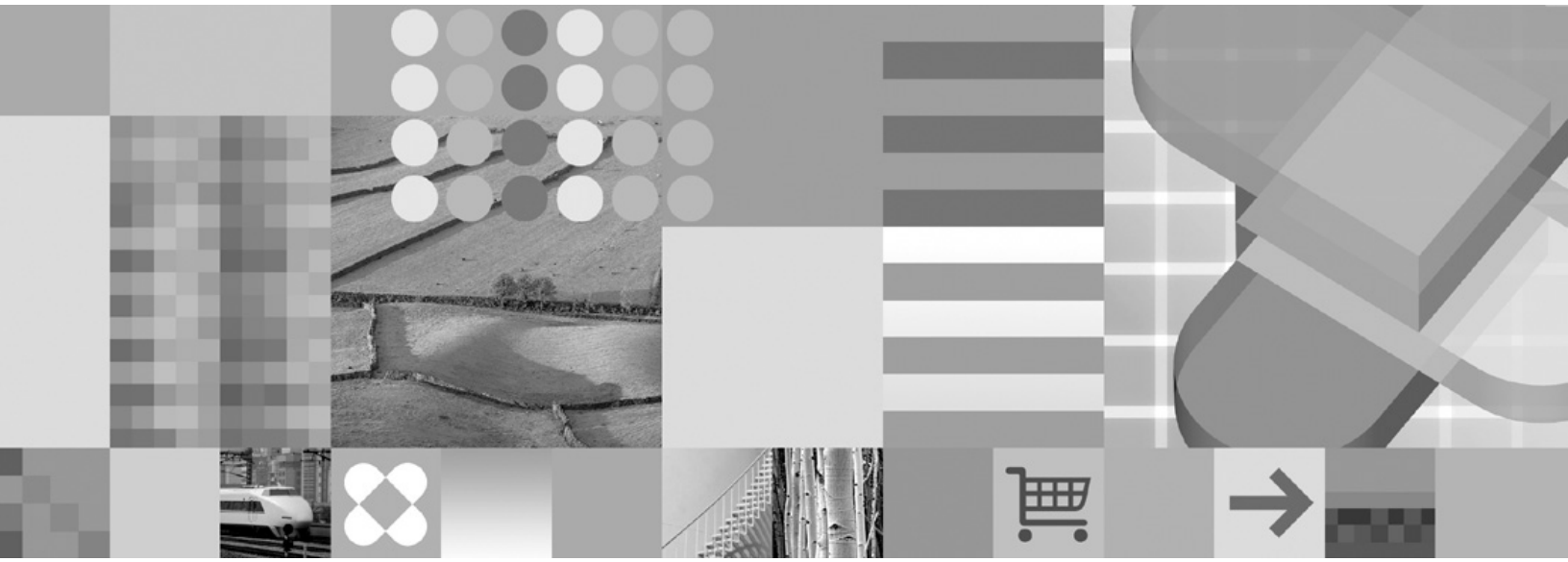
Developing Perl and PHP Applications

SC10-4234-00

DB2®

IBM

DB2 Version 9
for Linux, UNIX, and Windows



Developing Perl and PHP Applications

SC10-4234-00

Before using this information and the product it supports, be sure to read the general information under *Notices*.

Contents

Part 1. Developing PHP Applications

Chapter 1. Introduction to Developing PHP Applications

Introduction to PHP application development for DB2

PHP: Hypertext Preprocessor (PHP) is an interpreted programming language primarily intended for the development of Web applications. The first version of PHP was created by Rasmus Lerdorf and contributed under an open source license in 1995. PHP was initially a very simple HTML templating engine, but over time the developers of PHP added database access functionality, rewrote the interpreter, introduced object-oriented support, and improved performance. Today, PHP has become a popular language for Web application development because of its focus on practical solutions and support for the most commonly required functionality in Web applications.

For the easiest install and configuration experience on Linux®, UNIX®, or Windows® operating systems, you can download and install Zend Core for IBM for use in production systems. Paid support for Zend Core for IBM is available from Zend. On Windows, precompiled binary versions of PHP are available for download from http://php.net/. Most Linux distributions include a precompiled version of PHP. On UNIX operating systems that do not include a precompiled version of PHP, you can compile your own version of PHP.

PHP is a modular language that enables you to customize the available functionality through the use of extensions. These extensions can simplify tasks such as reading, writing, and manipulating XML, creating SOAP clients and servers, and encrypting communications between server and browser. The most popular extensions for PHP, however, provide read and write access to databases so that you can easily create a dynamic database-driven Web site. IBM® supports access to DB2 databases from PHP applications through two extensions offering distinct sets of features:

- ibm_db2 is an extension written, maintained, and supported by IBM for access to DB2 databases. The ibm_db2 extension offers a procedural application programming interface (API) that, in addition to the normal create, read, update, and write database operations, also offers extensive access to the database metadata. You can compile the ibm_db2 extension with either PHP 4 or PHP 5.
- PDO_ODBC is a driver for the PHP Data Objects (PDO) extension that offers access to DB2 databases through the standard object-oriented database interface introduced in PHP 5.1. Despite its name, you can compile the PDO_ODBC extension directly against the DB2 libraries to avoid the communications overhead and potential interference of an ODBC driver manager.

A third extension, Unified ODBC, has historically offered access to DB2 database systems. It is not recommended that you write new applications with this extension because ibm_db2 and PDO_ODBC both offer significant performance and stability benefits over Unified ODBC. The ibm_db2 extension API makes porting an application that was previously written for Unified ODBC almost as easy as globally changing the `odbc_` function name to `db2_` throughout the source code of your application.

Related tasks:

- "Setting up the PHP environment on Linux or UNIX" on page 5
- "Setting up the PHP environment on Windows" on page 4

Setting up the PHP environment on Windows

DB2 supports database access for client applications written in the PHP programming language using either or both of the ibm_db2 extension and the PDO_ODBC driver for the PHP Data Objects (PDO) extension. To install a binary version of PHP with support for DB2 on Windows, you can download and install the freely available Zend Core for IBM from http://zend.com/core/ibm/. However, you can also manually install the precompiled binary version of PHP on Windows.

Prerequisites:

The Apache HTTP Server must be installed on your system.

Procedure:

To install a precompiled version of PHP from http://www.php.net and enable support for DB2 on Windows:

1. Download the latest version of the PHP zip package and the collection of PECL modules zip package from http://www.php.net. The latest version of PHP at the time of writing is PHP 5.1.2.
2. Extract the PHP zip package into an install directory.
3. Extract the collection of PECL modules zip package into the `\ext\` subdirectory of your PHP installation directory.
4. Create a new file named `php.ini` in your installation directory by making a copy of the `php.ini-recommended` file.
5. Open the `php.ini` file in a text editor and add the following lines.
   - To enable the PDO extension and PDO_ODBC driver:

     ```
     extension=php_pdo.dll
     extension=php_pdo_odbc.dll
     ```

     Note: On Windows, the PDO_ODBC driver uses the Windows ODBC Driver Manager to connect to database systems. To access DB2 database systems with PDO_ODBC on Windows, you must install the IBM DB2® Driver for ODBC and CLI.
   - To enable the ibm_db2 extension:

     ```
     extension=php_ibm_db2.dll
     ```
6. Enable PHP support in Apache HTTP Server 2.x by adding the following lines to your `httpd.conf` file, in which *phpdir* refers to the PHP install directory:

   ```
   LoadModule php5_module 'phpdir/php5apache2.dll'
   AddType application/x-httpd-php .php
   PHPIniDir 'phpdir'
   ```
7. Restart the Apache HTTP Server to enable the changed configuration.

Related concepts:

- "Introduction to PHP application development for DB2" on page 3

Related tasks:

- "Connecting to a DB2 database with PHP (ibm_db2)" on page 7
- "Connecting to a DB2 database with PHP (PDO)" on page 23
- "Setting up the PHP environment on Linux or UNIX" on page 5

Setting up the PHP environment on Linux or UNIX

DB2 supports database access for client applications written in the PHP programming language using either or both of the ibm_db2 extension and the PDO_ODBC driver for the PHP Data Objects (PDO) extension. To install a binary version of PHP with support for DB2 on Linux or AIX, you can download and install the freely available Zend Core for IBM from http://zend.com/core/ibm/. However, you can also manually compile and install PHP from source.

Prerequisites:

- The Apache HTTP Server must be installed on your system.
- The DB2 development header files and libraries must be installed on your system.
- The gcc compiler and other development packages including apache-devel, autoconf, automake, bison, flex, gcc, and libxml2-devel package must be installed on your system.

Procedure:

To compile PHP from source with support for DB2 on Linux or UNIX:

1. Download the latest version of the PHP tarball from http://www.php.net. The latest version of PHP at the time of writing is PHP 5.1.2.
2. Untar the file by issuing the following command:
   ```
   tar xjf php-5.1.2.tar.bz2
   ```
3. Change directories into the newly created `php-5.1.2` directory.
4. Configure the makefile by issuing the `configure` command. Specify the features and extensions you want to include in your custom version of PHP. A typical configure command includes the following options:
   ```
   ./configure --enable-cli --disable-cgi --with-apxs2=/usr/sbin/apxs2
   --with-zlib --with-pdo-odbc=ibm-db2
   ```

   The configure options have the following effects:

   --enable-cli
   Enables the command line mode of PHP access.

   --disable-cgi
   Disables the Common Gateway Interface (CGI) mode of PHP access.

   --with-apxs2=/usr/sbin/apxs2
   Enables the Apache 2 dynamic server object (DSO) mode of PHP access.

   --with-zlib
   Enables zlib compression support.

   --with-pdo-odbc=ibm-db2
   Enables the PDO_ODBC driver using the DB2 Call Level Interface library to access database systems. To specify a location for the DB2 header files and libraries, append `,`*`location`* where *location* refers to the directory in which DB2 is installed.
5. Compile the files by issuing the `make` command.
6. Install the files by issuing the `make install` command. Depending on how you configured the PHP install directory using the `configure` command, you

might need root authority to successfully issue this command. This should install the executable files and update the Apache HTTP Server configuration to support PHP.

7. Install the ibm_db2 extension by issuing the following command as a user with root authority:

   ```
   pecl install ibm_db2
   ```

   This command downloads, configure, compiles, and installs the ibm_db2 extension for PHP.
8. Copy the `php.ini-recommended` file to the configuration file path for your new PHP installation. To determine the configuration file path, issue the `php -i` command and look for the `php.ini` keyword. Rename the file to `php.ini`.
9. Open the new `php.ini` file in a text editor and add the following lines, where *instance* refers to the name of the DB2 instance on Linux or UNIX..
   - To set the DB2 environment for PDO_ODBC:

     ```
     pdo_odbc.db2instance_name=instance
     ```
   - (Linux or UNIX) To enable the ibm_db2 extension and set the DB2 environment:

     ```
     extension=ibm_db2.so
     ibm_db2.instance_name=instance
     ```
10. Restart the Apache HTTP Server to enable the changed configuration.

Related concepts:

- "Introduction to PHP application development for DB2" on page 3

Related tasks:

- "Connecting to a DB2 database with PHP (ibm_db2)" on page 7
- "Connecting to a DB2 database with PHP (PDO)" on page 23
- "Setting up the PHP environment on Windows" on page 4

Related reference:

- "PHP samples" in *Samples Topics*

Chapter 2. Developing PHP Applications with ibm_db2

Connecting to a DB2 database with PHP (ibm_db2)

You must connect to a DB2 database before you can create, update, delete, or retrieve data from that data source. The ibm_db2 extension for PHP enables you to connect to a DB2 database using either a cataloged connection or a direct TCP/IP connection to the DB2 database management system. You can also create persistent connections to a database. Persistent connections improve performance by keeping the connection open between PHP requests and by reusing the connection when a subsequent PHP script requests a connection with an identical set of credentials.

Prerequisites:

Before connecting to a DB2 database through the ibm_db2 extension, you must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure:

1. Create a connection to a DB2 database:
   - To create a non-persistent connection to a DB2 database, call `db2_connect()` with a *database* value that specifies either a cataloged database name or a complete database connection string for a direct TCP/IP connection.
   - To create a persistent connection to a DB2 database, call `db2_pconnect()` with a *database* value that specifies either a cataloged database name or a complete database connection string for a direct TCP/IP connection.
2. Check the value returned by `db2_connect()` or `db2_pconnect`.
   - If the value returned by `db2_connect()` or `db2_pconnect` is FALSE, the connection attempt failed. You can retrieve diagnostic information through `db2_conn_error()` and `db2_conn_errormsg()`.
   - If the value returned by `db2_connect()` or `db2_pconnect` is not FALSE, the connection attempt succeeded. You can use the connection resource to create, update, delete, or retrieve data with other ibm_db2 functions.

When you create a connection by calling `db2_connect()` , PHP closes the connection to the database:

- When you call `db2_close()` for the connection,
- When you set the connection resource to NULL,
- Or when the PHP script finishes.

When you create a connection by calling `db2_pconnect()` , PHP ignores any calls to `db2_close()` for the specified connection resource and keeps the connection to the database open for subsequent PHP scripts.

Related reference:

- "db2_pconnect - Returns a persistent connection to a database" on page 45
- "db2_close - Closes a database connection" on page 38
- "db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt" on page 40
- "db2_conn_errormsg - Returns the last connection error message and SQLCODE value" on page 41

• "db2_connect - Returns a connection to a database" on page 42

Retrieving database metadata (ibm_db2)

Some classes of applications, such as administration interfaces, need to dynamically reflect the structure and SQL objects contained in arbitrary databases. One approach to retrieving metadata about a database is to issue SELECT statements directly against the system catalog tables; however, the schema of the system catalog tables may change between versions of DB2, or the schema of the system catalog tables on DB2 Database for Linux, UNIX, and Windows may differ from the schema of the system catalog tables on DB2 for z/OS. Rather than laboriously maintaining these differences in your application code, the ibm_db2 extension for PHP offers a standard set of functions that return metadata for databases served by DB2 Database for Linux, UNIX, and Windows, Cloudscape™, and, through DB2 Connect, DB2 for z/OS and DB2 Universal Database for iSeries.

Prerequisites:

- You must set up the PHP environment on your system and enable the ibm_db2 extension.
- You must have a connection resource returned from `db2_connect()` or `db2_pconnect()`.

Procedure:

1. Call the function that returns the metadata which you require:

   `db2_client_info()`
   Returns metadata about the DB2 client software and configuration.

   `db2_column_privileges()`
   Lists the columns and associated privileges for a table.

   `db2_columns()`
   Lists the columns and associated metadata for a table.

   `db2_foreign_keys()`
   Lists the foreign keys for a table.

   `db2_primary_keys()`
   Lists the primary keys for a table.

   `db2_procedure_columns()`
   Lists the parameters for one or more stored procedures.

   `db2_procedures()`
   Lists the stored procedures registered in the database.

   `db2_server_info()`
   Returns metadata about the database management system software and configuration.

   `db2_special_columns()`
   Lists the unique row identifiers for a table.

   `db2_statistics()`
   Lists the indexes and statistics for a table.

   `db2_table_privileges()`
   Lists tables and their associated privileges in the database.

Note that while most of the ibm_db2 metadata functions accept a qualifier or catalog parameter, this parameter should only be set to a non-NULL value when you are connected to DB2 for z/OS.

2. Depending on which metadata function you called,
   - The `db2_client_info()` and `db2_server_info()` functions directly return a single object with read-only properties. You can use the properties of these objects to create an application that behaves differently depending on the database management system to which it connects. For example, rather than encoding a limit of the lowest common denominator for all possible database management systems, a Web-based database administration application built on the ibm_db2 extension could use the `db2_server_info()->MAX_COL_NAME_LEN` property to dynamically display text fields for naming columns with maximum lengths that correspond to the maximum length of column names on the database management system to which it is connected.
   - The other metadata functions return result sets with columns defined for each function. Retrieve rows from the result set using the normal ibm_db2 functions for this purpose.

Note that calling metadata functions consumes a significant amount of database management system resources. If possible, consider caching the results of your calls for subsequent usage.

Related tasks:

- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

Related reference:

- "db2_foreign_keys - Returns a result set listing the foreign keys for a table" on page 73
- "db2_primary_keys - Returns a result set listing primary keys for a table" on page 74
- "db2_procedure_columns - Returns a result set listing stored procedure parameters" on page 76
- "db2_procedures - Returns a result set listing the stored procedures registered in a database" on page 78
- "db2_server_info - Returns an object with properties that describe the DB2 database management system" on page 79
- "db2_special_columns - Returns a result set listing the unique row identifier columns for a table" on page 83
- "db2_statistics - Returns a result set listing the index and statistics for a table" on page 85
- "db2_table_privileges - Returns a result set listing the tables and associated privileges in a database" on page 87
- "db2_tables - Returns a result set listing the tables and associated metadata in a database" on page 88
- "db2_client_info - Returns an object with properties that describe the DB2 database client" on page 68
- "db2_column_privileges - Returns a result set listing the columns and associated privileges for a table" on page 70
- "db2_columns - Returns a result set listing the columns and associated metadata for a table" on page 71

Executing XQuery expressions in PHP (ibm_db2)

After connecting to a DB2 database, your PHP script is ready to issue XQuery expressions. The `db2_exec()` and `db2_execute()` functions execute SQL statements, through which you can pass your XQuery expressions. A typical use of `db2_exec()` is to set the default schema for your application in a common include file or base class.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Restrictions:

To avoid the security threat of injection attacks, `db2_exec()` should only be used to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the XQuery expression can expose your application to injection attacks.

Procedure:

1. Call `db2_exec()` with the following arguments:
   a. The connection resource;
   b. A string containing the SQL statement, including the XQuery expression. The XQuery expression needs to be wrapped in a XMLQUERY clause in the SQL statement.
   c. (Optional): an array containing statement options

      DB2_ATTR_CASE
      For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to *DB2_CASE_NATURAL*, which returns column names as they are returned by DB2. You can set this parameter to *DB2_CASE_LOWER* to force column names to lower case, or to *DB2_CASE_UPPER* to force column names to upper case.

      DB2_ATTR_CURSOR
      This option sets the type of cursor that ibm_db2 returns for result sets. By default, ibm_db2 returns a forward-only cursor (*DB2_FORWARD_ONLY*) which returns the next row in a result set for every call to `db2_fetch_array()`, `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()`. You can set this parameter to *DB2_SCROLLABLE* to request a scrollable cursor so that the ibm_db2 fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set .
2. Check the value returned by `db2_exec()`:
   - If the value is FALSE, the SQL statement failed. You can retrieve diagnostic information through the `db2_stmt_error()` and `db2_stmt_errormsg()` functions.
   - If the value is not FALSE, the SQL statement succeeded and returned a statement resource that can be used in subsequent function calls related to this query.

Example:

```
<?php
$xquery = '$doc/customerinfo/phone';
$stmt = db2_exec($conn, "select xmlquery('$xquery'
PASSING INFO AS \"doc\") from customer");?>
```

Executing a single SQL statement in PHP (ibm_db2)

After connecting to a DB2 database, most PHP scripts will execute one or more SQL statements. The `db2_exec()` function executes a single SQL statement that accepts no input parameters. A typical use of `db2_exec()` is to set the default schema for your application in a common include file or base class.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Restrictions:

To avoid the security threat of SQL injection attacks, `db2_exec()` should only be used to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Procedure:

1. Call `db2_exec()` with the following arguments:
   a. The connection resource;
   b. A string containing the SQL statement;
   c. (Optional): an array containing statement options

      DB2_ATTR_CASE
      For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to *DB2_CASE_NATURAL*, which returns column names as they are returned by DB2. You can set this parameter to *DB2_CASE_LOWER* to force column names to lower case, or to *DB2_CASE_UPPER* to force column names to upper case.

      DB2_ATTR_CURSOR
      This option sets the type of cursor that ibm_db2 returns for result sets. By default, ibm_db2 returns a forward-only cursor (*DB2_FORWARD_ONLY*) which returns the next row in a result set for every call to `db2_fetch_array()`, `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()`. You can set this parameter to *DB2_SCROLLABLE* to request a scrollable cursor so that the ibm_db2 fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set .
2. Check the value returned by `db2_exec()`:
   - If the value is FALSE, the SQL statement failed. You can retrieve diagnostic information through the `db2_stmt_error()` and `db2_stmt_errormsg()` functions.

- If the value is not FALSE, the SQL statement succeeded and returned a statement resource that can be used in subsequent function calls related to this query.

If the SQL statement selected rows using a scrollable cursor, or inserted, updated, or deleted rows, you can call `db2_num_rows()` to return the number of rows that the statement returned or affected. If the SQL statement returned a result set, you can begin fetching rows.

Related tasks:

- "Fetching columns from result sets in PHP (ibm_db2)" on page 15
- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_exec - Executes an SQL statement directly" on page 50

Preparing and executing SQL statements in PHP (ibm_db2)

Most SQL statements in PHP applications use variable input to determine the results of the SQL statement. To pass user-supplied input to an SQL statement safely, prepare a statement using parameter markers (?) representing the variable input. When you execute the prepared statement, you bind input values to the parameter markers. The database engine ensures that each input value is treated as a single parameter, preventing SQL injection attacks against your application. Compared to statements issued through `db2_exec()`, prepared statements offer a performance advantage because the database management system creates an access plan for each prepared statement that it can reuse if the statement is reissued subsequently.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Restrictions:

You can only use parameter markers as a place holder for column or predicate values. The SQL compiler would be unable to create an access plan for a statement that used parameter markers in place of column names, table names, or other SQL identifiers.

Procedure:

To prepare and execute an SQL statement:

1. Call `db2_prepare()` with the following arguments:
   a. The connection resource
   b. A string containing the SQL statement, including parameter markers (?) for any column or predicate values that require variable input
   c. (Optional): An array containing statement options

      DB2_ATTR_CASE
      For compatibility with database systems that do not follow the SQL

standard, this option sets the case in which column names will be returned to the application. By default, the case is set to *DB2_CASE_NATURAL*, which returns column names as they are returned by DB2. You can set this parameter to *DB2_CASE_LOWER* to force column names to lower case, or to *DB2_CASE_UPPER* to force column names to upper case.

DB2_ATTR_CURSOR
This option sets the type of cursor that ibm_db2 returns for result sets. By default, ibm_db2 returns a forward-only cursor (*DB2_FORWARD_ONLY*) which returns the next row in a result set for every call to `db2_fetch_array()`, `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()`. You can set this parameter to *DB2_SCROLLABLE* to request a scrollable cursor so that the ibm_db2 fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set.

2. Check the value returned by `db2_prepare()`.
   - If the value is FALSE, the SQL statement failed. You can retrieve diagnostic information through the `db2_stmt_error()` and `db2_stmt_errormsg()` functions.
   - If the value is not FALSE, the SQL statement succeeded and returned a statement resource that can be used in subsequent function calls related to this query.
3. (Optional): Call `db2_bind_param()` for each parameter marker in the SQL statement with the following arguments:
   a. The statement resource
   b. An integer representing the position of the parameter marker in the SQL statement
   c. The value to use in place of the parameter marker
4. Call `db2_execute` with the following arguments:
   a. The statement resource
   b. (Optional): An array containing the values to use in place of the parameter markers, in order

Example:

```
$sql = "SELECT firstnme, lastname FROM employee WHERE bonus > ? AND bonus < ?";
$stmt = db2_prepare($conn, $sql);
if (!$stmt) {
  // Handle errors
}

// Explicitly bind parameters
db2_bind_param($stmt, 1, $_POST['lower']);
db2_bind_param($stmt, 2, $_POST['upper']);

db2_execute($stmt);
// Process results

// Invoke prepared statement again using dynamically bound parameters
db2_execute($stmt, array($_POST['lower'], $_POST['upper']);
```

If you execute a prepared statement that returns one or more result sets, you can begin retrieving rows from the statement resource by calling the `db2_fetch_array()` , `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()` functions.

Related tasks:

- "Fetching columns from result sets in PHP (ibm_db2)" on page 15
- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16
- "Handling errors and warning messages (ibm_db2)" on page 18

Related reference:

- "db2_bind_param - Binds a PHP variable to an SQL statement parameter" on page 48
- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

Inserting large objects in PHP (ibm_db2)

The ibm_db2 extension supports the entire range of DB2 data types, including character large object (CLOB) and binary large object (BLOB) data types. When you insert a large object into a database, you can treat the large object simply as a PHP string. However, treating a large object as a PHP string is an approach that consumes more resources on your PHP server than necessary. Rather than loading all of the data for a large object into a PHP string, and then passing that to DB2 through an INSERT statement, you can insert large objects directly from a file on your PHP server.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure:

To insert a large object into the database directly from a file:

1. Call `db2_prepare()` to prepare an INSERT statement with a parameter marker representing the large object column.
2. Set the value of a PHP variable to the path and name of the file that contains the data for the large object. The path can be relative or absolute, and is subject to the access permissions of the PHP executable.
3. Call `db2_bind_param()` to bind the parameter marker to the file that contains the data for the large object. The third parameter is a string representing the name of the PHP variable that holds the name of the file containing the data for the large object. The fourth parameter is `DB2_PARAM_FILE`, which tells the ibm_db2 extension to retrieve the data from a file.
4. Call `db2_execute()` to issue the INSERT statement and bind the data from the file into the database.

Example:

```
$stmt = db2_prepare($conn, "INSERT INTO animal_pictures(picture) VALUES (?)");

$picture = "/opt/albums/spook/grooming.jpg";
$rc = db2_bind_param($stmt, 1, "picture", DB2_PARAM_FILE);
$rc = db2_execute($stmt);
```

Related tasks:

- "Fetching large objects in PHP (ibm_db2)" on page 17

Related reference:

- "db2_bind_param - Binds a PHP variable to an SQL statement parameter" on page 48
- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

Fetching columns from result sets in PHP (ibm_db2)

When you execute a statement that returns one or more result sets, you usually need to iterate through the returned rows of each result set. If your result set includes columns with extremely large data (such as a column defined with a BLOB or CLOB data type), you might prefer to retrieve the data on a column-by-column basis to avoid using too much memory in your PHP process.

Prerequisites:

- You must set up the PHP environment on your system and enable the ibm_db2 extension.
- You must have a statement resource returned from `db2_exec()` or `db2_execute()` with one or more associated result sets.

Procedure:

1. Call the `db2_fetch_row()` function to advance the cursor to the next row in the result set. The first time you call a fetch function for a given result set advances the cursor to the first row of the result set. If you requested a scrollable cursor, you can also specify the number of the row in the result set that you want to retrieve.
2. Check the result returned by `db2_fetch_row()`. If the result is `FALSE`, there are no more rows in the result set.
3. Call the `db2_result()` function to retrieve the value from the requested column by passing either an integer representing the position of the column in the row (starting with 0 for the first column), or a string representing the name of the column.

Example:

```
<?php
$sql = 'SELECT name, breed FROM animals WHERE weight < ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array(10));
while (db2_fetch_row($stmt)) {
    $name = db2_result($stmt, 0);
    $breed = db2_result($stmt, 'BREED');
    print "$name $breed";
}
?>
```

Related reference:

- "db2_fetch_row - Sets the result set pointer to the next row or requested row" on page 64
- "db2_result - Returns a single column from a row in the result set" on page 67

Fetching rows from result sets in PHP (ibm_db2)

When you execute a statement that returns one or more result sets, you usually need to iterate through the returned rows.

Prerequisites:

- You must set up the PHP environment on your system and enable the ibm_db2 extension.
- You must have a statement resource returned from `db2_exec()` or `db2_execute()` with one or more associated result sets.

Procedure:

Call the ibm_db2 fetch function that returns the data from the row in the format you prefer:

`db2_fetch_array()`
Returns an array containing the data corresponding to the columns of the row indexed by column position starting at 0

`db2_fetch_assoc()`
Returns an array containing the data corresponding to the columns of the row indexed by column name.

`db2_fetch_both()`
Returns an array containing the data corresponding to the columns of the row indexed by both column name and by column position starting at 0.

`db2_fetch_object()`
Returns an object containing the data from the row. The object holds properties matching the column names of the row which, when accessed, return the corresponding values of the columns.

You must pass the statement resource as the first argument. If you requested a scrollable cursor when you executed `db2_exec()` or `db2_prepare()`, you can pass an absolute row number as the second argument. With the default forward-only cursor, each call to a fetch method returns the next row in the result set. You can continue fetching rows until the fetch method returns `FALSE`, which signifies that you have reached the end of the result set.

Example:

```
$stmt = db2_exec($conn, "SELECT firstnme, lastname FROM employee");
while ($row = db2_fetch_object($stmt)) {
  print "Name: <p>{$row->FIRSTNME} {$row->LASTNAME}</p>";
}
```

Related tasks:

- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set" on page 58
- "db2_fetch_assoc - Returns an array, indexed by column name, representing a row in a result set" on page 60

- "db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set" on page 61
- "db2_fetch_object - Returns an object with properties representing columns in the fetched row" on page 63

Fetching large objects in PHP (ibm_db2)

The ibm_db2 extension supports the entire range of DB2 data types, including character large object (CLOB) and binary large object (BLOB) data types. When you fetch a large object from a result set, you can treat the large object simply as a PHP string. However, treating a large object as a PHP string is an approach that consumes more resources on your PHP server than necessary. If your ultimate goal is to create a file that contains the data for a large object, you can save system resources by fetching large objects directly into a file on your PHP server.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure:

To fetch a large object from the database directly into a file:

1. Create a PHP variable representing a stream. For example, the return value from a call to `fopen()`.
2. Call `db2_prepare()` to create a SELECT statement.
3. Call `db2_bind_param()` to bind the output column for the large object to the PHP variable representing the stream. The third parameter is a string representing the name of the PHP variable that holds the name of the file that is to contain the data from the large object. The fourth parameter is `DB2_PARAM_FILE`, which tells the ibm_db2 extension to write the data into a file.
4. Call `db2_execute()` to issue the SQL statement.
5. Call an ibm_db2 fetch function of your choice (for example, `db2_fetch_object()`), to retrieve the next row in the result set.

Example:

```
$stmt = db2_prepare($conn, "SELECT name, picture FROM animal_pictures");
$picture = fopen("/opt/albums/spook/grooming.jpg", "wb");
$rc = db2_bind_param($stmt, 1, "nickname", DB2_CHAR, 32);
$rc = db2_bind_param($stmt, 2, "picture", DB2_PARAM_FILE);
$rc = db2_execute($stmt);
$rc = db2_fetch_object($stmt);
```

Managing transactions in PHP (ibm_db2)

By default, the ibm_db2 extension opens every connection in autocommit mode. Autocommit mode helps prevent locking escalation issues that can impede the performance of highly scalable Web applications. In some scripts, however, you might need to roll back a transaction containing one or more SQL statements. The ibm_db2 extension enables you to exert fine-grained control over your transactions.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Restrictions:

You must use a regular connection created with `db2_connect()` to control database transactions in PHP. Persistent connections always use autocommit mode.

Procedure:

To begin a transaction:

1. Create a database connection using the `"AUTOCOMMIT" => DB2_AUTOCOMMIT_OFF` setting in the `db2_connect()` options array. You can also turn autocommit off for an existing connection resource by calling `db2_autocommit($conn, DB2_AUTOCOMMIT_OFF)`. Calling `db2_autocommit()` requires additional communication from PHP to the database management system and may affect the performance of your PHP scripts.
2. Issue one or more SQL statements within the scope of the database transaction using the connection resource for which transactions have been enabled.
3. Commit or rollback the transaction:
   - To commit the transaction, call `db2_commit()`.
   - To rollback the transaction, call `db2_rollback()`.
4. (Optional): Return the database connection to autocommit mode by calling `db2_autocommit($conn, DB2_AUTOCOMMIT_ON)`. If you issue another SQL statement without returning the database connection to autocommit mode, you begin a new transaction that will require a commit or rollback.

If you issue SQL statements in a transaction and the script ends without explicitly committing or rolling back the transaction, the ibm_db2 extension automatically rolls back any work performed in the transaction.

Example:

```
$conn = db2_connect('SAMPLE', 'db2inst1', 'ibmdb2', array(
  'AUTOCOMMIT' => DB2_AUTOCOMMIT_ON));

// Issue one or more SQL statements within the transaction
$result = db2_exec($conn, 'DELETE FROM TABLE employee');
if ($result === FALSE) {
  print '<p>Unable to complete transaction!</p>';
  db2_rollback($conn);
}
else {
  print '<p>Successfully completed transaction!</p>';
  db2_commit($conn);
}
```

Related reference:

- "db2_autocommit - Returns or sets the AUTOCOMMIT state for a database connection" on page 37
- "db2_commit - Commits a transaction" on page 39
- "db2_rollback - Rolls back a transaction" on page 46

Handling errors and warning messages (ibm_db2)

Problems occasionally happen when you attempt to connect to a database or issue an SQL statement. The password for your connection might be incorrect, the table you referred to in a SELECT statement might not exist, or the syntax for an SQL statement might be invalid. You need to code defensively and use the

error-handling functions offered by the ibm_db2 extension to enable your application to recover gracefully from a problem.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure:

1. Check the value returned from the ibm_db2 function to ensure the function returned successfully. If the function can return the value 0, such as `db2_num_rows()`, you must explicitly test whether the value was `FALSE` using PHP's `===` operator.
2. If the function returned `FALSE` instead of the connection resource, statement resource, or numeric value you expected, call the ibm_db2 error handling function appropriate to the application context and the needs of your application:

   Connection errors

   To retrieve the SQLSTATE returned by the last connection attempt, call `db2_conn_error()`. To retrieve a descriptive error message appropriate for an application error log, call `db2_conn_errormsg()`.

   ```
   $connection = db2_connect($database, $user, $password);
   if (!$connection) {
       $this->state = db2_conn_error();
       return false;
   }
   ```

   SQL errors (executing SQL statements directly and fetching results)

   To retrieve the SQLSTATE returned by the last attempt to prepare or execute an SQL statement, or to fetch a result from a result set, call `db2_stmt_error()`. To retrieve a descriptive error message appropriate for an application error log, call `db2_stmt_errormsg()`.

   ```
   $stmt = db2_prepare($connection, "DELETE FROM employee
   WHERE firstnme = ?");
   if (!$stmt) {
       $this->state = db2_stmt_error();
       return false;
   }
   ```

   SQL errors (executing prepared statements)

   If `db2_prepare()` returned successfully, but a subsequent call to `db2_execute()` fails, call `db2_stmt_error()` or `db2_stmt_errormsg()` and pass the resource returned from the call to `db2_prepare()` as the argument.

   ```
   $success = db2_execute($stmt, array('Dan');
   if (!$success) {
       $this->state = db2_stmt_error($stmt);
       return $false;
   }
   ```

3. To avoid the possibility of security vulnerabilities resulting from directly displaying the raw SQLSTATE returned from the database, and to offer a better overall user experience in your Web application, use a `switch` structure to recover from known error states or return custom error messages.

   ```
   switch($this->state):
       case '22001':
           // More data than allowed for the defined column
           $message = "You entered too many characters for this value.";
           break;
   ```

Related reference:

- "db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt" on page 40
- "db2_conn_errormsg - Returns the last connection error message and SQLCODE value" on page 41
- "db2_stmt_error - Returns a string containing the SQLSTATE returned by an SQL statement" on page 57
- "db2_stmt_errormsg - Returns a string containing the last SQL statement error message" on page 57

Calling stored procedures with OUT or INOUT parameters in PHP (ibm_db2)

DB2 supports stored procedures with parameters that only accept an input value (IN parameters), that only return an output value (OUT parameters), or that accept an input value and return an output value (INOUT). With the ibm_db2 extension for PHP you can handle IN parameters like any other parameter marker in an SQL statement. However, the ibm_db2 extension also enables you to CALL stored procedures with OUT and INOUT parameters and retrieve the output values from those parameters.

Prerequisites:

You must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure:

To call a stored procedure with OUT or INOUT parameters:

1. Call `db2_prepare()` to prepare a CALL statement with parameter markers representing the OUT and INOUT parameters.
2. Call `db2_bind_param()` to bind each parameter marker to the name of the PHP variable that will hold the output value of the parameter after the CALL statement has been issued. For INOUT parameters, the value of the PHP variable is passed as the input value of the parameter when the CALL statement is issued. Set the fourth parameter for `db2_bind_param()` to either `DB2_PARAM_OUT`, representing an OUT parameter, or `DB2_PARAM_INOUT`, representing an INOUT parameter.
3. Call `db2_execute()` to issue the CALL statement and bind the data from the stored procedure into the PHP variables.

Example:

```
$sql = 'CALL match_animal(?, ?)';
$stmt = db2_prepare($conn, $sql);

$second_name = "Rickety Ride";
$weight = 0;

db2_bind_param($stmt, 1, "second_name", DB2_PARAM_INOUT);
db2_bind_param($stmt, 2, "weight", DB2_PARAM_OUT);

print "Values of bound parameters _before_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";

db2_execute($stmt);
```

```
print "Values of bound parameters _after_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";
```

Related reference:

- "db2_bind_param - Binds a PHP variable to an SQL statement parameter" on page 48
- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

Calling stored procedures that return multiple result sets in PHP (ibm_db2)

DB2 enables you to create and call stored procedures that return more than one result set. The ibm_db2 extension for PHP fully supports this capability through the `db2_next_result()` function. You can use this function to fetch rows from different result sets returned by a single call to the same stored procedure in any order you prefer.

Prerequisites:

- You must set up the PHP environment on your system and enable the ibm_db2 extension.
- You must have a statement resource returned from calling a stored procedure with `db2_exec()` or `db2_execute()`.

Procedure:

To return multiple result sets from a stored procedure:

1. The first result set is associated with the statement resource returned by the CALL statement.
2. Pass the original statement resource as the first argument to `db2_next_result()` to retrieve the second and subsequent result sets. This function returns `FALSE` when no more result sets are available.

Example:

```
$stmt = db2_exec($conn, 'CALL multiResults()');

print "Fetching first result set\n";
while ($row = db2_fetch_array($stmt)) {
    // work with row
}

print "\nFetching second result set\n";
$result_2 = db2_next_result($stmt);
if ($result_2) {
  while ($row = db2_fetch_array($result_2)) {
    // work with row
  }
}

print "\nFetching third result set\n";
$result_3 = db2_next_result($stmt);
if ($result_3) {
  while ($row = db2_fetch_array($result_3)) {
    // work with row
  }
}
```

Related tasks:

- "Calling stored procedures with OUT or INOUT parameters in PHP (ibm_db2)" on page 20

Related reference:

- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_next_result - Requests the next result set from a stored procedure" on page 65
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

Chapter 3. Developing PHP Applications with PDO

Connecting to a DB2 database with PHP (PDO)

You must connect to a DB2 database before you can create, update, delete, or retrieve data from that data source. The PHP Data Objects (PDO) interface for PHP enables you to connect to a DB2 database using either a cataloged connection or a direct TCP/IP connection to the DB2 database management system through the PDO_ODBC extension. You can also create persistent connections to a data source that improve performance by keeping the connection open between PHP requests and reusing the connection when a subsequent PHP script requests a connection with an identical set of credentials.

Prerequisites:

You must set up the PHP 5.1 or higher environment on your system and enable the PDO and PDO_ODBC extensions.

Procedure:

1. Create a connection to the DB2 database by calling the PDO constructor within a `try{}` block. Pass a *DSN* value that specifies `odbc:` for the PDO_ODBC extension, followed by either a cataloged database name or a complete database connection string for a direct TCP/IP connection.
   - (Windows): By default, PDO_ODBC uses Windows ODBC Driver Manager connection pooling to minimize connection resources and improve connection performance.
   - (Linux and UNIX): PDO_ODBC offers persistent connections if you pass `array(PDO::ATTR_PERSISTENT => TRUE)` as the fourth argument to the PDO constructor.
2. (Optional): Set error handling options for the PDO connection in the fourth argument to the PDO constructor:
   - by default, PDO sets an error message that can be retrieved through `PDO::errorInfo()` and an SQLCODE that can be retrieved through `PDO::errorCode()` when any error occurs; to request this mode explicitly, set `PDO::ATTR_ERRMODE => PDO::ERRMODE_SILENT`
   - to issue a PHP `E_WARNING` when any error occurs, in addition to setting the error message and SQLCODE, set `PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNING`
   - to throw a PHP exception when any error occurs, set `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION`
3. Catch any exception thrown by the `try{}` block in a corresponding `catch {}` block.

```
try {
  $connection = new PDO("odbc:SAMPLE", "db2inst1", "ibmdb2", array(
    PDO::ATTR_PERSISTENT => TRUE,
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION)
  );
}
catch (Exception $e) {
  echo($e->getMessage());
}
```

When you create a connection through PDO, PHP closes the connection to the database:

- when you set the PDO object to NULL,
- or when the PHP script finishes.

Related reference:

- "PDO::__construct - Creates a PDO instance representing a connection to a database" on page 99
- "PDO::errorCode - Fetch the SQLSTATE associated with the last operation on the database handle" on page 102
- "PDO::errorInfo - Fetch extended error information associated with the last operation on the database handle" on page 103

Executing a single SQL statement in PHP that returns no result sets (PDO)

After connecting to a DB2 database, most PHP scripts will execute one or more SQL statements. The `PDO::exec()` method executes a single SQL statement that accepts no input parameters and returns no result set. A typical use of `PDO::exec()` is to set the default schema for your application in a common include file or base class.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO_ODBC extension.

Restrictions:

To avoid the security threat of SQL injection attacks, `PDO::exec()` should only be used to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Procedure:

To execute a single SQL statement in PHP:

1. Call the `PDO::exec()` method on the PDO connection object with a string containing the SQL statement.
2. If the SQL statement inserted, modified, or deleted rows, `PDO::exec()` returns an integer value representing the number of rows that were inserted, modified, or deleted. To determine if `PDO::exec()` returned FALSE indicating an error condition or 0 indicating that no rows were inserted, modified, or deleted, you must use the === operator to strictly test the returned value against FALSE.

Example:

```
$conn = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$result = $conn->exec('SET SCHEMA myapp');
if ($result === FALSE) {
  print "Failed to set schema: " . $conn->errorMsg();
}
```

Related tasks:

- "Executing a single SQL statement in PHP that returns a result set (PDO)" on page 25
- "Preparing and executing SQL statements (PDO)" on page 26

Related reference:

- "PDO::exec - Execute an SQL statement and return the number of affected rows" on page 104

Executing a single SQL statement in PHP that returns a result set (PDO)

After connecting to a DB2 database, most PHP scripts will execute one or more SQL statements. The `PDO::query()` method executes a single SQL statement that accepts no input parameters and returns one or more result sets. A typical use of `PDO::query()` is to execute a static SELECT statement.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO_ODBC extension.

Restrictions:

To avoid the security threat of SQL injection attacks, `PDO::query()` should only be used to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Procedure:

To execute a single SQL statement in PHP that returns a result set:

1. Call the `PDO::query()` method on the PDO connection object with a string containing the SQL statement.
2. Check the value returned by `PDO::query()`.
   - If the value is FALSE, the SQL statement failed. You can retrieve diagnostic information through the `PDO::errorCode()` and `PDO::errorInfo()` methods.
   - If the value is not FALSE, the SQL statement succeeded and returned a PDOStatement resource that can be used in subsequent method calls.

Example:

```
$conn = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$result = $conn->query('SELECT firstnme, lastname FROM employee');
if (!$result) {
  print "<p>Could not retrieve employee list: " . $conn->errorMsg(). "</p>";
}
while ($row = $conn->fetch()) {
  print "<p>Name: {$row[0] $row[1]}</p>";
}
```

After creating a PDOStatement object with `PDO::query()`, you can immediately begin retrieving rows from the object with the `PDOStatement::fetch()` or `PDOStatement::fetchAll()` methods.

Related tasks:

- "Executing a single SQL statement in PHP that returns no result sets (PDO)" on page 24
- "Preparing and executing SQL statements (PDO)" on page 26

Related reference:

- "PDO::query - Executes an SQL statement, returning a result set as a PDOStatement object" on page 108

Preparing and executing SQL statements (PDO)

Most SQL statements in PHP applications use variable input to determine the results of the SQL statement. To pass user-supplied input to an SQL statement safely, prepare a statement using parameter markers (`?`) or named variables representing the variable input. When you execute the prepared statement, you bind input values to the parameter markers. The database engine ensures that each input value is treated as a single parameter, preventing SQL injection attacks against your application. Compared to statements issued through `PDO::exec()`, prepared statements offer a performance advantage because the database management system creates an access plan for each prepared statement that it can reuse if the statement is reissued subsequently.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO_ODBC extension.

Restrictions:

- You can only use parameter markers as a place holder for column or predicate values. The SQL compiler would be unable to create an access plan for a statement that used parameter markers in place of column names, table names, or other SQL identifiers.
- You cannot use both question mark parameter markers (`?`) and named parameter markers (`:name`) in the same SQL statement.

Procedure:

To prepare and execute an SQL statement:

1. Call `PDO::prepare()` with the following arguments:
   a. A string containing the SQL statement including either parameter markers (`?`) or named variables (`:name`) for any column or predicate values that require variable input
   b. (Optional): An array containing statement options

      PDO::ATTR_CURSOR
      This option sets the type of cursor that PDO returns for result sets. By default, PDO returns a forward-only cursor (*PDO::CURSOR_FWDONLY*) which returns the next row in a result set for every call to `PDOStatement::fetch()`. You can set this parameter to *PDO::CURSOR_SCROLL* to request a scrollable cursor.
2. Check the value returned by `PDO::prepare()`.
   - If the value is FALSE, the SQL statement failed. You can retrieve diagnostic information through the `PDO::errorCode()` and `PDO::errorInfo()` methods.
   - If the value is not FALSE, the SQL statement succeeded and returned a PDOStatement object that can be used in subsequent method calls.

3. (Optional): Call `PDOStatement::bindParam()` for each parameter marker in the SQL statement with the following arguments:
   a. The parameter identifier. For question mark parameter markers (?), this is an integer representing the 1-indexed position of the parameter in the SQL statement. For named parameter markers (`:name`), this is a string representing the parameter name.
   b. The value to use in place of the parameter marker
4. Call `PDOStatement::execute()`, optionally passing an array containing the values to use in place of the parameter markers, either in order for question mark parameter markers, or as a `:name => value` associative array for named parameter markers.

Example:

```
$sql = "SELECT firstnme, lastname FROM employee WHERE bonus > ? AND bonus < ?";
$stmt = $conn->prepare($sql);
if (!$stmt) {
  // Handle errors
}

// Explicitly bind parameters
$stmt->bindParam(1, $_POST['lower']);
$stmt->bindParam(2, $_POST['upper']);

$stmt->execute($stmt);

// Invoke statement again using dynamically bound parameters
$stmt->execute($stmt, array($_POST['lower'], $_POST['upper']);
```

If you successfully execute a prepared statement that returns one or more result sets, you can begin retrieving rows from the statement resource by calling the `PDOStatement::fetch()` or `PDOStatement::fetchAll()` methods.

Related reference:

- "PDO::prepare - Prepares a statement for execution and returns a statement object" on page 107
- "PDOStatement::bindParam - Binds a parameter to the specified variable name" on page 113
- "PDOStatement::execute - Executes a prepared statement" on page 118

Inserting large objects in PHP (PDO)

The PDO extension supports the entire range of DB2 data types, including character large object (CLOB) and binary large object (BLOB) data types. When you insert a large object into a database, you can treat the large object simply as a PHP string. However, treating a large object as a PHP string is an approach that consumes more resources on your PHP server than necessary. Rather than loading all of the data for a large object into a PHP string, and then passing that to DB2 through an INSERT statement, you can insert large objects directly from a file on your PHP server.

Prerequisites:

You must set up the PHP 5.1 or higher environment on your system and enable the PDO and PDO_ODBC extensions.

Procedure:

To insert a large object into the database directly from a file:

1. Call `PDO::prepare()` to create a PDOStatement object from an INSERT statement with a parameter marker representing the large object column.
2. Create a PHP variable representing a stream–for example, the return value from a call to `fopen()`.
3. Call `PDOStatement::bindParam()` to bind the parameter marker to the PHP variable representing the stream of data for the large object. The third parameter is a string representing the name of the PHP variable that holds the name of the file containing the data for the large object. The fourth parameter is a PHP constant, `PDO::PARAM_LOB`, which tells the PDO extension to retrieve the data from a file.
4. Call `PDOStatement::execute()` to issue the INSERT statement and bind the data from the file into the database.

Example:

```
$stmt = $conn->prepare("INSERT INTO animal_pictures(picture) VALUES (?)");
$picture = fopen("/opt/albums/spook/grooming.jpg", "rb");
$stmt->bindParam($stmt, 1, $picture, PDO::PARAM_LOB);
$stmt->execute();
```

Related reference:

- "PDO::prepare - Prepares a statement for execution and returns a statement object" on page 107
- "PDOStatement::bindParam - Binds a parameter to the specified variable name" on page 113
- "PDOStatement::execute - Executes a prepared statement" on page 118

Fetching columns from result sets in PHP (PDO)

When you execute a statement that returns one or more result sets, you usually need to iterate through the returned rows of each result set. In some cases, you only need to return a single column from each row in the result set. While you could rewrite a SELECT statement for that purpose, you might not have the privileges required to rewrite a stored procedure that returns more columns than you require.

Prerequisites:

- You must set up the PHP environment on your system and enable the PDO and PDO_ODBC extensions.
- You must have a statement resource returned from `PDO::query()` or `PDOStatement::execute()` with one or more associated result sets.

Restrictions:

If you decide to fetch a column from a row, instead of retrieving all of the columns in the entire row simultaneously, you can only return a single column from each row.

Procedure:

To return a single column from a single row in the result set:

1. Call the `PDOStatement::fetchColumn()` method, specifying the column you want to retrieve as the first argument of the method. Column numbers start at 0. If you do not specify a column, `PDOStatement::fetchColumn()` returns the first column in the row.

To return an array containing a single column from all of the remaining rows in the result set:

1. Call the `PDOStatement::fetchAll()` method, passing `PDO::FETCH_COLUMN` as the first argument, and the column you want to retrieve as the second argument, to return an array of the values for the selected column from the result set. Column numbers start at 0. If you do not specify a column, `PDOStatement::fetchAll(PDO::FETCH_COLUMN)` returns the first column in the row.

Related tasks:

- "Fetching large objects in PHP (PDO)" on page 31
- "Fetching rows from result sets in PHP (PDO)" on page 29

Related reference:

- "PDOStatement::fetchColumn - Returns a single column from the next row of a result set" on page 124
- "PDO::query - Executes an SQL statement, returning a result set as a PDOStatement object" on page 108
- "PDOStatement::execute - Executes a prepared statement" on page 118
- "PDOStatement::fetchAll - Returns an array containing all of the result set rows" on page 122

Fetching rows from result sets in PHP (PDO)

When you execute a statement that returns one or more result sets, you usually need to iterate through the returned rows.

Prerequisites:

- You must set up the PHP environment on your system and enable the PDO extension.
- You must have a PDOStatement object returned from `PDO::query()` or `PDOStatement::execute()` with one or more associated result sets.

Procedure:

To return a single row from a result set as an array or object, call the `PDOStatement::fetch()` method.

To return all of the rows from the result set as an array of arrays or objects, call the `PDOStatement::fetchAll()` method.

By default, PDO returns each row as an array indexed by column name and 0-indexed column position in the row. You can request a different return style by passing one of the following constants as the first parameter of `PDOStatement::fetch()`:

`PDO::FETCH_ASSOC`
Returns an array indexed by column name as returned in your result set.

PDO::FETCH_BOTH (default)
Returns an array indexed by both column name and 0-indexed column number as returned in your result set

PDO::FETCH_BOUND
Returns TRUE and assigns the values of the columns in your result set to the PHP variables to which they were bound with the `PDOStatement::bindParam()` method.

PDO::FETCH_CLASS
Returns a new instance of the requested class, mapping the columns of the result set to named properties in the class.

PDO::FETCH_INTO
Updates an existing instance of the requested class, mapping the columns of the result set to named properties in the class.

PDO::FETCH_LAZY
Combines `PDO::FETCH_BOTH` and `PDO::FETCH_OBJ`, creating the object variable names as they are accessed.

PDO::FETCH_NUM
Returns an array indexed by column number as returned in your result set, starting at column 0.

PDO::FETCH_OBJ
Returns an anonymous object with property names that correspond to the column names returned in your result set.

(Optional): If you requested a scrollable cursor when you called `PDO::query()` or `PDOStatement::execute()`, you can pass two more arguments to `PDOStatement::fetch()`:

1. The fetch orientation for this fetch request:

   PDO::FETCH_ORI_NEXT (default)
   Fetches the next row in the result set.

   PDO::FETCH_ORI_PRIOR
   Fetches the previous row in the result set.

   PDO::FETCH_ORI_FIRST
   Fetches the first row in the result set.

   PDO::FETCH_ORI_LAST
   Fetches the last row in the result set.

   PDO::FETCH_ORI_ABS
   Fetches the absolute row in the result set. Requires a positive integer as the third argument to `PDOStatement::fetch()`.

   PDO::FETCH_ORI_REL
   Fetches the relative row in the result set. Requires a positive or negative integer as the third argument to `PDOStatement::fetch()`.
2. An integer requesting the absolute or relative row in the result set, corresponding to the fetch orientation requested in the second argument to `PDOStatement::fetch()`.

`PDOStatement::fetch()` returns `FALSE` when the last row in the result set has been retrieved for a forward-only result set.

Example:

```
$stmt = $conn->query("SELECT firstnme, lastname FROM employee");
while ($row = $stmt->fetch(PDO::FETCH_NUM)) {
  print "Name: <p>{$row[0] $row[1]}</p>";
}
```

Related tasks:

- "Fetching large objects in PHP (PDO)" on page 31

Related reference:

- "PDOStatement::fetch - Fetches the next row from a result set" on page 119
- "PDOStatement::fetchAll - Returns an array containing all of the result set rows" on page 122

Fetching large objects in PHP (PDO)

The PDO extension supports the entire range of DB2 data types, including character large object (CLOB) and binary large object (BLOB) data types. When you fetch a large object from a result set, you can treat the large object simply as a PHP string. However, treating a large object as a PHP string is an approach that consumes more resources on your PHP server than necessary. If your ultimate goal is to create a file that contains the data for a large object, you can save system resources by fetching large objects directly into a file on your PHP server.

Prerequisites:

You must set up the PHP 5.1 or higher environment on your system and enable the PDO and PDO_ODBC extensions.

Procedure:

To fetch a large object from the database directly into a file:

1. Create a PHP variable representing a stream–for example, the return value from a call to `fopen()`.
2. Call `PDO::prepare()` to create a PDOStatement object from an SQL statement.
3. Call `PDOStatement::bindColumn()` to bind the output column for the large object to the PHP variable representing the stream. The third parameter is a string representing the name of the PHP variable that holds the name of the file that is to contain the data from the large object. The fourth parameter is a PHP constant, `PDO::PARAM_LOB`, which tells the PDO extension to write the data into a file. Note that you must call `PDOStatement::bindColumn()` to assign a different PHP variable for every column in the result set.
4. Call `PDOStatement::execute()` to issue the SQL statement.
5. Call `PDOStatement::fetch(PDO::FETCH_BOUND)` to retrieve the next row in the result set, binding the column output into the PHP variables you associated with the `PDOStatement::bindColumn()` method.

Example:

```
$stmt = $conn->prepare("SELECT name, picture FROM animal_pictures");
$picture = fopen("/opt/albums/spook/grooming.jpg", "wb");
$stmt->bindColumn($stmt, 1, $nickname, PDO::PARAM_STR, 32);
$stmt->bindColumn($stmt, 2, $picture, PDO::PARAM_LOB);
$stmt->execute();
$stmt->fetch(PDO::FETCH_BOUND);
```

Related tasks:

- "Fetching rows from result sets in PHP (PDO)" on page 29

Related reference:
- "PDO::prepare - Prepares a statement for execution and returns a statement object" on page 107
- "PDOStatement::bindParam - Binds a parameter to the specified variable name" on page 113
- "PDOStatement::execute - Executes a prepared statement" on page 118
- "PDOStatement::fetch - Fetches the next row from a result set" on page 119

Managing transactions in PHP (PDO)

By default, PDO opens every connection in autocommit mode. Autocommit mode helps prevent locking escalation issues that can impede the performance of highly scalable Web applications. In some scripts, however, you might need to roll back a transaction containing one or more SQL statements. PDO enables you to exert fine-grained control over your transactions.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO extension.

Procedure:

To begin a transaction:

1. Call `PDO::beginTransaction()` to begin a new transaction.
2. Issue one or more SQL statements within the scope of the database transaction using the connection resource for which transactions have been enabled.
3. Commit or rollback the transaction:
   - To commit the transaction, call `PDO::commit()`.
   - To rollback the transaction, call `PDO::rollBack()`.

After you commit or rollback the transaction, PDO automatically resets the database connection to autocommit mode. If you issue SQL statements in a transaction and the script ends without explicitly committing or rolling back the transaction, PDO automatically rolls back any work performed in the transaction.

Example:

```
$conn = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2', array(
  PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
  // PDO::ERRMODE_EXCEPTION means an SQL error throws an exception
try {
  // Issue these SQL statements in a transaction within a try{} block
  $conn->beginTransaction();

  // One or more SQL statements

  $conn->commit();
}
catch (Exception $e) {
  // If something raised an exception in our transaction block of statements,
  // roll back any work performed in the transaction
  print '<p>Unable to complete transaction!</p>';
  $conn->rollBack();
}
```

Related reference:

- "PDO::beginTransaction - Initiates a transaction" on page 101
- "PDO::commit - Commits a transaction" on page 102
- "PDO::rollBack - Rolls back a transaction" on page 110

Handling errors and warnings in PHP (PDO)

Problems occasionally happen when you attempt to connect to a database or issue an SQL statement. The password for your connection might be incorrect, the table you referred to in a SELECT statement might not exist, or the syntax for an SQL statement might be invalid. You need to code defensively and use the error-handling functions offered by PDO to enable your application to recover gracefully from a problem.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO and PDO_ODBC extensions.

Restrictions:

PDO gives you the option of handling errors as warnings, errors, or exceptions. However, when you create a new PDO connection object, PDO always throws a `PDOException` object if an error occurs. If you do not catch the exception, PHP prints a backtrace of the error information which might expose your database connection credentials, including your user name and password.

Procedure:

- To catch a `PDOException` object and handle the associated error:
  1. Wrap the call to the PDO constructor in a `try` block.
  2. Following the `try` block, include a `catch` block that catches the `PDOException` object.
  3. Retrieve the error message associated with the error by invoking the `Exception::getMessage()` method on the `PDOException` object.
- To retrieve the SQLSTATE associated with a `PDO` or `PDOStatement` object, invoke the `errorCode()` method on the object.
- To retrieve an array of error information associated with a `PDO` or `PDOStatement` object, invoke the `errorInfo()` method on the object. The array contains a string representing the SQLSTATE as the first element, an integer representing the SQL or CLI error code as the second element, and a string containing the full text error message as the third element.

Calling stored procedures with OUT or INOUT parameters in PHP (PDO)

DB2 supports stored procedures with parameters that only accept an input value (IN parameters), that only return an output value (OUT parameters), or that accept an input value and return an output value (INOUT). With the PDO_ODBC extension for PHP you can handle IN parameters like any other parameter marker in an SQL statement. However, the PDO_ODBC extension also enables you to CALL stored procedures with OUT and INOUT parameters and retrieve the output values from those parameters.

Prerequisites:

You must set up the PHP environment on your system and enable the PDO and PDO_ODBC extensions.

Procedure:

To call a stored procedure with OUT or INOUT parameters:

1. Call `PDO::prepare()` to prepare a CALL statement with parameter markers representing the OUT and INOUT parameters.
2. Call `PDOStatement::bindParam()` to bind each parameter marker to the name of the PHP variable that will hold the output value of the parameter after the CALL statement has been issued. For INOUT parameters, the value of the PHP variable is passed as the input value of the parameter when the CALL statement is issued. Set the third parameter for `PDOStatement::bindParam()` to the type of data being bound:

   PDO::PARAM_NULL
   Represents the SQL NULL data type.

   PDO::PARAM_INT
   Represents SQL integer types.

   PDO::PARAM_LOB
   Represents SQL large object types.

   PDO::PARAM_STR
   Represents SQL character data types.
3. For an INOUT parameter, use the bitwise OR operator to append `PDO::PARAM_INPUT_OUTPUT` to the type of data being bound.
4. Set the fourth parameter of `PDOStatement::bindParam()` to the maximum expected length of the output value.

Example:

```
$sql = 'CALL match_animal(?, ?)';
$stmt = $conn->prepare($sql);

$second_name = "Rickety Ride";
$weight = 0;

$stmt->bindParam(1, $second_name, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 32);
$stmt->bindParam(2, $weight, PDO::PARAM_INT, 10);

print "Values of bound parameters _before_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";

$stmt->execute();

print "Values of bound parameters _after_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";
```

Related tasks:

- "Calling stored procedures that return multiple result sets in PHP (PDO)" on page 35

Related reference:

- "PDO::prepare - Prepares a statement for execution and returns a statement object" on page 107

- "PDOStatement::bindParam - Binds a parameter to the specified variable name" on page 113
- "PDOStatement::execute - Executes a prepared statement" on page 118

Calling stored procedures that return multiple result sets in PHP (PDO)

DB2 enables you to create and call stored procedures that return more than one result set. The PDO_ODBC extension for PHP supports this capability through the `nextRowset()` method. You can use this method to fetch rows from different result sets returned by a single call to the same stored procedure.

Prerequisites:

- You must set up the PHP 5.1 or higher environment on your system and enable the PDO and PDO_ODBC extensions.
- You must have a PDOStatement object returned from calling a stored procedure with `PDO::query()` or `PDOStatement::execute()`.

Procedure:

To return multiple result sets from a stored procedure:

1. The first result set is associated with the PDOStatement object returned by the CALL statement. You can fetch rows from the PDOStatement object until no more rows are available in the first result set.
2. Call the `nextRowset()` method of the PDOStatement object to return the next result set. You can fetch rows from the PDOStatement object until no more rows are available in the next result set.

Example:

```
$sql = 'CALL multiple_results()';
$stmt = $conn->query($sql);
do {
   $rows = $stmt->fetchAll(PDO::FETCH_NUM);
   if ($rows) {
       print_r($rows);
   }
} while ($stmt->nextRowset());
```

Related tasks:

- "Calling stored procedures with OUT or INOUT parameters in PHP (PDO)" on page 33

Related reference:

- "PDOStatement::nextRowset - Advances to the next result set in a statement handle associated with multiple result sets" on page 127

Chapter 4. ibm_db2 Extension Reference

Connection functions

db2_autocommit - Returns or sets the AUTOCOMMIT state for a database connection

Syntax:

mixed db2_autocommit(resource *connection*, [**bool** *value*])

Description:

Sets or gets the AUTOCOMMIT behavior of the specified connection resource.

Parameters:

connection

A valid database connection resource variable as returned from `db2_connect()` or `db2_pconnect()`.

value

One of the following constants:

DB2_AUTOCOMMIT_OFF

Turns AUTOCOMMIT off.

DB2_AUTOCOMMIT_ON

Turns AUTOCOMMIT on.

Return Values:

When `db2_autocommit()` receives only the **connection** parameter, it returns the current state of AUTOCOMMIT for the requested connection as an integer value. A value of 0 indicates that AUTOCOMMIT is off, while a value of 1 indicates that AUTOCOMMIT is on.

When db2_autocommit receives both the **connection** parameter and **autocommit** parameter, it attempts to set the AUTOCOMMIT state of the requested connection to the corresponding state. Returns TRUE on success or FALSE on failure.

Examples:

Retrieving the AUTOCOMMIT value for a connection:

In the following example, a connection which has been created with AUTOCOMMIT turned off is tested with the `db2_autocommit()` function.

```
<?php
$options = array('autocommit' => DB2_AUTOCOMMIT_OFF);
$conn = db2_connect($database, $user, $password, $options);
$ac = db2_autocommit($conn);
if ($ac == 0) {
```

```
    print "$ac -- AUTOCOMMIT is off.";
} else {
    print "$ac -- AUTOCOMMIT is on.";
}
?>
```

The preceding example returns the following output:

```
0 -- AUTOCOMMIT is off.
```

Setting the AUTOCOMMIT value for a connection:

In the following example, a connection which was initially created with AUTOCOMMIT turned off has its behavior changed to turn AUTOCOMMIT on.

```
<?php
$options = array('autocommit' => DB2_AUTOCOMMIT_OFF);
$conn = db2_connect($database, $user, $password, $options);

// Turn AUTOCOMMIT on
$rc = db2_autocommit($conn, DB2_AUTOCOMMIT_ON);
if ($rc) {
    print "Turning AUTOCOMMIT on succeeded.\n";
}

// Check AUTOCOMMIT state
$ac = db2_autocommit($conn);
if ($ac == 0) {
    print "$ac -- AUTOCOMMIT is off.";
} else {
    print "$ac -- AUTOCOMMIT is on.";
}
?>
```

The preceding example returns the following output:

```
Turning AUTOCOMMIT on succeeded.
1 -- AUTOCOMMIT is on.
```

Related tasks:

- "Managing transactions in PHP (ibm_db2)" on page 17

Related reference:

- "db2_commit - Commits a transaction" on page 39
- "db2_rollback - Rolls back a transaction" on page 46

db2_close - Closes a database connection

Syntax:

bool db2_close(resource *connection*)

Description:

This function closes a DB2 client connection created with `db2_connect()` and returns the corresponding resources to the database management system.

If you attempt to close a persistent DB2 client connection created with `db2_pconnect()`, the close request is ignored and the persistent DB2 client connection remains available for the next caller.

Parameters:

connection

Specifies an active DB2 client connection.

Return Values:

Returns TRUE on success or FALSE on failure.

Examples:

Closing a connection:

The following example demonstrates a successful attempt to close a connection to an IBM DB2, Cloudscape, or Apache Derby database.

```
<?php
$conn = db2_connect('SAMPLE', 'db2inst1', 'ibmdb2');
$rc = db2_close($conn);
if ($rc) {
    echo "Connection was successfully closed.";
}
?>
```

The preceding example returns the following output:

```
Connection was successfully closed.
```

Related tasks:

- "Connecting to a DB2 database with PHP (ibm_db2)" on page 7

Related reference:

- "db2_connect - Returns a connection to a database" on page 42
- "db2_pconnect - Returns a persistent connection to a database" on page 45

db2_commit - Commits a transaction

Syntax:

bool db2_commit(**resource** *connection*)

Description:

Commits an in-progress transaction on the specified connection resource and begins a new transaction. PHP applications normally default to AUTOCOMMIT mode, so `db2_commit()` is not necessary unless AUTOCOMMIT has been turned off for the connection resource.

Parameters:

connection

A valid database connection resource variable as returned from `db2_connect()`.

Return Values:

Returns TRUE on success or FALSE on failure.

Related tasks:

- "Managing transactions in PHP (ibm_db2)" on page 17

Related reference:

- "db2_rollback - Rolls back a transaction" on page 46
- "db2_autocommit - Returns or sets the AUTOCOMMIT state for a database connection" on page 37

db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt

Syntax:

string db2_conn_error([**resource** *connection*])

Description:

`db2_conn_error()` returns an SQLSTATE value representing the reason the last attempt to connect to a database failed. As `db2_connect()` returns FALSE in the event of a failed connection attempt, you do not pass any parameters to `db2_conn_error()` to retrieve the SQLSTATE value.

If, however, the connection was successful but becomes invalid over time, you can pass the **connection** parameter to retrieve the SQLSTATE value for a specific connection.

To learn what the SQLSTATE value means, you can issue the following command at a DB2 Command Line Processor prompt:

```
db2 '? sqlstate-value'
```

You can also call `db2_conn_errormsg()` to retrieve an explicit error message and the associated SQLCODE value.

Parameters:

connection

A connection resource associated with a connection that initially succeeded, but which over time became invalid.

Return Values:

Returns the SQLSTATE value resulting from a failed connection attempt. Returns an empty string if there is no error associated with the last connection attempt.

Examples:

Retrieving an SQLSTATE value for a failed connection attempt:

The following example demonstrates how to return an SQLSTATE value after deliberately passing invalid parameters to `db2_connect()`.

```
<?php
$conn = db2_connect('badname', 'baduser', 'badpassword');
if (!$conn) {
    print "SQLSTATE value: " . db2_conn_error();
}
?>
```

The preceding example returns the following output:

```
SQLSTATE value: 08001
```

Related tasks:

- "Handling errors and warning messages (ibm_db2)" on page 18

Related reference:

- "db2_conn_errormsg - Returns the last connection error message and SQLCODE value" on page 41
- "db2_stmt_error - Returns a string containing the SQLSTATE returned by an SQL statement" on page 57
- "db2_stmt_errormsg - Returns a string containing the last SQL statement error message" on page 57

db2_conn_errormsg - Returns the last connection error message and SQLCODE value

Syntax:

string db2_conn_errormsg([**resource** *connection*])

Description:

`db2_conn_errormsg()` returns an error message and SQLCODE value representing the reason the last database connection attempt failed. As `db2_connect()` returns FALSE in the event of a failed connection attempt, do not pass any parameters to `db2_conn_errormsg()` to retrieve the associated error message and SQLCODE value.

If, however, the connection was successful but becomes invalid over time, you can pass the **connection** parameter to retrieve the associated error message and SQLCODE value for a specific connection.

Parameters:

connection

A connection resource associated with a connection that initially succeeded, but which over time became invalid.

Return Values:

Returns a string containing the error message and SQLCODE value resulting from a failed connection attempt. If there is no error associated with the last connection attempt, `db2_conn_errormsg()` returns an empty string.

Examples:

Retrieving the error message returned by a failed connection attempt:

The following example demonstrates how to return an error message and SQLCODE value after deliberately passing invalid parameters to db2_connect().

```
<?php
$conn = db2_connect('badname', 'baduser', 'badpassword');
if (!$conn) {
    print db2_conn_errormsg();
}
?>
```

The preceding example returns the following output:

```
[IBM][CLI Driver] SQL1013N  The database alias name
or database name "BADNAME" could not be found.  SQLSTATE=42705
 SQLCODE=-1013
```

Related tasks:

- "Handling errors and warning messages (ibm_db2)" on page 18

Related reference:

- "db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt" on page 40
- "db2_stmt_error - Returns a string containing the SQLSTATE returned by an SQL statement" on page 57
- "db2_stmt_errormsg - Returns a string containing the last SQL statement error message" on page 57

db2_connect - Returns a connection to a database

Syntax:

resource db2_connect(**string** *database*, **string** *username*, **string** *password*, [**array** *options*])

Description:

Creates a new connection to an IBM DB2, IBM Cloudscape, or Apache Derby database.

Parameters:

database

For a cataloged connection to a database, *database* represents the database alias in the DB2 client catalog.

For an uncataloged connection to a database, *database* represents a complete connection string in the following format:

```
DRIVER={IBM DB2 ODBC DRIVER};DATABASE=database;HOSTNAME=hostname;
PORT=port;PROTOCOL=TCPIP;UID=username;PWD=password
```

where the parameters represent the following values:

database

The name of the database.

hostname

The hostname or IP address of the database server.

port

The TCP/IP port on which the database is listening for requests.

username

The username with which you are connecting to the database.

password

The password with which you are connecting to the database.

username

The user name with which you are connecting to the database.

For uncataloged connections, you must pass a NULL value or empty string.

password

The password with which you are connecting to the database.

For uncataloged connections, you must pass a NULL value or empty string.

options

An associative array of connection options that affect the behavior of the connection, where valid array keys include:

autocommit

Passing the **DB2_AUTOCOMMIT_ON** value turns autocommit on for this connection handle.

Passing the **DB2_AUTOCOMMIT_OFF** value turns autocommit off for this connection handle.

Return Values:

Returns a connection handle resource if the connection attempt is successful. If the connection attempt fails, `db2_connect()` returns FALSE.

Examples:

Creating a cataloged connection:

Cataloged connections require you to have previously cataloged the target database through the DB2 Command Line Processor or DB2 Configuration Assistant.

```
<?php
$database = 'SAMPLE';
$user = 'db2inst1';
$password = 'ibmdb2';

$conn = db2_connect($database, $user, $password);

if ($conn) {
    echo "Connection succeeded.";
    db2_close($conn);
}
else {
    echo "Connection failed.";
}
?>
```

The preceding example returns the following output:

```
Connection succeeded.
```

Creating an uncataloged connection:

An uncataloged connection enables you to dynamically connect to a database.

```
<?php
$database = 'SAMPLE';
$user = 'db2inst1';
$password = 'ibmdb2';
$hostname = 'localhost';
$port = 50000;

$conn_string = "DRIVER={IBM DB2 ODBC DRIVER};DATABASE=$database;" .
  "HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;UID=$user;PWD=$password;";
$conn = db2_connect($conn_string, '', '');

if ($conn) {
    echo "Connection succeeded.";
    db2_close($conn);
}
else {
    echo "Connection failed.";
}
?>
```

The preceding example returns the following output:

```
Connection succeeded.
```

Creating a connection with autocommit off by default:

Passing an array of options to `db2_connect()` enables you to modify the default behavior of the connection handle.

```
<?php
$database = 'SAMPLE';
$user = 'db2inst1';
$password = 'ibmdb2';
$options = array('autocommit' => DB2_AUTOCOMMIT_OFF);

$conn = db2_connect($database, $user, $password, $options);

if ($conn) {
    echo "Connection succeeded.\n";
    if (db2_autocommit($conn)) {
         echo "Autocommit is on.\n";
    }
    else {
         echo "Autocommit is off.\n";
    }
    db2_close($conn);
}
else {
    echo "Connection failed.";
}
?>
```

The preceding example returns the following output:

```
Connection succeeded.
Autocommit is off.
```

Related tasks:

- "Connecting to a DB2 database with PHP (ibm_db2)" on page 7

Related reference:

- "db2_close - Closes a database connection" on page 38
- "db2_pconnect - Returns a persistent connection to a database" on page 45

db2_pconnect - Returns a persistent connection to a database

Syntax:

resource db2_pconnect(string *database*, **string** *username*, **string** *password*, [**array** *options*])

Description:

Returns a persistent connection to an IBM DB2, IBM Cloudscape, or Apache Derby database.

Calling `db2_close()` on a persistent connection always returns TRUE, but the underlying DB2 client connection remains open and waiting to serve the next matching `db2_pconnect()` request.

Note that you are strongly urged to only use persistent connections on connections with autocommit turned on. If you attempt to combine transactions with persistent connections, issuing `db2_commit()` or `db2_rollback()` against a persistent connection will affect every persistent connection that is currently using the same underlying DB2 client connection. You may also rapidly experience locking escalation if you do not use autocommit for your persistent connections.

Parameters:

database

The database alias in the DB2 client catalog.

username

The username with which you are connecting to the database.

password

The password with which you are connecting to the database.

options

An associative array of connection options that affect the behavior of the connection, where valid array keys include:

autocommit

Passing the **DB2_AUTOCOMMIT_ON** value turns autocommit on for this connection handle.

Passing the **DB2_AUTOCOMMIT_OFF** value turns autocommit off for this connection handle.

Return Values:

Returns a connection handle resource if the connection attempt is successful. `db2_pconnect()` tries to reuse an existing connection resource that exactly matches the *database*, *username*, and *password* parameters. If the connection attempt fails, `db2_pconnect()` returns FALSE.

Examples:

A db2_pconnect() example:

In the following example, the first call to `db2_pconnect()` returns a new persistent connection resource. The second call to `db2_pconnect()` returns a persistent connection resource that simply reuses the first persistent connection resource.

```
<?php
$database = 'SAMPLE';
$user = 'db2inst1';
$password = 'ibmdb2';

$pconn = db2_pconnect($database, $user, $password);

if ($pconn) {
    echo "Persistent connection succeeded.";
}
else {
    echo "Persistent connection failed.";
}

$pconn2 = db2_pconnect($database, $user, $password);
if ($pconn) {
    echo "Second persistent connection succeeded.";
}
else {
    echo "Second persistent connection failed.";
}
?>
```

The preceding example returns the following output:

```
Persistent connection succeeded.
Second persistent connection succeeded.
```

Related tasks:

- "Connecting to a DB2 database with PHP (ibm_db2)" on page 7

Related reference:

- "db2_connect - Returns a connection to a database" on page 42

db2_rollback - Rolls back a transaction

Syntax:

bool db2_rollback(**resource** *connection*)

Description:

Rolls back an in-progress transaction on the specified connection resource and begins a new transaction. PHP applications normally default to AUTOCOMMIT mode, so `db2_rollback()` normally has no effect unless AUTOCOMMIT has been turned off for the connection resource.

If the specified connection resource is a persistent connection, all transactions in progress for all applications using that persistent connection will be rolled back. For this reason, persistent connections are not recommended for use in applications that require transactions.

Parameters:

connection

A valid database connection resource variable as returned from `db2_connect()` or `db2_pconnect()`.

Return Values:

Returns TRUE on success or FALSE on failure.

Examples:

Rolling back a DELETE statement:

In the following example, we count the number of rows in a table, turn off AUTOCOMMIT mode on a database connection, delete all of the rows in the table and return the count of **0** to prove that the rows have been removed. We then issue `db2_rollback()` and return the updated count of rows in the table to show that the number is the same as before we issued the DELETE statement. The return to the original state of the table demonstrates that the roll back of the transaction succeeded.

```
<?php
$conn = db2_connect($database, $user, $password);

if ($conn) {
    $stmt = db2_exec($conn, "SELECT count(*) FROM animals");
    $res = db2_fetch_array( $stmt );
    echo $res[0] . "\n";

    // Turn AUTOCOMMIT off
    db2_autocommit($conn, DB2_AUTOCOMMIT_OFF);

    // Delete all rows from ANIMALS
    db2_exec($conn, "DELETE FROM animals");

    $stmt = db2_exec($conn, "SELECT count(*) FROM animals");
    $res = db2_fetch_array( $stmt );
    echo $res[0] . "\n";

    // Roll back the DELETE statement
    db2_rollback( $conn );

    $stmt = db2_exec( $conn, "SELECT count(*) FROM animals" );
    $res = db2_fetch_array( $stmt );
    echo $res[0] . "\n";
    db2_close($conn);
}
?>
```

The preceding example returns the following output:

```
7
0
7
```

Related tasks:

- "Managing transactions in PHP (ibm_db2)" on page 17

Related reference:

- "db2_commit - Commits a transaction" on page 39

- "db2_autocommit - Returns or sets the AUTOCOMMIT state for a database connection" on page 37

Statement functions

db2_bind_param - Binds a PHP variable to an SQL statement parameter

Syntax:

bool db2_bind_param(**resource** *stmt*, **int** *parameter-number*, **string** *variable-name*, [**int** *parameter-type*, [**int** *data-type*, [**int** *precision*, [**int** *scale*]]]])

Description:

Binds a PHP variable to an SQL statement parameter in a statement resource returned by `db2_prepare()`. This function gives you more control over the parameter type, data type, precision, and scale for the parameter than simply passing the variable as part of the optional input array to `db2_execute()`.

Parameters:

stmt

A prepared statement returned from `db2_prepare()`.

parameter-number

Specifies the 1-indexed position of the parameter in the prepared statement.

variable-name

A string specifying the name of the PHP variable to bind to the parameter specified by *parameter-number*.

parameter-type

A constant specifying whether the PHP variable should be bound to the SQL parameter as an input parameter (**DB2_PARAM_IN**), an output parameter (**DB2_PARAM_OUT**), or as a parameter that accepts input and returns output (**DB2_PARAM_INOUT**).

data-type

A constant specifying the SQL data type that the PHP variable should be bound as: one of **DB2_BINARY**, **DB2_CHAR**, **DB2_DOUBLE**, or **DB2_LONG** .

precision

Specifies the precision with which the variable should be bound to the database.

scale

Specifies the scale with which the variable should be bound to the database.

Return Values:

Returns TRUE on success or FALSE on failure.

Examples:

Binding PHP variables to a prepared statement:

The SQL statement in the following example uses two input parameters in the WHERE clause. We call `db2_bind_param()` to bind two PHP variables to the corresponding SQL parameters. Notice that the PHP variables do not have to be declared or assigned before the call to `db2_bind_param()`; in the example, **$lower_limit** is assigned a value before the call to `db2_bind_param()`, but **$upper_limit** is assigned a value after the call to `db2_bind_param()`. The variables must be bound and, for parameters that accept input, must have any value assigned, before calling `db2_execute()`.

```
<?php

$sql = 'SELECT name, breed, weight FROM animals
    WHERE weight > ? AND weight < ?';
$conn = db2_connect($database, $user, $password);
$stmt = db2_prepare($conn, $sql);

// We can declare the variable before calling db2_bind_param()
$lower_limit = 1;

db2_bind_param($stmt, 1, "lower_limit", DB2_PARAM_IN);
db2_bind_param($stmt, 2, "upper_limit", DB2_PARAM_IN);

// We can also declare the variable after calling db2_bind_param()
$upper_limit = 15.0;

if (db2_execute($stmt)) {
    while ($row = db2_fetch_array($stmt)) {
        print "{$row[0]}, {$row[1]}, {$row[2]}\n";
    }
}
?>
```

The preceding example returns the following output:

```
Pook, cat, 3.2
Rickety Ride, goat, 9.7
Peaches, dog, 12.3
```

Calling stored procedures with IN and OUT parameters:

The stored procedure match_animal in the following example accepts three different parameters:

1. an input (IN) parameter that accepts the name of the first animal as input
2. an input-output (INOUT) parameter that accepts the name of the second animal as input and returns the string **TRUE** if an animal in the database matches that name
3. an output (OUT) parameter that returns the sum of the weight of the two identified animals

In addition, the stored procedure returns a result set consisting of the animals listed in alphabetic order starting at the animal corresponding to the input value of the first parameter and ending at the animal corresponding to the input value of the second parameter.

```
<?php

$sql = 'CALL match_animal(?, ?, ?)';
$conn = db2_connect($database, $user, $password);
$stmt = db2_prepare($conn, $sql);
```

```
$name = "Peaches";
$second_name = "Rickety Ride";
db2_bind_param($stmt, 1, "name", DB2_PARAM_IN);
db2_bind_param($stmt, 2, "second_name", DB2_PARAM_INOUT);
db2_bind_param($stmt, 3, "weight", DB2_PARAM_OUT);

print "Values of bound parameters _before_ CALL:\n";
print "  1: {$name} 2: {$second_name} 3: {$weight}\n\n";

if (db2_execute($stmt)) {
    print "Values of bound parameters _after_ CALL:\n";
    print "  1: {$name} 2: {$second_name} 3: {$weight}\n\n";

    print "Results:\n";
    while ($row = db2_fetch_array($stmt)) {
        print "  {$row[0]}, {$row[1]}, {$row[2]}\n";
    }
}
?>
```

The preceding example returns the following output:

```
Values of bound parameters _before_ CALL:
  1: Peaches 2: Rickety Ride 3:

Values of bound parameters _after_ CALL:
  1: Peaches 2: TRUE 3: 22

Results:
  Peaches, dog, 12.3
  Pook, cat, 3.2
  Rickety Ride, goat, 9.7
```

Related tasks:

- "Calling stored procedures with OUT or INOUT parameters in PHP (ibm_db2)" on page 20
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

db2_exec - Executes an SQL statement directly

Syntax:

resource db2_exec(**resource** *connection*, **string** *statement*, [**array** *options*])

Description:

Prepares and executes an SQL statement.

If you plan to interpolate PHP variables into the SQL statement, understand that this is one of the more common security exposures. Consider calling `db2_prepare()` to prepare an SQL statement with parameter markers for input values. Then you can call `db2_execute()` to pass in the input values and avoid SQL injection attacks.

If you plan to repeatedly issue the same SQL statement with different parameters, consider calling `db2_prepare()` and `db2_execute()` to enable the database management system to reuse its access plan and increase the efficiency of your database access.

Parameters:

connection

A valid database connection resource variable as returned from `db2_connect()` or `db2_pconnect()`.

statement

An SQL statement. The statement cannot contain any parameter markers.

options

An associative array containing statement options. You can use this parameter to request a scrollable cursor on database management system that support this functionality.

cursor

Passing the **DB2_FORWARD_ONLY** value requests a forward-only cursor for this SQL statement. This is the default type of cursor, and it is supported by all database management system. It is also much faster than a scrollable cursor.

Passing the **DB2_SCROLLABLE** value requests a scrollable cursor for this SQL statement. This type of cursor enables you to fetch rows non-sequentially from the database management system. However, it is only supported by DB2 management system, and is much slower than forward-only cursors.

Return Values:

Returns a statement resource if the SQL statement was issued successfully, or FALSE if the database failed to execute the SQL statement.

Examples:

Creating a table with db2_exec():

The following example uses `db2_exec()` to issue a set of DDL statements in the process of creating a table.

```
<?php
$conn = db2_connect($database, $user, $password);

// Create the test table
$create = 'CREATE TABLE animals (id INTEGER, breed VARCHAR(32),
    name CHAR(16), weight DECIMAL(7,2))';
$result = db2_exec($conn, $create);
if ($result) {
    print "Successfully created the table.\n";
}

// Populate the test table
$animals = array(
    array(0, 'cat', 'Pook', 3.2),
    array(1, 'dog', 'Peaches', 12.3),
    array(2, 'horse', 'Smarty', 350.0),
    array(3, 'gold fish', 'Bubbles', 0.1),
```

```
    array(4, 'budgerigar', 'Gizmo', 0.2),
    array(5, 'goat', 'Rickety Ride', 9.7),
    array(6, 'llama', 'Sweater', 150)
);

foreach ($animals as $animal) {
    $rc = db2_exec($conn, "INSERT INTO animals (id, breed, name, weight)
      VALUES ({$animal[0]}, '{$animal[1]}', '{$animal[2]}', {$animal[3]})");
    if ($rc) {
        print "Insert... ";
    }
}
?>
```

The preceding example returns the following output:

```
Successfully created the table.
Insert... Insert... Insert... Insert... Insert... Insert... Insert...
```

Executing a SELECT statement with a scrollable cursor:

The following example demonstrates how to request a scrollable cursor for an SQL statement issued by `db2_exec()`.

```
<?php
$conn = db2_connect($database, $user, $password);
$sql = "SELECT name FROM animals
    WHERE weight < 10.0
    ORDER BY name";
if ($conn) {
    require_once('prepare.inc');
    $stmt = db2_exec($conn, $sql, array('cursor' => DB2_SCROLLABLE));
    while ($row = db2_fetch_array($stmt)) {
        print "$row[0]\n";
    }
}
?>
```

The preceding example returns the following output:

```
Bubbles
Gizmo
Pook
Rickety Ride
```

Related tasks:

- "Executing a single SQL statement in PHP (ibm_db2)" on page 11
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

db2_execute - Executes a prepared SQL statement

Syntax:

bool db2_execute(**resource** *stmt*, [**array** *parameters*])

Description:

`db2_execute()` executes an SQL statement that was prepared by `db2_prepare()`.

If the SQL statement returns a result set, for example, a SELECT statement or a CALL to a stored procedure that returns one or more result sets, you can retrieve a row as an array from the **stmt** resource using `db2_fetch_assoc()`, `db2_fetch_both()`, or `db2_fetch_array()`. Alternatively, you can use `db2_fetch_row()` to move the result set pointer to the next row and fetch a column at a time from that row with `db2_result()`.

Refer to `db2_prepare()` for a brief discussion of the advantages of using `db2_prepare()` and `db2_execute()` rather than `db2_exec()`.

Parameters:

stmt

A prepared statement returned from `db2_prepare()`.

parameters

An array of input parameters matching any parameter markers contained in the prepared statement.

Return Values:

Returns TRUE on success or FALSE on failure.

Examples:

Preparing and executing an SQL statement with parameter markers:

The following example prepares an INSERT statement that accepts four parameter markers, then iterates over an array of arrays containing the input values to be passed to `db2_execute()`.

```
<?php
$pet = array(0, 'cat', 'Pook', 3.2);

$insert = 'INSERT INTO animals (id, breed, name, weight)
    VALUES (?, ?, ?, ?)';

$stmt = db2_prepare($conn, $insert);
if ($stmt) {
    $result = db2_execute($stmt, $pet);
    if ($result) {
        print "Successfully added new pet.";
    }
}
?>
```

The preceding example returns the following output:

```
Successfully added new pet.
```

Calling a stored procedure with an OUT parameter:

The following example prepares a CALL statement that accepts one parameter marker representing an OUT parameter, binds the PHP variable **$my_pets** to the parameter using db2_bind_param, then issues db2_execute to execute the CALL statement. After the CALL to the stored procedure has been made, the value of **$num_pets** changes to reflect the value returned by the stored procedure for that OUT parameter.

```
<?php
$num_pets = 0;
$res = db2_prepare($conn, "CALL count_my_pets(?)");
$rc = db2_bind_param($res, 1, "num_pets", DB2_PARAM_OUT);
$rc = db2_execute($res);
print "I have $num_pets pets!";
?>
```

The preceding example returns the following output:

```
I have 7 pets!
```

Related tasks:

- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_exec - Executes an SQL statement directly" on page 50
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

db2_free_result - Frees resources associated with a result set

Syntax:

bool db2_free_result(resource *stmt*)

Description:

Frees the system and database resources that are associated with a result set. These resources are freed implicitly when a script finishes, but you can call `db2_free_result()` to explicitly free the result set resources before the end of the script.

Parameters:

stmt

A valid statement resource.

Return Values:

Returns TRUE on success or FALSE on failure.

Related reference:

- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_free_stmt - Frees resources associated with the indicated statement resource" on page 54
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

db2_free_stmt - Frees resources associated with the indicated statement resource

Syntax:

bool db2_free_stmt(resource *stmt*)

Description:

db2_free_stmt - Frees resources associated with the indicated statement resource

Frees the system and database resources that are associated with a statement resource. These resources are freed implicitly when a script finishes, but you can call `db2_free_stmt()` to explicitly free the statement resources before the end of the script.

Parameters:

stmt

A valid statement resource.

Return Values:

Returns TRUE on success or FALSE on failure.

Related reference:

- "db2_execute - Executes a prepared SQL statement" on page 52
- "db2_free_result - Frees resources associated with a result set" on page 54
- "db2_prepare - Prepares an SQL statement to be executed" on page 55

db2_prepare - Prepares an SQL statement to be executed

Syntax:

resource db2_prepare(**resource** *connection*, **string** *statement*, [**array** *options*])

Description:

db2_prepare creates a prepared SQL statement which can include 0 or more parameter markers (**?** characters) representing parameters for input, output, or input/output. You can pass parameters to the prepared statement using `db2_bind_param()`, or for input values only, as an array passed to `db2_execute()`.

There are three main advantages to using prepared statements in your application:

1. Performance: when you prepare a statement, the database management system creates an optimized access plan for retrieving data with that statement. Subsequently issuing the prepared statement with `db2_execute()` enables the statements to reuse that access plan and avoids the overhead of dynamically creating a new access plan for every statement you issue.
2. Security: when you prepare a statement, you can include parameter markers for input values. When you execute a prepared statement with input values for placeholders, the database management system checks each input value to ensure that the type matches the column definition or parameter definition.
3. Advanced functionality: Parameter markers not only enable you to pass input values to prepared SQL statements, they also enable you to retrieve OUT and INOUT parameters from stored procedures using `db2_bind_param()`.

Parameters:

connection

A valid database connection resource variable as returned from `db2_connect()` or `db2_pconnect()`.

statement

An SQL statement, optionally containing one or more parameter markers.

options

An associative array containing statement options. You can use this parameter to request a scrollable cursor on database management systems that support this functionality.

cursor

Passing the **DB2_FORWARD_ONLY** value requests a forward-only cursor for this SQL statement. This is the default type of cursor, and it is supported by all database database management systems. It is also much faster than a scrollable cursor.

Passing the **DB2_SCROLLABLE** value requests a scrollable cursor for this SQL statement. This type of cursor enables you to fetch rows non-sequentially from the database management system. However, it is only supported by DB2 database management systems, and is much slower than forward-only cursors.

Return Values:

Returns a statement resource if the SQL statement was successfully parsed and prepared by the database management system. Returns FALSE if the database management system returned an error. You can determine which error was returned by calling `db2_stmt_error()` or `db2_stmt_errormsg()`.

Examples:

Preparing and executing an SQL statement with parameter markers:

The following example prepares an INSERT statement that accepts four parameter markers, then iterates over an array of arrays containing the input values to be passed to `db2_execute()`.

```
<?php
$animals = array(
    array(0, 'cat', 'Pook', 3.2),
    array(1, 'dog', 'Peaches', 12.3),
    array(2, 'horse', 'Smarty', 350.0),
);

$insert = 'INSERT INTO animals (id, breed, name, weight)
    VALUES (?, ?, ?, ?)';
$stmt = db2_prepare($conn, $insert);
if ($stmt) {
    foreach ($animals as $animal) {
        $result = db2_execute($stmt, $animal);
    }
}
?>
```

Related tasks:

- "Fetching rows from result sets in PHP (ibm_db2)" on page 16
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_bind_param - Binds a PHP variable to an SQL statement parameter" on page 48
- "db2_execute - Executes a prepared SQL statement" on page 52

db2_stmt_error - Returns a string containing the SQLSTATE returned by an SQL statement

Syntax:

string db2_stmt_error([**resource** *stmt*])

Description:

Returns a string containing the SQLSTATE value returned by an SQL statement.

If you do not pass a statement resource as an argument to `db2_stmt_error()`, the driver returns the SQLSTATE value associated with the last attempt to return a statement resource, for example, from `db2_prepare()` or `db2_exec()`.

To learn what the SQLSTATE value means, you can issue the following command at a DB2 Command Line Processor prompt:

```
db2 '? sqlstate-value'
```

. You can also call `db2_stmt_errormsg()` to retrieve an explicit error message and the associated SQLCODE value.

Parameters:

stmt

A valid statement resource.

Return Values:

Returns a string containing an SQLSTATE value.

Related tasks:

- "Handling errors and warning messages (ibm_db2)" on page 18

Related reference:

- "db2_stmt_errormsg - Returns a string containing the last SQL statement error message" on page 57
- "db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt" on page 40
- "db2_conn_errormsg - Returns the last connection error message and SQLCODE value" on page 41

db2_stmt_errormsg - Returns a string containing the last SQL statement error message

Syntax:

string db2_stmt_errormsg([**resource** *stmt*])

db2_stmt_errormsg - Returns a string containing the last SQL statement error message

Description:

Returns a string containing the last SQL statement error message.

If you do not pass a statement resource as an argument to `db2_stmt_errormsg()`, the driver returns the error message associated with the last attempt to return a statement resource, for example, from `db2_prepare()` or `db2_exec()`.

Parameters:

stmt

A valid statement resource.

Return Values:

Returns a string containing the error message and SQLCODE value for the last error that occurred issuing an SQL statement.

Related tasks:

- "Handling errors and warning messages (ibm_db2)" on page 18

Related reference:

- "db2_conn_error - Returns a string containing the SQLSTATE returned by the last connection attempt" on page 40
- "db2_conn_errormsg - Returns the last connection error message and SQLCODE value" on page 41
- "db2_stmt_error - Returns a string containing the SQLSTATE returned by an SQL statement" on page 57

Fetch functions

db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set

Syntax:

array db2_fetch_array(**resource** *stmt*, [**int** *row_number*])

Description:

Returns an array, indexed by column position, representing a row in a result set. The columns are 0-indexed.

Parameters:

stmt

A valid **stmt** resource containing a result set.

row_number

Requests a specific 1-indexed row from the result set. Passing this parameter results in a PHP warning if the result set uses a forward-only cursor.

Return Values:

db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set

Returns a 0-indexed array with column values indexed by the column position representing the next or requested row in the result set. Returns FALSE if there are no rows left in the result set, or if the row requested by **row_number** does not exist in the result set.

Examples:

Iterating through a forward-only cursor:

If you call `db2_fetch_array()` without a specific row number, it automatically retrieves the next row in the result set.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$stmt = db2_prepare($conn, $sql);
$result = db2_execute($stmt);

while ($row = db2_fetch_array($stmt)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row[0], $row[1], $row[2], $row[3]);
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                               3.20
5     Rickety Ride    goat                              9.70
2     Smarty          horse                           350.00
```

Retrieving specific rows with db2_fetch_array() from a scrollable cursor:

If your result set uses a scrollable cursor, you can call `db2_fetch_array()` with a specific row number. The following example retrieves every other row in the result set, starting with the second row.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$result = db2_exec($stmt, $sql, array('cursor' => DB2_SCROLLABLE));

$i=2;
while ($row = db2_fetch_array($result, $i)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row[0], $row[1], $row[2], $row[3]);
    $i = $i + 2;
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                               3.20
5     Rickety Ride    goat                              9.70
2     Smarty          horse                           350.00
```

Related tasks:

- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

Related reference:

- "db2_fetch_assoc - Returns an array, indexed by column name, representing a row in a result set" on page 60

- "db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set" on page 61
- "db2_fetch_object - Returns an object with properties representing columns in the fetched row" on page 63

db2_fetch_assoc - Returns an array, indexed by column name, representing a row in a result set

Syntax:

array db2_fetch_assoc(**resource** *stmt*, [**int** *row_number*])

Description:

Returns an array, indexed by column name, representing a row in a result set.

Parameters:

stmt

A valid **stmt** resource containing a result set.

row_number

Requests a specific 1-indexed row from the result set. Passing this parameter results in a PHP warning if the result set uses a forward-only cursor.

Return Values:

Returns an associative array with column values indexed by the column name representing the next or requested row in the result set. Returns FALSE if there are no rows left in the result set, or if the row requested by **row_number** does not exist in the result set.

Examples:

Iterating through a forward-only cursor:

If you call `db2_fetch_assoc()` without a specific row number, it automatically retrieves the next row in the result set.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$stmt = db2_prepare($conn, $sql);
$result = db2_execute($stmt);

while ($row = db2_fetch_assoc($stmt)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row['ID'], $row['NAME'], $row['BREED'], $row['WEIGHT']);
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                                3.20
5     Rickety Ride    goat                               9.70
2     Smarty          horse                            350.00
```

Retrieving specific rows with db2_fetch_assoc() from a scrollable cursor:

If your result set uses a scrollable cursor, you can call `db2_fetch_assoc()` with a specific row number. The following example retrieves every other row in the result set, starting with the second row.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$result = db2_exec($stmt, $sql, array('cursor' => DB2_SCROLLABLE));

$i=2;
while ($row = db2_fetch_assoc($result, $i)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row['ID'], $row['NAME'], $row['BREED'], $row['WEIGHT']);
    $i = $i + 2;
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                               3.20
5     Rickety Ride    goat                              9.70
2     Smarty          horse                           350.00
```

Related tasks:

- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

Related reference:

- "db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set" on page 58
- "db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set" on page 61
- "db2_fetch_object - Returns an object with properties representing columns in the fetched row" on page 63

db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set

Syntax:

array db2_fetch_both(**resource** *stmt*, [**int** *row_number*])

Description:

Returns an array, indexed by both column name and position, representing a row in a result set. Note that the row returned by `db2_fetch_both()` requires more memory than the single-indexed arrays returned by `db2_fetch_assoc()` or `db2_fetch_array()`.

Parameters:

stmt

A valid **stmt** resource containing a result set.

row_number

Requests a specific 1-indexed row from the result set. Passing this parameter results in a PHP warning if the result set uses a forward-only cursor.

db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set

Return Values:

Returns an associative array with column values indexed by both the column name and 0-indexed column number. The array represents the next or requested row in the result set. Returns FALSE if there are no rows left in the result set, or if the row requested by **row_number** does not exist in the result set.

Examples:

Iterating through a forward-only cursor:

If you call `db2_fetch_both()` without a specific row number, it automatically retrieves the next row in the result set. The following example accesses columns in the returned array by both column name and by numeric index.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$stmt = db2_prepare($conn, $sql);
$result = db2_execute($stmt);

while ($row = db2_fetch_both($stmt)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row['ID'], $row[0], $row['BREED'], $row[3]);
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                               3.20
5     Rickety Ride    goat                              9.70
2     Smarty          horse                           350.00
```

Retrieving specific rows with db2_fetch_both() from a scrollable cursor:

If your result set uses a scrollable cursor, you can call `db2_fetch_both()` with a specific row number. The following example retrieves every other row in the result set, starting with the second row.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$result = db2_exec($stmt, $sql, array('cursor' => DB2_SCROLLABLE));

$i=2;
while ($row = db2_fetch_both($result, $i)) {
    printf ("%-5d %-16s %-32s %10s\n",
        $row[0], $row['NAME'], $row[2], $row['WEIGHT']);
    $i = $i + 2;
}
?>
```

The preceding example returns the following output:

```
0     Pook            cat                               3.20
5     Rickety Ride    goat                              9.70
2     Smarty          horse                           350.00
```

Related tasks:

- "Fetching large objects in PHP (ibm_db2)" on page 17
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

Related reference:

- "db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set" on page 58
- "db2_fetch_assoc - Returns an array, indexed by column name, representing a row in a result set" on page 60
- "db2_fetch_object - Returns an object with properties representing columns in the fetched row" on page 63

db2_fetch_object - Returns an object with properties representing columns in the fetched row

Syntax:

object db2_fetch_object(resource *stmt*, [**int** *row_number*])

Description:

Returns an object in which each property represents a column returned in the row fetched from a result set.

Parameters:

stmt

A valid **stmt** resource containing a result set.

row_number

Requests a specific 1-indexed row from the result set. Passing this parameter results in a PHP warning if the result set uses a forward-only cursor.

Return Values:

Returns an object representing a single row in the result set. The properties of the object map to the names of the columns in the result set.

The IBM DB2, Cloudscape, and Apache Derby database management systems typically fold column names to upper-case, so the object properties will reflect that case.

If your SELECT statement calls a scalar function to modify the value of a column, the database management systems return the column number as the name of the column in the result set. If you prefer a more descriptive column name and object property, you can use the AS clause to assign a name to the column in the result set.

Returns FALSE if no row was retrieved.

Examples:

A db2_fetch_object() example:

The following example issues a SELECT statement with a scalar function, RTRIM(), that removes whitespace from the end of the column. Rather than creating an object with the properties ″BREED″ and ″2″, we use the AS clause in the SELECT statement to assign the name ″name″ to the modified column. The database

management system folds the column names to upper-case, resulting in an object with the properties ″BREED″ and ″NAME″.

```
<?php
$conn = db2_connect($database, $user, $password);

$sql = "SELECT breed, RTRIM(name) AS name
    FROM animals
    WHERE id = ?";

if ($conn) {
    $stmt = db2_prepare($conn, $sql);
    db2_execute($stmt, array(0));

    while ($pet = db2_fetch_object($stmt)) {
        echo "Come here, {$pet->NAME}, my little {$pet->BREED}!";
    }
    db2_close($conn);
}
?>
```

The preceding example returns the following output:

```
Come here, Pook, my little cat!
```

Related tasks:

- “Fetching large objects in PHP (ibm_db2)” on page 17
- “Fetching rows from result sets in PHP (ibm_db2)” on page 16

Related reference:

- “db2_fetch_array - Returns an array, indexed by column position, representing a row in a result set” on page 58
- “db2_fetch_assoc - Returns an array, indexed by column name, representing a row in a result set” on page 60
- “db2_fetch_both - Returns an array, indexed by both column name and position, representing a row in a result set” on page 61

db2_fetch_row - Sets the result set pointer to the next row or requested row

Syntax:

bool db2_fetch_row(**resource** *stmt*, [**int** *row_number*])

Description:

Use `db2_fetch_row()` to iterate through a result set, or to point to a specific row in a result set if you requested a scrollable cursor.

To retrieve individual fields from the result set, call the `db2_result()` function.

Rather than calling `db2_fetch_row()` and `db2_result()`, most applications will call one of `db2_fetch_assoc()`, `db2_fetch_both()`, or `db2_fetch_array()` to advance the result set pointer and return a complete row as an array.

Parameters:

stmt

A valid **stmt** resource.

row_number

With scrollable cursors, you can request a specific row number in the result set. Row numbering is 1-indexed.

Return Values:

Returns TRUE if the requested row exists in the result set. Returns FALSE if the requested row does not exist in the result set.

Examples:

Iterating through a result set:

The following example demonstrates how to iterate through a result set with `db2_fetch_row()` and retrieve columns from the result set with `db2_result()`.

```
<?php
$sql = 'SELECT name, breed FROM animals WHERE weight < ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array(10));
while (db2_fetch_row($stmt)) {
    $name = db2_result($stmt, 0);
    $breed = db2_result($stmt, 1);
    print "$name $breed";
}
?>
```

The preceding example returns the following output:

```
cat Pook
gold fish Bubbles
budgerigar Gizmo
goat Rickety Ride
```

Related tasks:

- "Fetching columns from result sets in PHP (ibm_db2)" on page 15

Related reference:

- "db2_result - Returns a single column from a row in the result set" on page 67

db2_next_result - Requests the next result set from a stored procedure

Syntax:

resource db2_next_result(resource *stmt*)

Description:

A stored procedure can return zero or more result sets. While you handle the first result set in exactly the same way you would handle the results returned by a simple SELECT statement, to fetch the second and subsequent result sets from a stored procedure you must call the `db2_next_result()` function and return the result to a uniquely named PHP variable.

Parameters:

stmt

A prepared statement returned from db2_exec() or db2_execute().

Return Values:

Returns a new statement resource containing the next result set if the stored procedure returned another result set. Returns FALSE if the stored procedure did not return another result set.

Examples:

Calling a stored procedure that returns multiple result sets:

In the following example, we call a stored procedure that returns three result sets. The first result set is fetched directly from the same statement resource on which we invoked the CALL statement, while the second and third result sets are fetched from statement resources returned from our calls to the db2_next_result() function.

```
<?php
$conn = db2_connect($database, $user, $password);

if ($conn) {
  $stmt = db2_exec($conn, 'CALL multiResults()');

  print "Fetching first result set\n";
  while ($row = db2_fetch_array($stmt)) {
    var_dump($row);
  }

  print "\nFetching second result set\n";
  $res = db2_next_result($stmt);
  if ($res) {
    while ($row = db2_fetch_array($res)) {
      var_dump($row);
    }
  }

  print "\nFetching third result set\n";
  $res2 = db2_next_result($stmt);
  if ($res2) {
    while ($row = db2_fetch_array($res2)) {
      var_dump($row);
    }
  }

  db2_close($conn);
}
?>
```

The preceding example returns the following output:

```
Fetching first result set
array(2) {
  [0]=>
  string(16) "Bubbles         "
  [1]=>
  int(3)
}
array(2) {
  [0]=>
  string(16) "Gizmo           "
  [1]=>
  int(4)
```

```
}

Fetching second result set
array(4) {
  [0]=>
  string(16) "Sweater         "
  [1]=>
  int(6)
  [2]=>
  string(5) "llama"
  [3]=>
  string(6) "150.00"
}
array(4) {
  [0]=>
  string(16) "Smarty          "
  [1]=>
  int(2)
  [2]=>
  string(5) "horse"
  [3]=>
  string(6) "350.00"
}

Fetching third result set
array(1) {
  [0]=>
  string(16) "Bubbles         "
}
array(1) {
  [0]=>
  string(16) "Gizmo           "
}
```

Related tasks:

- "Calling stored procedures that return multiple result sets in PHP (ibm_db2)" on page 21
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

db2_result - Returns a single column from a row in the result set

Syntax:

mixed db2_result(**resource** *stmt*, **mixed** *column*)

Description:

Use `db2_result()` to return the value of a specified column in the current row of a result set. You must call `db2_fetch_row()` before calling `db2_result()` to set the location of the result set pointer.

Parameters:

stmt

A valid **stmt** resource.

column

Either an integer mapping to the 0-indexed field in the result set, or a string matching the name of the column.

db2_result - Returns a single column from a row in the result set

Return Values:

Returns the value of the requested field if the field exists in the result set. Returns NULL if the field does not exist, and issues a warning.

Examples:

A db2_result() example:

The following example demonstrates how to iterate through a result set with `db2_fetch_row()` and retrieve columns from the result set with `db2_result()`.

```
<?php
$sql = 'SELECT name, breed FROM animals WHERE weight < ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array(10));
while (db2_fetch_row($stmt)) {
    $name = db2_result($stmt, 0);
    $breed = db2_result($stmt, 'BREED');
    print "$name $breed";
}
?>
```

The preceding example returns the following output:

```
cat Pook
gold fish Bubbles
budgerigar Gizmo
goat Rickety Ride
```

Related tasks:

- "Fetching columns from result sets in PHP (ibm_db2)" on page 15
- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

Related reference:

- "db2_fetch_row - Sets the result set pointer to the next row or requested row" on page 64

Metadata functions

Database metadata functions

db2_client_info - Returns an object with properties that describe the DB2 database client

Syntax:

object db2_client_info(resource *connection*)

Description:

This function returns an object with read-only properties that return information about the DB2 database client. The following table lists the DB2 client properties:

Table 1. DB2 client properties

Property name	Return type	Description
APPL_CODEPAGE	int	The application code page.

Table 1. DB2 client properties (continued)

Property name	Return type	Description
CONN_CODEPAGE	int	The code page for the current connection.
DATA_SOURCE_NAME	string	The data source name (DSN) used to create the current connection to the database.
DRIVER_NAME	string	The name of the library that implements the DB2 Call Level Interface specification.
DRIVER_ODBC_VER	string	The version of ODBC that the DB2 client supports. This returns a string ″MM.mm″ where *MM* is the major version and *mm* is the minor version. The DB2 client always returns ″03.51″.
DRIVER_VER	string	The version of the client, in the form of a string ″MM.mm.uuuu″ where *MM* is the major version, *mm* is the minor version, and *uuuu* is the update. For example, ″08.02.0001″ represents major version 8, minor version 2, update 1.
ODBC_SQL_CONFORMANCE	string	The level of ODBC SQL grammar supported by the client: **MINIMUM** Supports the minimum ODBC SQL grammar. **CORE** Supports the core ODBC SQL grammar. **EXTENDED** Supports extended ODBC SQL grammar.
ODBC_VER	string	The version of ODBC that the ODBC driver manager supports. This returns a string ″MM.mm.rrrr″ where *MM* is the major version, *mm* is the minor version, and *rrrr* is the release. The DB2 client always returns ″03.01.0000″.

Parameters:

connection

Specifies an active DB2 client connection.

Return Values:

Returns an object on a successful call. Returns FALSE on failure.

Examples:

A db2_client_info example:

To retrieve information about the client, you must pass a valid database connection resource to db2_client_info().

```
<?php
$conn = db2_connect( 'SAMPLE', 'db2inst1', 'ibmdb2' );
$client = db2_client_info( $conn );

if ($client) {
    echo "DRIVER_NAME: ";   var_dump( $client->DRIVER_NAME );
    echo "DRIVER_VER: ";   var_dump( $client->DRIVER_VER );
    echo "DATA_SOURCE_NAME: ";   var_dump( $client->DATA_SOURCE_NAME );
    echo "DRIVER_ODBC_VER: ";   var_dump( $client->DRIVER_ODBC_VER );
    echo "ODBC_VER: ";    var_dump( $client->ODBC_VER );
    echo "ODBC_SQL_CONFORMANCE: ";  var_dump( $client->ODBC_SQL_CONFORMANCE );
    echo "APPL_CODEPAGE: ";   var_dump( $client->APPL_CODEPAGE );
    echo "CONN_CODEPAGE: ";   var_dump( $client->CONN_CODEPAGE );
}
else {
    echo "Error retrieving client information.
     Perhaps your database connection was invalid.";
}
db2_close($conn);

?>
```

The preceding example returns the following output:

```
DRIVER_NAME: string(8) "libdb2.a"
DRIVER_VER: string(10) "08.02.0001"
DATA_SOURCE_NAME: string(6) "SAMPLE"
DRIVER_ODBC_VER: string(5) "03.51"
ODBC_VER: string(10) "03.01.0000"
ODBC_SQL_CONFORMANCE: string(8) "EXTENDED"
APPL_CODEPAGE: int(819)
CONN_CODEPAGE: int(819)
```

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_server_info - Returns an object with properties that describe the DB2 database management system" on page 79

db2_column_privileges - Returns a result set listing the columns and associated privileges for a table

Syntax:

resource db2_column_privileges(**resource** *connection*, [**string** *qualifier*, [**string** *schema*, [**string** *table-name*, [**string** *column-name*]]]])

Description:

Returns a result set listing the columns and associated privileges for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. To match all schemas, pass NULL or an empty string.

table-name

The name of the table or view. To match all tables in the database, pass NULL or an empty string.

column-name

The name of the column. To match all columns in the table, pass NULL or an empty string.

Return Values:

Returns a statement resource with a result set containing rows describing the column privileges for columns matching the specified parameters. The rows are composed of the following columns:

Column name	Description
TABLE_CAT	Name of the catalog. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema.
TABLE_NAME	Name of the table or view.
COLUMN_NAME	Name of the column.
GRANTOR	Authorization ID of the user who granted the privilege.
GRANTEE	Authorization ID of the user to whom the privilege was granted.
PRIVILEGE	The privilege for the column.
IS_GRANTABLE	Whether the GRANTEE is permitted to grant this privilege to other users.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_columns - Returns a result set listing the columns and associated metadata for a table" on page 71

db2_columns - Returns a result set listing the columns and associated metadata for a table

Syntax:

resource db2_columns(**resource** *connection*, [**string** *qualifier*, [**string** *schema*, [**string** *table-name*, [**string** *column-name*]]]])

db2_columns - Returns a result set listing the columns and associated metadata for a table

Description:

Returns a result set listing the columns and associated metadata for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. To match all schemas, pass '%'.

table-name

The name of the table or view. To match all tables in the database, pass NULL or an empty string.

column-name

The name of the column. To match all columns in the table, pass NULL or an empty string.

Return Values:

Returns a statement resource with a result set containing rows describing the columns matching the specified parameters. The rows are composed of the following columns:

Column name	Description
TABLE_CAT	Name of the catalog. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema.
TABLE_NAME	Name of the table or view.
COLUMN_NAME	Name of the column.
DATA_TYPE	The SQL data type for the column represented as an integer value.
TYPE_NAME	A string representing the data type for the column.
COLUMN_SIZE	An integer value representing the size of the column.
BUFFER_LENGTH	Maximum number of bytes necessary to store data from this column.
DECIMAL_DIGITS	The scale of the column, or NULL where scale is not applicable.
NUM_PREC_RADIX	An integer value of either **10** (representing an exact numeric data type), **2** (representing an approximate numeric data type), or NULL (representing a data type for which radix is not applicable).
NULLABLE	An integer value representing whether the column is nullable or not.

Column name	Description
REMARKS	Description of the column.
COLUMN_DEF	Default value for the column.
SQL_DATA_TYPE	An integer value representing the size of the column.
SQL_DATETIME_SUB	Returns an integer value representing a datetime subtype code, or NULL for SQL data types to which this does not apply.
CHAR_OCTET_LENGTH	Maximum length in octets for a character data type column, which matches COLUMN_SIZE for single-byte character set data, or NULL for non-character data types.
ORDINAL_POSITION	The 1-indexed position of the column in the table.
IS_NULLABLE	A string value where 'YES' means that the column is nullable and 'NO' means that the column is not nullable.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_column_privileges - Returns a result set listing the columns and associated privileges for a table" on page 70

db2_foreign_keys - Returns a result set listing the foreign keys for a table

Syntax:

resource db2_foreign_keys(**resource** *connection*, **string** *qualifier*, **string** *schema*, **string** *table-name*)

Description:

Returns a result set listing the foreign keys for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. If **schema** is NULL, `db2_foreign_keys()` matches the schema for the current connection.

table-name

The name of the table.

db2_foreign_keys - Returns a result set listing the foreign keys for a table

Return Values:

Returns a statement resource with a result set containing rows describing the foreign keys for the specified table. The result set is composed of the following columns:

Column name	Description
PKTABLE_CAT	Name of the catalog for the table containing the primary key. The value is NULL if this table does not have catalogs.
PKTABLE_SCHEM	Name of the schema for the table containing the primary key.
PKTABLE_NAME	Name of the table containing the primary key.
PKCOLUMN_NAME	Name of the column containing the primary key.
FKTABLE_CAT	Name of the catalog for the table containing the foreign key. The value is NULL if this table does not have catalogs.
FKTABLE_SCHEM	Name of the schema for the table containing the foreign key.
FKTABLE_NAME	Name of the table containing the foreign key.
FKCOLUMN_NAME	Name of the column containing the foreign key.
KEY_SEQ	1-indexed position of the column in the key.
UPDATE_RULE	Integer value representing the action applied to the foreign key when the SQL operation is UPDATE.
DELETE_RULE	Integer value representing the action applied to the foreign key when the SQL operation is DELETE.
FK_NAME	The name of the foreign key.
PK_NAME	The name of the primary key.
DEFERRABILITY	An integer value representing whether the foreign key deferrability is SQL_INITIALLY_DEFERRED, SQL_INITIALLY_IMMEDIATE, or SQL_NOT_DEFERRABLE.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_primary_keys - Returns a result set listing primary keys for a table" on page 74

db2_primary_keys - Returns a result set listing primary keys for a table

Syntax:

db2_primary_keys - Returns a result set listing primary keys for a table

resource db2_primary_keys(**resource** *connection*, **string** *qualifier*, **string** *schema*, **string** *table-name*)

Description:

Returns a result set listing the primary keys for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. If **schema** is NULL, `db2_primary_keys()` matches the schema for the current connection.

table-name

The name of the table.

Return Values:

Returns a statement resource with a result set containing rows describing the primary keys for the specified table. The result set is composed of the following columns:

Table 2. db2_primary_keys() result set columns

Column name	Description
TABLE_CAT	Name of the catalog for the table containing the primary key. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema for the table containing the primary key.
TABLE_NAME	Name of the table containing the primary key.
COLUMN_NAME	Name of the column containing the primary key.
KEY_SEQ	1-indexed position of the column in the key.
PK_NAME	The name of the primary key.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_foreign_keys - Returns a result set listing the foreign keys for a table" on page 73

db2_procedure_columns - Returns a result set listing stored procedure parameters

Syntax:

resource db2_procedure_columns(resource *connection*, **string** *qualifier*, **string** *schema*, **string** *procedure*, **string** *parameter*)

Description:

Returns a result set listing the parameters for one or more stored procedures.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the procedures. This parameter accepts a search pattern containing _ and % as wildcards.

procedure

The name of the procedure. This parameter accepts a search pattern containing _ and % as wildcards.

parameter

The name of the parameter. This parameter accepts a search pattern containing _ and % as wildcards. If this parameter is NULL, all parameters for the specified stored procedures are returned.

Return Values:

Returns a statement resource with a result set containing rows describing the parameters for the stored procedures matching the specified parameters. The rows are composed of the following columns:

Table 3. db2_procedure_columns() result set columns

Column name	Description
PROCEDURE_CAT	The catalog that contains the procedure. The value is NULL if this table does not have catalogs.
PROCEDURE_SCHEM	Name of the schema that contains the stored procedure.
PROCEDURE_NAME	Name of the procedure.
COLUMN_NAME	Name of the parameter.

Table 3. db2_procedure_columns() result set columns (continued)

Column name	Description
COLUMN_TYPE	An integer value representing the type of the parameter: **Return value** **Parameter type** **1 (SQL_PARAM_INPUT)** Input (IN) parameter. **2 (SQL_PARAM_INPUT_OUTPUT)** Input/output (INOUT) parameter. **3 (SQL_PARAM_OUTPUT)** Output (OUT) parameter.
DATA_TYPE	The SQL data type for the parameter represented as an integer value.
TYPE_NAME	A string representing the data type for the parameter.
COLUMN_SIZE	An integer value representing the size of the parameter.
BUFFER_LENGTH	Maximum number of bytes necessary to store data for this parameter.
DECIMAL_DIGITS	The scale of the parameter, or NULL where scale is not applicable.
NUM_PREC_RADIX	An integer value of either **10** (representing an exact numeric data type), **2** (representing an approximate numeric data type), or NULL (representing a data type for which radix is not applicable).
NULLABLE	An integer value representing whether the parameter is nullable or not.
REMARKS	Description of the parameter.
COLUMN_DEF	Default value for the parameter.
SQL_DATA_TYPE	An integer value representing the size of the parameter.
SQL_DATETIME_SUB	Returns an integer value representing a datetime subtype code, or NULL for SQL data types to which this does not apply.
CHAR_OCTET_LENGTH	Maximum length in octets for a character data type parameter, which matches COLUMN_SIZE for single-byte character set data, or NULL for non-character data types.
ORDINAL_POSITION	The 1-indexed position of the parameter in the CALL statement.
IS_NULLABLE	A string value where 'YES' means that the parameter accepts or returns NULL values and 'NO' means that the parameter does not accept or return NULL values.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_procedures - Returns a result set listing the stored procedures registered in a database" on page 78

db2_procedures - Returns a result set listing the stored procedures registered in a database

Syntax:

resource db2_procedures(**resource** *connection*, **string** *qualifier*, **string** *schema*, **string** *procedure*)

Description:

Returns a result set listing the stored procedures registered in a database.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the procedures. This parameter accepts a search pattern containing _ and % as wildcards.

procedure

The name of the procedure. This parameter accepts a search pattern containing _ and % as wildcards.

Return Values:

Returns a statement resource with a result set containing rows describing the stored procedures matching the specified parameters. The rows are composed of the following columns:

Table 4. db2_procedures() result set columns

Column name	Description
PROCEDURE_CAT	The catalog that contains the procedure. The value is NULL if this table does not have catalogs.
PROCEDURE_SCHEM	Name of the schema that contains the stored procedure.
PROCEDURE_NAME	Name of the procedure.
NUM_INPUT_PARAMS	Number of input (IN) parameters for the stored procedure.
NUM_OUTPUT_PARAMS	Number of output (OUT) parameters for the stored procedure.
NUM_RESULT_SETS	Number of result sets returned by the stored procedure.
REMARKS	Any comments about the stored procedure.

Table 4. db2_procedures() result set columns (continued)

Column name	Description
PROCEDURE_TYPE	Always returns **1**, indicating that the stored procedure does not return a return value.

db2_server_info - Returns an object with properties that describe the DB2 database management system

Syntax:

object db2_server_info(**resource** *connection*)

Description:

This function returns an object with read-only properties that return information about the IBM DB2, Cloudscape, or Apache Derby database management system. The following table lists the database management system properties:

Table 5. Database management system properties

Property name	Return type	Description
DBMS_NAME	string	The name of the database management system to which you are connected. For DB2 management systems this is a combination of **DB2** followed by the operating system on which the database management system is running.
DBMS_VER	string	The version of the database management system, in the form of a string "MM.mm.uuuu" where *MM* is the major version, *mm* is the minor version, and *uuuu* is the update. For example, "08.02.0001" represents major version 8, minor version 2, update 1.
DB_CODEPAGE	int	The code page of the database to which you are connected.
DB_NAME	string	The name of the database to which you are connected.

Table 5. Database management system properties (continued)

Property name	Return type	Description
DFT_ISOLATION	string	The default transaction isolation level supported by the database management system: **UR** Uncommitted read: changes are immediately visible by all concurrent transactions. **CS** Cursor stability: a row read by one transaction can be altered and committed by a second concurrent transaction. **RS** Read stability: a transaction can add or remove rows matching a search condition or a pending transaction. **RR** Repeatable read: data affected by pending transaction is not available to other transactions. **NC** No commit: any changes are visible at the end of a successful operation. Explicit commits and rollbacks are not allowed.
IDENTIFIER_QUOTE_CHAR	string	The character used to delimit an identifier.
INST_NAME	string	The instance on the database management system that contains the database.
ISOLATION_OPTION	array	An array of the isolation options supported by the database management system. The isolation options are described in the DFT_ISOLATION property.
KEYWORDS	array	An array of the keywords reserved by the database management system.

Table 5. Database management system properties (continued)

Property name	Return type	Description
LIKE_ESCAPE_CLAUSE	bool	TRUE if the database management system supports the use of % and _ wildcard characters. FALSE if the database management system does not support these wildcard characters.
MAX_COL_NAME_LEN	int	Maximum length of a column name supported by the database management system, expressed in bytes.
MAX_IDENTIFIER_LEN	int	Maximum length of an SQL identifier supported by the database management system, expressed in characters.
MAX_INDEX_SIZE	int	Maximum size of columns combined in an index supported by the database management system, expressed in bytes.
MAX_PROC_NAME_LEN	int	Maximum length of a procedure name supported by the database management system, expressed in bytes.
MAX_ROW_SIZE	int	Maximum length of a row in a base table supported by the database management system, expressed in bytes.
MAX_SCHEMA_NAME_LEN	int	Maximum length of a schema name supported by the database management system, expressed in bytes.
MAX_STATEMENT_LEN	int	Maximum length of an SQL statement supported by the database management system, expressed in bytes.
MAX_TABLE_NAME_LEN	int	Maximum length of a table name supported by the database management system, expressed in bytes.
NON_NULLABLE_COLUMNS	bool	TRUE if the database management system supports columns that can be defined as NOT NULL, FALSE if the database management system does not support columns defined as NOT NULL.
PROCEDURES	bool	TRUE if the database management system supports the use of the CALL statement to call stored procedures, FALSE if the database management system does not support the CALL statement.

Table 5. Database management system properties (continued)

Property name	Return type	Description
SPECIAL_CHARS	string	A string containing all of the characters other than a-Z, 0-9, and underscore that can be used in an identifier name.
SQL_CONFORMANCE	string	The level of conformance to the ANSI/ISO SQL-92 specification offered by the database management system: **ENTRY** Entry-level SQL-92 compliance. **FIPS127** FIPS-127-2 transitional compliance. **FULL** Full level SQL-92 compliance. **INTERMEDIATE** Intermediate level SQL-92 compliance.

Parameters:

connection

Specifies an active DB2 client connection.

Return Values:

Returns an object on a successful call. Returns FALSE on failure.

Examples:

A db2_server_info() example:

To retrieve information about the database management system, you must pass a valid database connection resource to `db2_server_info()`.

```
<?php

$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');

$server = db2_server_info( $conn );

if ($server) {
    echo "DBMS_NAME: ";                var_dump( $server->DBMS_NAME );
    echo "DBMS_VER: ";                 var_dump( $server->DBMS_VER );
    echo "DB_CODEPAGE: ";              var_dump( $server->DB_CODEPAGE );
    echo "DB_NAME: ";                  var_dump( $server->DB_NAME );
    echo "INST_NAME: ";                var_dump( $server->INST_NAME );
    echo "SPECIAL_CHARS: ";            var_dump( $server->SPECIAL_CHARS );
    echo "KEYWORDS: ";                 var_dump( sizeof($server->KEYWORDS) );
    echo "DFT_ISOLATION: ";            var_dump( $server->DFT_ISOLATION );
    echo "ISOLATION_OPTION: ";
```

```
    $il = '';
    foreach( $server->ISOLATION_OPTION as $opt )
    {
       $il .= $opt." ";
    }
    var_dump( $il );
    echo "SQL_CONFORMANCE: ";           var_dump( $server->SQL_CONFORMANCE );
    echo "PROCEDURES: ";                var_dump( $server->PROCEDURES );
    echo "IDENTIFIER_QUOTE_CHAR: ";     var_dump( $server->IDENTIFIER_QUOTE_CHAR );
    echo "LIKE_ESCAPE_CLAUSE: ";        var_dump( $server->LIKE_ESCAPE_CLAUSE );
    echo "MAX_COL_NAME_LEN: ";          var_dump( $server->MAX_COL_NAME_LEN );
    echo "MAX_ROW_SIZE: ";              var_dump( $server->MAX_ROW_SIZE );
    echo "MAX_IDENTIFIER_LEN: ";        var_dump( $server->MAX_IDENTIFIER_LEN );
    echo "MAX_INDEX_SIZE: ";            var_dump( $server->MAX_INDEX_SIZE );
    echo "MAX_PROC_NAME_LEN: ";         var_dump( $server->MAX_PROC_NAME_LEN );
    echo "MAX_SCHEMA_NAME_LEN: ";       var_dump( $server->MAX_SCHEMA_NAME_LEN );
    echo "MAX_STATEMENT_LEN: ";         var_dump( $server->MAX_STATEMENT_LEN );
    echo "MAX_TABLE_NAME_LEN: ";        var_dump( $server->MAX_TABLE_NAME_LEN );
    echo "NON_NULLABLE_COLUMNS: ";      var_dump( $server->NON_NULLABLE_COLUMNS );

    db2_close($conn);
}
?>
```

The preceding example returns the following output:

```
DBMS_NAME: string(9) "DB2/LINUX"
DBMS_VER: string(10) "08.02.0000"
DB_CODEPAGE: int(1208)
DB_NAME: string(6) "SAMPLE"
INST_NAME: string(8) "db2inst1"
SPECIAL_CHARS: string(2) "@#"
KEYWORDS: int(179)
DFT_ISOLATION: string(2) "CS"
ISOLATION_OPTION: string(12) "UR CS RS RR "
SQL_CONFORMANCE: string(7) "FIPS127"
PROCEDURES: bool(true)
IDENTIFIER_QUOTE_CHAR: string(1) """
LIKE_ESCAPE_CLAUSE: bool(true)
MAX_COL_NAME_LEN: int(30)
MAX_ROW_SIZE: int(32677)
MAX_IDENTIFIER_LEN: int(18)
MAX_INDEX_SIZE: int(1024)
MAX_PROC_NAME_LEN: int(128)
MAX_SCHEMA_NAME_LEN: int(30)
MAX_STATEMENT_LEN: int(2097152)
MAX_TABLE_NAME_LEN: int(128)
NON_NULLABLE_COLUMNS: bool(true)
```

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_client_info - Returns an object with properties that describe the DB2 database client" on page 68

db2_special_columns - Returns a result set listing the unique row identifier columns for a table

Syntax:

resource db2_special_columns(**resource** *connection*, **string** *qualifier*, **string** *schema*, **string** *table_name*, **int** *scope*)

db2_special_columns - Returns a result set listing the unique row identifier columns for a table

Description:

Returns a result set listing the unique row identifier columns for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables.

table_name

The name of the table.

scope

Integer value representing the minimum duration for which the unique row identifier is valid. This can be one of the values listed in Table 6.

Table 6. Minimum duration of validity for unique row identifiers

Integer value	SQL constant	Description
0	SQL_SCOPE_CURROW	Row identifier is valid only while the cursor is positioned on the row.
1	SQL_SCOPE_TRANSACTION	Row identifier is valid for the duration of the transaction.
2	SQL_SCOPE_SESSION	Row identifier is valid for the duration of the connection.

Return Values:

Returns a statement resource with a result set containing rows with unique row identifier information for a table. The rows are composed of the following columns:

Table 7. db2_special_columns() result set columns

Column name	Description
SCOPE	See Table 6 for the definition of the values specifying the minimum duration of validity for unique row identifiers.
COLUMN_NAME	Name of the unique column.
DATA_TYPE	SQL data type for the column.
TYPE_NAME	Character string representation of the SQL data type for the column.
COLUMN_SIZE	An integer value representing the size of the column.
BUFFER_LENGTH	Maximum number of bytes necessary to store data from this column.

Table 7. db2_special_columns() result set columns (continued)

Column name	Description
DECIMAL_DIGITS	The scale of the column, or NULL where scale is not applicable.
NUM_PREC_RADIX	An integer value of either **10** (representing an exact numeric data type), **2** (representing an approximate numeric data type), or NULL (representing a data type for which radix is not applicable).
PSEUDO_COLUMN	Always returns 1.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_statistics - Returns a result set listing the index and statistics for a table" on page 85

db2_statistics - Returns a result set listing the index and statistics for a table

Syntax:

resource db2_statistics(**resource** *connection*, **string** *qualifier*, **string** *schema*, **string** *table-name*, **bool** *unique*)

Description:

Returns a result set listing the index and statistics for a table.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema that contains the targeted table. If this parameter is NULL, the statistics and indexes are returned for the schema of the current user.

table_name

The name of the table.

unique

An integer value representing the type of index information to return.

0

Return only the information for unique indexes on the table.

1

Return the information for all indexes on the table.

Return Values:

Returns a statement resource with a result set containing rows describing the statistics and indexes for the base tables matching the specified parameters. The rows are composed of the following columns:

Table 8. db2_statistics() result set columns

Column name	Description
TABLE_CAT	The catalog that contains the table. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema that contains the table.
TABLE_NAME	Name of the table.
NON_UNIQUE	An integer value representing whether the index prohibits unique values, or whether the row represents statistics on the table itself: **Return value** **Parameter type** **0 (SQL_FALSE)** The index allows duplicate values. **1 (SQL_TRUE)** The index values must be unique. **NULL** This row provides statistics information for the table itself.
INDEX_QUALIFIER	A string value representing the qualifier that would have to be prepended to INDEX_NAME to fully qualify the index.
INDEX_NAME	A string representing the name of the index.
TYPE	An integer value representing the type of information contained in this row of the result set: **Return value** **Parameter type** **0 (SQL_TABLE_STAT)** The row contains statistics about the table itself. **1 (SQL_INDEX_CLUSTERED)** The row contains information about a clustered index. **2 (SQL_INDEX_HASH)** The row contains information about a hashed index. **3 (SQL_INDEX_OTHER)** The row contains information about a type of index that is neither clustered nor hashed.
ORDINAL_POSITION	The 1-indexed position of the column in the index. NULL if the row contains statistics information about the table itself.

Table 8. db2_statistics() result set columns (continued)

Column name	Description
COLUMN_NAME	The name of the column in the index. NULL if the row contains statistics information about the table itself.
ASC_OR_DESC	**A** if the column is sorted in ascending order, **D** if the column is sorted in descending order, NULL if the row contains statistics information about the table itself.
CARDINALITY	If the row contains information about an index, this column contains an integer value representing the number of unique values in the index. If the row contains information about the table itself, this column contains an integer value representing the number of rows in the table.
PAGES	If the row contains information about an index, this column contains an integer value representing the number of pages used to store the index. If the row contains information about the table itself, this column contains an integer value representing the number of pages used to store the table.
FILTER_CONDITION	Always returns NULL.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_special_columns - Returns a result set listing the unique row identifier columns for a table" on page 83

db2_table_privileges - Returns a result set listing the tables and associated privileges in a database

Syntax:

resource db2_table_privileges(**resource** *connection*, [**string** *qualifier*, [**string** *schema*, [**string** *table_name*]]])

Description:

Returns a result set listing the tables and associated privileges in a database.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. This parameter accepts a search pattern containing _ and % as wildcards.

table_name

The name of the table. This parameter accepts a search pattern containing _ and % as wildcards.

Return Values:

Returns a statement resource with a result set containing rows describing the privileges for the tables that match the specified parameters. The rows are composed of the following columns:

Table 9. db2_privileges() result set columns

Column name	Description
TABLE_CAT	The catalog that contains the table. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema that contains the table.
TABLE_NAME	Name of the table.
GRANTOR	Authorization ID of the user who granted the privilege.
GRANTEE	Authorization ID of the user to whom the privilege was granted.
PRIVILEGE	The privilege that has been granted. This can be one of ALTER, CONTROL, DELETE, INDEX, INSERT, REFERENCES, SELECT, or UPDATE.
IS_GRANTABLE	A string value of "YES" or "NO" indicating whether the grantee can grant the privilege to other users.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_special_columns - Returns a result set listing the unique row identifier columns for a table" on page 83
- "db2_statistics - Returns a result set listing the index and statistics for a table" on page 85
- "db2_tables - Returns a result set listing the tables and associated metadata in a database" on page 88

db2_tables - Returns a result set listing the tables and associated metadata in a database

Syntax:

db2_tables - Returns a result set listing the tables and associated metadata in a database

resource db2_tables(**resource** *connection*, [**string** *qualifier*, [**string** *schema*, [**string** *table-name*, [**string** *table-type*]]]])

Description:

Returns a result set listing the tables and associated metadata in a database.

Parameters:

connection

A valid connection to an IBM DB2, Cloudscape, or Apache Derby database.

qualifier

A qualifier for DB2 for z/OS. For other databases, pass NULL or an empty string.

schema

The schema which contains the tables. This parameter accepts a search pattern containing _ and % as wildcards.

table-name

The name of the table. This parameter accepts a search pattern containing _ and % as wildcards.

table-type

A list of comma-delimited table type identifiers. To match all table types, pass NULL or an empty string. Valid table type identifiers include: ALIAS, HIERARCHY TABLE, INOPERATIVE VIEW, NICKNAME, MATERIALIZED QUERY TABLE, SYSTEM TABLE, TABLE, TYPED TABLE, TYPED VIEW, and VIEW.

Return Values:

Returns a statement resource with a result set containing rows describing the tables that match the specified parameters. The rows are composed of the following columns:

Table 10. db2_tables() result set columns

Column name	Description
TABLE_CAT	The catalog that contains the table. The value is NULL if this table does not have catalogs.
TABLE_SCHEM	Name of the schema that contains the table.
TABLE_NAME	Name of the table.
TABLE_TYPE	Table type identifier for the table.
REMARKS	Description of the table.

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_special_columns - Returns a result set listing the unique row identifier columns for a table" on page 83

- "db2_statistics - Returns a result set listing the index and statistics for a table" on page 85
- "db2_table_privileges - Returns a result set listing the tables and associated privileges in a database" on page 87

Statement metadata functions

db2_cursor_type - Returns the cursor type used by a statement resource

Syntax:

int db2_cursor_type(**resource** *stmt*)

Description:

Returns the cursor type used by a statement resource. Use this to determine if you are working with a forward-only cursor or scrollable cursor.

Parameters:

stmt

A valid statement resource.

Return Values:

Returns either **DB2_FORWARD_ONLY** if the statement resource uses a forward-only cursor or **DB2_SCROLLABLE** if the statement resource uses a scrollable cursor.

Related tasks:

- "Fetching rows from result sets in PHP (ibm_db2)" on page 16

db2_field_display_size - Returns the maximum number of bytes required to display a column

Syntax:

int db2_field_display_size(**resource** *stmt*, **mixed** *column*)

Description:

Returns the maximum number of bytes required to display a column in a result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns an integer value with the maximum number of bytes required to display the specified column. If the column does not exist in the result set, `db2_field_display_size()` returns FALSE.

Related reference:

- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94

db2_field_name - Returns the name of the column in the result set

Syntax:

string db2_field_name(**resource** *stmt*, **mixed** *column*)

Description:

Returns the name of the specified column in the result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns a string containing the name of the specified column. If the specified column does not exist in the result set, `db2_field_name()` returns FALSE.

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93

- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94

db2_field_num - Returns the position of the named column in a result set

Syntax:

int db2_field_num(**resource** *stmt*, **mixed** *column*)

Description:

Returns the position of the named column in a result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns an integer containing the 0-indexed position of the named column in the result set. If the specified column does not exist in the result set, `db2_field_num()` returns FALSE.

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94

db2_field_precision - Returns the precision of the indicated column in a result set

Syntax:

int db2_field_precision(**resource** *stmt*, **mixed** *column*)

Description:

Returns the precision of the indicated column in a result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns an integer containing the precision of the specified column. If the specified column does not exist in the result set, `db2_field_precision()` returns FALSE.

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94

db2_field_scale - Returns the scale of the indicated column in a result set

Syntax:

int db2_field_scale(**resource** *stmt*, **mixed** *column*)

Description:

Returns the scale of the indicated column in a result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns an integer containing the scale of the specified column. If the specified column does not exist in the result set, `db2_field_scale()` returns FALSE.

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94

db2_field_type - Returns the data type of the indicated column in a result set

Syntax:

string db2_field_type(**resource** *stmt*, **mixed** *column*)

Description:

Returns the data type of the indicated column in a result set.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns a string containing the defined data type of the specified column. If the specified column does not exist in the result set, `db2_field_type()` returns FALSE.

db2_field_width - Returns the width of the current value of the indicated column in a result set

Syntax:

int db2_field_width(**resource** *stmt*, **mixed** *column*)

Description:

Returns the width of the current value of the indicated column in a result set. This is the maximum width of the column for a fixed-length data type, or the actual width of the column for a variable-length data type.

Parameters:

stmt

Specifies a statement resource containing a result set.

column

Specifies the column in the result set. This can either be an integer representing the 0-indexed position of the column, or a string containing the name of the column.

Return Values:

Returns an integer containing the width of the specified character or binary data type column in a result set. If the specified column does not exist in the result set, `db2_field_width()` returns FALSE.

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94

db2_num_fields - Returns the number of fields contained in a result set

Syntax:

int db2_num_fields(**resource** *stmt*)

Description:

Returns the number of fields contained in a result set. This is most useful for handling the result sets returned by dynamically generated queries, or for result sets returned by stored procedures, where your application cannot otherwise know how to retrieve and use the results.

Parameters:

stmt

A valid statement resource containing a result set.

Return Values:

Returns an integer value representing the number of fields in the result set associated with the specified statement resource. Returns FALSE if the statement resource is not a valid input value.

Examples:

Retrieving the number of fields in a result set:

The following example demonstrates how to retrieve the number of fields returned in a result set.

```
<?php

$sql = "SELECT id, name, breed, weight FROM animals ORDER BY breed";
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, $sql);
$columns = db2_num_fields($stmt);

echo "There are {$columns} columns in the result set.";
?>
```

The preceding example returns the following output:

```
There are 4 columns in the result set.
```

Related tasks:

- "Retrieving database metadata (ibm_db2)" on page 8

Related reference:

- "db2_field_display_size - Returns the maximum number of bytes required to display a column" on page 90
- "db2_num_rows - Returns the number of rows affected by an SQL statement" on page 96
- "db2_field_name - Returns the name of the column in the result set" on page 91
- "db2_field_num - Returns the position of the named column in a result set" on page 92
- "db2_field_precision - Returns the precision of the indicated column in a result set" on page 92
- "db2_field_scale - Returns the scale of the indicated column in a result set" on page 93
- "db2_field_type - Returns the data type of the indicated column in a result set" on page 94
- "db2_field_width - Returns the width of the current value of the indicated column in a result set" on page 94

db2_num_rows - Returns the number of rows affected by an SQL statement

Syntax:

int db2_num_rows(**resource** *stmt*)

Description:

Returns the number of rows deleted, inserted, or updated by an SQL statement.

To determine the number of rows that will be returned by a SELECT statement, issue SELECT COUNT(*) with the same predicates as your intended SELECT statement and retrieve the value.

If your application logic checks the number of rows returned by a SELECT statement and branches if the number of rows is 0, consider modifying your application to attempt to return the first row with one of `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_array()`, or `db2_fetch_row()`, and branch if the fetch function returns FALSE.

db2_num_rows - Returns the number of rows affected by an SQL statement

If you issue a SELECT statement using a scrollable cursor, `db2_num_rows()` returns the number of rows returned by the SELECT statement. However, the overhead associated with scrollable cursors significantly degrades the performance of your application, so if this is the only reason you are considering using scrollable cursors, you should use a forward-only cursor and either call SELECT COUNT(*) or rely on the *boolean* return value of the fetch functions to achieve the equivalent functionality with much better performance.

Parameters:

stmt

A valid **stmt** resource containing a result set.

Return Values:

Returns the number of rows affected by the last SQL statement issued by the specified statement handle.

Related tasks:

- "Executing a single SQL statement in PHP (ibm_db2)" on page 11
- "Preparing and executing SQL statements in PHP (ibm_db2)" on page 12

Related reference:

- "db2_num_fields - Returns the number of fields contained in a result set" on page 95

db2_num_rows - Returns the number of rows affected by an SQL statement

Chapter 5. PDO_ODBC Driver Reference

PDO object methods

PDO::__construct - Creates a PDO instance representing a connection to a database

Syntax:

PDO PDO::__construct(string *dsn*, [**string** *username*, [**string** *password*, [**array** *driver_options*]]])

Description:

Creates a PDO instance to represent a connection to the requested database.

Parameters:

dsn

The Data Source Name, or DSN, contains the information required to connect to the database.

In general, a DSN consists of the PDO driver name, followed by a colon, followed by the PDO driver-specific connection syntax. To create a cataloged connection to a DB2 database through the PDO_ODBC driver, the DSN syntax is "**odbc:***database-name*". For example, a DSN of "odbc:SAMPLE" connects to a cataloged DB2 database named SAMPLE using the PDO_ODBC driver.

To create an uncataloged connection to a DB2 database, the DSN syntax is as follows:

```
odbc:DRIVER={IBM DB2 ODBC DRIVER};HOSTNAME=hostname;PORT=port;
DATABASE=database;PROTOCOL=TCPIP;UID= USER;PWD=password;
```

For example, the following DSN connects to a DB2 database named SAMPLE running on LOCALHOST over TCP/IP port 50000, using a user name of "db2inst1" and a password of "ibmdb2":

```
odbc:DRIVER={IBM DB2 ODBC DRIVER};HOSTNAME=localhost;PORT=50000;
DATABASE=SAMPLE;PROTOCOL=TCPIP;UID=db2inst1;PWD=ibmdb2;
```

The **dsn** parameter supports three different methods of specifying the arguments required to create a database connection:

Driver invocation

dsn contains the full DSN.

Uniform Resource Identifier (URI) invocation

dsn consists of **uri:** followed by a URI that defines the location of a file containing the DSN string. The URI can specify a local file or a remote URL.

```
uri:file:///path/to/dsnfile
```

Aliasing

dsn consists of a name **name** that maps to **pdo.dsn.***name* in `php.ini` defining the DSN string.

The alias must be defined in `php.ini`, and not `.htaccess` or `httpd.conf`.

username

The user name for the DSN string. This parameter is optional if you have specified the UID= clause in the *dsn* parameter.

password

The password for the DSN string. This parameter is optional if you have specified the PWD= clause in the *dsn* parameter.

driver_options

An associative array of driver-specific connection options.

Return Values:

Returns a PDO object representing a successful database connection, or throws an exception if the connection attempt fails.

Exceptions:

PDO::construct throws a PDOException if the attempt to connect to the requested database fails.

Examples:

Create a PDO instance via driver invocation:

```
<?php
/* Connect to an ODBC database using driver invocation */
$dsn = 'odbc:SAMPLE';
$user = 'dbuser';
$password = 'dbpass';

try {
    $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

Create a PDO instance via URI invocation:

The following example assumes that the file `/usr/local/dbconnect` exists with file permissions that enable PHP to read the file. The file contains the PDO DSN to connect to a DB2 database through the PDO_ODBC driver:

```
odbc:DRIVER={IBM DB2 ODBC DRIVER};HOSTNAME=localhost;
PORT=50000;DATABASE=SAMPLE;PROTOCOL=TCPIP;UID=db2inst1;PWD=ibmdb2;
```

The PHP script can then create a database connection by simply passing the **uri:** parameter and pointing to the file URI:

```
<?php
/* Connect to an ODBC database using driver invocation */
$dsn = 'uri:file:///usr/local/dbconnect';
```

```
$user = '';
$password = '';

try {
    $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

Create a PDO instance using an alias:

The following example assumes that `php.ini` contains the following entry to enable a connection to a DB2 database cataloged as SAMPLE using only the alias **MYDB**:

```
[PDO]
pdo.dsn.MYDB="odbc:SAMPLE"
```

```
<?php
/* Connect to an ODBC database using an alias */
$dsn = 'mydb';
$user = 'db2inst1';
$password = 'ibmdb2';

try {
    $dbh = new PDO($dsn, $user, $password);
}catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

PDO::beginTransaction - Initiates a transaction

Syntax:

bool PDO::beginTransaction()

Description:

Turns off autocommit mode. While autocommit mode is turned off, changes made to the database via the PDO object instance are not committed until you end the transaction by calling `PDO::commit()`. Calling `PDO::rollback()` will roll back all changes to the database and return the connection to autocommit mode.

Examples:

Roll back a transaction:

The following example begins a transaction and issues two statements that modify the database before rolling back the changes.

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema and data */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
    SET name = 'hamburger'");

/* Recognize mistake and roll back changes */
```

```
$dbh->rollBack();

/* Database connection is now back in autocommit mode */
?>
```

PDO::commit - Commits a transaction

Syntax:

bool PDO::commit()

Description:

Commits a transaction, returning the database connection to autocommit mode until the next call to `PDO::beginTransaction()` starts a new transaction.

Examples:

Commit a transaction:

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema */
$sth = $dbh->exec("DROP TABLE fruit");

/* Commit the changes */
$dbh->commit();

/* Database connection is now back in autocommit mode */
?>
```

PDO::errorCode - Fetch the SQLSTATE associated with the last operation on the database handle

Syntax:

int PDO::errorCode()

Description:

Fetches the SQLSTATE associated with the last operation on the database handle.

Return Values:

Returns a SQLSTATE, a five-character alphanumeric identifier defined in the ANSI SQL-92 standard. An SQLSTATE consists of a two-character class value followed by a three-character subclass value. A class value of 01 indicates a warning and is accompanied by a return code of `SQL_SUCCESS_WITH_INFO`. Class values other than '01', except for the class 'IM', indicate an error. The class 'IM' is specific to warnings and errors that derive from the implementation of PDO (or perhaps ODBC, if you're using the ODBC driver) itself. The subclass value '000' in any class indicates that there is no subclass for that SQLSTATE.

`PDO::errorCode()` only retrieves error codes for operations performed directly on the database handle. If you create a PDOStatement object through `PDO::prepare()` or `PDO::query()` and invoke an error on the statement handle, `PDO::errorCode()`

will not reflect that error. You must call PDOStatement::errorCode() to return the error code for an operation performed on a particular statement handle.

Examples:

Retrieving a SQLSTATE code:

```
<?php
/* Provoke an error -- the BONES table does not exist */
$dbh->exec("INSERT INTO bones(skull) VALUES ('lucy')");

echo "\nPDO::errorCode(): ";
print $dbh->errorCode();
?>
```

The preceding example returns the following output:

```
PDO::errorCode(): 42S02
```

PDO::errorInfo - Fetch extended error information associated with the last operation on the database handle

Syntax:

array PDO::errorInfo()

Description:

Return Values:

PDO::errorInfo() returns an array of error information about the last operation performed by this database handle. The array consists of the following fields:

Element	Information
0	SQLSTATE error code (a five-character alphanumeric identifier defined in the ANSI SQL standard).
1	Driver-specific error code.
2	Driver-specific error message.

PDO::errorInfo() only retrieves error information for operations performed directly on the database handle. If you create a PDOStatement object through PDO::prepare() or PDO::query() and invoke an error on the statement handle, PDO::errorInfo() will not reflect the error from the statement handle. You must call PDOStatement::errorInfo() to return the error information for an operation performed on a particular statement handle.

Examples:

Displaying errorInfo() fields for a PDO_ODBC connection to a DB2 database:

```
<?php
/* Provoke an error -- the BONES table does not exist */
$err = $dbh->prepare('SELECT skull FROM bones');
$err->execute();
echo "\nPDO::errorInfo():\n";
print_r($err->errorInfo());
?>
```

The preceding example returns the following output:

```
PDO::errorInfo():
Array
(
  [0] => 42S02
  [1] => -204
  [2] => [IBM][CLI Driver][DB2/LINUX] SQL0204N "DANIELS.BONES"
         is an undefined name. SQLSTATE=42704
)
```

PDO::exec - Execute an SQL statement and return the number of affected rows

Syntax:

int PDO::exec(string *statement*)

Description:

`PDO::exec()` executes an SQL statement in a single function call, returning the number of rows affected by the statement.

`PDO::exec()` does not return results from a SELECT statement. For a SELECT statement that you only need to issue once during your program, consider issuing `PDO::query()`. For a statement that you need to issue multiple times, prepare a PDOStatement object with `PDO::prepare()` and issue the statement with `PDOStatement::execute()`.

Parameters:

statement

The SQL statement to prepare and execute.

Return Values:

`PDO::exec()` returns the number of rows that were modified or deleted by the SQL statement you issued. If no rows were affected, `PDO::exec()` returns **0**.

This function may return Boolean FALSE, but may also return a non-Boolean value which evaluates to FALSE, such as **0** or ″″. Use the === operator for testing the return value of this function.

The following example incorrectly relies on the return value of `PDO::exec()`, wherein a statement that affected 0 rows results in a call to die:

```
<?php
$db->exec() or die($db->errorInfo());
?>
```

Examples:

Issuing a DELETE statement:

Count the number of rows deleted by a DELETE statement with no WHERE clause.

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');
```

```
/* Delete all rows from the FRUIT table */
$count = $dbh->exec("DELETE FROM fruit WHERE colour = 'red'");

/* Return number of rows that were deleted */
print("Deleted $count rows.\n");
?>
```

The preceding example returns the following output:

```
Deleted 1 rows.
```

PDO::getAttribute - Retrieve a database connection attribute

Syntax:

mixed PDO::getAttribute(int *attribute*)

Description:

This function returns the value of a database connection attribute. To retrieve PDOStatement attributes, refer to `PDOStatement::getAttribute()`.

Note that some database and driver combinations may not support all of the database connection attributes.

Parameters:

attribute

One of the **PDO::ATTR_*** constants. The constants that apply to database connections are as follows: **PDO::ATTR_AUTOCOMMIT PDO::ATTR_CASE PDO::ATTR_CLIENT_VERSION PDO::ATTR_CONNECTION_STATUS PDO::ATTR_DRIVER_NAME PDO::ATTR_ERRMODE PDO::ATTR_ORACLE_NULLS PDO::ATTR_PERSISTENT PDO::ATTR_PREFETCH PDO::ATTR_SERVER_INFO PDO::ATTR_SERVER_VERSION PDO::ATTR_TIMEOUT**

Return Values:

A successful call returns the value of the requested PDO attribute. An unsuccessful call returns **null**.

Examples:

Retrieving database connection attributes:

```
<?php
$conn = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');
$attributes = array(
    "AUTOCOMMIT", "ERRMODE", "CASE", "CLIENT_VERSION", "CONNECTION_STATUS",
    "ORACLE_NULLS", "PERSISTENT", "PREFETCH", "SERVER_INFO", "SERVER_VERSION",
    "TIMEOUT"
);

foreach ($attributes as $val) {
    echo "PDO::ATTR_$val: ";
    echo $conn->getAttribute(constant("PDO::ATTR_$val")) . "\n";
}
?>
```

PDO::getAvailableDrivers - Return an array of available PDO drivers

Syntax:

array PDO::getAvailableDrivers()

Description:

This function returns a list of all PDO drivers which can be used to construct a PDO object. This is a static method.

Return Values:

`PDO::getAvailableDrivers()` returns an array of PDO driver names. If no drivers are available, it returns an empty array.

Examples:

A PDO::getAvailableDrivers example:

```
<?php
print_r(PDO::getAvailableDrivers());
?>
```

The preceding example returns the following output something similar to:

```
Array
(
    [0] => odbc
)
```

PDO::lastInsertId - Returns the ID of the last inserted row or sequence value

Syntax:

string PDO::lastInsertId([**string** *sequence-name*])

Description:

Returns the ID of the last row that was inserted into the database using the same PDO database connection.

Parameters:

sequence-name

Name of the sequence object from which the ID should be returned. If you do not specify this parameter, PDO_ODBC returns the value of the identity column for the last inserted row.

Return Values:

If a sequence name was not specified for the **name** parameter, `PDOStatement::lastInsertId()` returns a string representing the identity column of the last row that was inserted into the database.

If a sequence name was specified for the **name** parameter, `PDOStatement::lastInsertId()` returns a string representing the last value retrieved from the specified sequence object.

PDO::prepare - Prepares a statement for execution and returns a statement object

Syntax:

PDOStatement PDO::prepare(string *statement*, [**array** *driver_options*])

Description:

Prepares an SQL statement to be executed by the `PDOStatement::execute()` method. The SQL statement can contain zero or more named (*:name*) or question mark (*?*) parameter markers for which real values will be substituted when the statement is executed. You cannot use both named and question mark parameter markers within the same SQL statement; pick one or the other parameter style.

Calling `PDO::prepare()` and `PDOStatement::execute()` for statements that will be issued multiple times with different parameter values optimizes the performance of your application by allowing the driver to negotiate client and server side caching of the query plan and meta information, and helps to prevent SQL injection attacks by eliminating the need to manually quote the parameters.

PDO supports named parameter markers in the PDO_ODBC driver for DB2 by rewriting named parameters to question mark parameters.

Parameters:

statement

This must be a valid SQL statement for the target database management system.

driver_options

This associative array sets attribute values for the PDOStatement object that this method returns. You would most commonly use this to set the **PDO::ATTR_CURSOR** value to **PDO::CURSOR_SCROLL** to request a scrollable cursor.

Return Values:

If the database management system successfully prepares the statement, `PDO::prepare()` returns a PDOStatement object.

Examples:

Prepare an SQL statement with named parameters:

```
<?php
/* Execute a prepared statement by passing an array of values */
$sql = 'SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour';
$sth = $dbh->prepare($sql, array(PDO::ATTR_CURSOR, PDO::CURSOR_FWDONLY));
$sth->execute(array(':calories' => 150, ':colour' => 'red'));
```

```
$red = $sth->fetchAll();
$sth->execute(array(':calories' => 175, ':colour' => 'yellow'));
$yellow = $sth->fetchAll();
?>
```

Prepare an SQL statement with question mark parameters:

```
<?php
/* Execute a prepared statement by passing an array of values */
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->execute(array(150, 'red'));
$red = $sth->fetchAll();
$sth->execute(array(175, 'yellow'));
$yellow = $sth->fetchAll();
?>
```

PDO::query - Executes an SQL statement, returning a result set as a PDOStatement object

Syntax:

PDOStatement PDO::query(string *statement*)

Description:

`PDO::query()` executes an SQL statement in a single function call, returning the result set (if any) returned by the statement as a PDOStatement object.

For a query that you need to issue multiple times, you will realize better performance if you prepare a PDOStatement object using `PDO::prepare()` and issue the statement with multiple calls to `PDOStatement::execute()`.

If you do not fetch all of the data in a result set before issuing your next call to PDO::query, your call may fail. Call `PDOStatement::closeCursor()` to release the database resources associated with the PDOStatement object before issuing your next call to `PDO::query()`.

Parameters:

statement

The SQL statement to prepare and execute.

Return Values:

`PDO::query()` returns a PDOStatement object.

Examples:

Demonstrate PDO::query:

A nice feature of `PDO::query()` is that it enables you to iterate over the rowset returned by a successfully executed SELECT statement.

```
<?php
function getFruit($conn) {
    $sql = 'SELECT name, colour, calories FROM fruit ORDER BY name';
    foreach ($conn->query($sql) as $row) {
        print $row['NAME'] . "\t";
```

```
        print $row['COLOUR'] . "\t";
        print $row['CALORIES'] . "\n";
    }
}
?>
```

The preceding example returns the following output:

```
apple   red     150
banana  yellow  250
kiwi    brown   75
lemon   yellow  25
orange  orange  300
pear    green   150
watermelon      pink    90
```

PDO::quote - Quotes a string for use in a query

Syntax:

string PDO::quote(**string** *string*, [**int** *parameter_type*])

Description:

The PDO_ODBC driver for DB2 does not support this method. `PDO::quote()` is intended to place quotes around the input string and escape any single quotes within the input string.

If you are using this function to build SQL statements, you are strongly recommended to use `PDO::prepare()` to prepare SQL statements with bound parameters instead of using `PDO::quote()` to interpolate user input into a SQL statement. Prepared statements with bound parameters are not only more portable, more convenient, and vastly more secure, but are also much faster than interpolating user input into slight variations on the same basic SQL statement.

Parameters:

string

The string to be quoted.

parameter_type

Provides a data type hint for drivers that have alternate quoting styles. The default value is **PDO::PARAM_STR**.

Return Values:

Returns a quoted string that is theoretically safe to pass into an SQL statement. Returns FALSE if the driver does not support quoting in this way.

Examples:

Quoting a normal string:

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Simple string */
```

```
$string = 'Nice';
print "Unquoted string: $string\n";
print "Quoted string: " . $conn->quote($string) . "\n";
?>
```

The preceding example returns the following output:

```
Unquoted string: Nice
Quoted string: 'Nice'
```

Quoting a dangerous string:

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Dangerous string */
$string = 'Naughty \' string';
print "Unquoted string: $string\n";
print "Quoted string:" . $conn->quote($string) . "\n";
?>
```

The preceding example returns the following output:

```
Unquoted string: Naughty ' string
Quoted string: 'Naughty '' string'
```

Quoting a complex string:

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Complex string */
$string = "Co'mpl''ex \"st'\"ring";
print "Unquoted string: $string\n";
print "Quoted string: " . $conn->quote($string) . "\n";
?>
```

The preceding example returns the following output:

```
Unquoted string: Co'mpl''ex "st'"ring
Quoted string: 'Co''mpl''''ex "st''"ring'
```

PDO::rollBack - Rolls back a transaction

Syntax:

bool PDO::rollBack()

Description:

Rolls back the current transaction, as initiated by `PDO::beginTransaction()`. It is an error to call this method if no transaction is active.

If the database was set to autocommit mode, this function will restore autocommit mode after it has rolled back the transaction.

Examples:

Roll back a transaction:

The following example begins a transaction and issues two statements that modify the database before rolling back the changes.

```
<?php
/* Begin a transaction, turning off autocommit */
$dbh->beginTransaction();

/* Change the database schema and data */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
    SET name = 'hamburger'");

/* Recognize mistake and roll back changes */
$dbh->rollBack();

/* Database connection is now back in autocommit mode */
?>
```

PDO::setAttribute - Set an attribute

Syntax:

bool PDO::setAttribute(*int* **attribute**, *mixed* **value**)

Description:

Sets an attribute on the database handle. The available generic attributes are listed below.

PDO::ATTR_CASE: Forces column names to a specific case.

- **PDO::CASE_LOWER**: Force column names to lower case.
- **PDO::CASE_NATURAL**: Leave column names as returned by the database driver.
- **PDO::CASE_UPPER**: Force column names to upper case.

PDO::ATTR_ERRMODE: Error reporting.

- **PDO::ERRMODE_SILENT**: Just set error codes.
- **PDO::ERRMODE_WARNING**: Raise E_WARNING level errors.
- **PDO::ERRMODE_EXCEPTION**: Throw exceptions.

PDO::ATTR_ORACLE_NULLS : Conversion of NULL and empty strings.

- **PDO::NULL_NATURAL**: No conversion.
- **PDO::NULL_EMPTY_STRING**: Empty string is converted to NULL.
- **PDO::NULL_TO_STRING**: NULL is converted to an empty string.

PDO::ATTR_STRINGIFY_FETCHES:
Convert numeric values to strings when fetching. Requires *bool*.

PDO::ATTR_STATEMENT_CLASS:
Set user-supplied statement class derived from PDOStatement. Cannot be used with persistent PDO instances. Requires **array(string classname, array(mixed ctor_args))**.

PDO::ATTR_AUTOCOMMIT:
Whether to autocommit every single statement.

PDOstatement object methods

PDOStatement::bindColumn - Bind a column to a PHP variable

Syntax:

bool PDOStatement::bindColumn(**mixed** *column*, **mixed** *param*, [**int** *type*])

Description:

`PDOStatement::bindColumn()` arranges to have a particular variable bound to a given column in the result-set from a query. Each call to `PDOStatement::fetch()` or `PDOStatement::fetchAll()` will update all the variables that are bound to columns.

Since information about the columns is not always available to PDO until the statement is executed, portable applications should call this function after `PDO::execute()`.

Parameters:

column

Number of the column (1-indexed) or name of the column in the result set. If using the column name, be aware that the name should match the case of the column, as returned by the driver.

param

Name of the PHP variable to which the column will be bound.

type

Data type of the parameter, specified by the **PDO::PARAM_*** constants.

Examples:

Binding result set output to PHP variables:

Binding columns in the result set to PHP variables is an effective way to make the data contained in each row immediately available to your application. The following example demonstrates how PDO allows you to bind and retrieve columns with a variety of options and with intelligent defaults.

```
<?php
function readData($dbh) {
  $sql = 'SELECT name, colour, calories FROM fruit';
  try {
    $stmt = $dbh->prepare($sql);
    $stmt->execute();

    /* Bind by column number */
    $stmt->bindColumn(1, $name);
    $stmt->bindColumn(2, $colour);

    /* Bind by column name */
    $stmt->bindColumn('calories', $cals);

    while ($row = $stmt->fetch(PDO::FETCH_BOUND)) {
      $data = $name . "\t" . $colour . "\t" . $cals . "\n";
      print $data;
    }
  }
```

```
  catch (PDOException $e) {
    print $e->getMessage();
  }
}
readData($dbh);
?>
```

The preceding example returns the following output:

```
apple   red     150
banana  yellow  175
kiwi    green   75
orange  orange  150
mango   red     200
strawberry      red     25
```

PDOStatement::bindParam - Binds a parameter to the specified variable name

Syntax:

bool PDOStatement::bindParam(**mixed** *parameter*, **mixed** *variable*, [**int** *data_type*, [**int** *length*, [**mixed** *driver_options*]]])

Description:

Binds a PHP variable to a corresponding named or question mark placeholder in the SQL statement that was use to prepare the statement. Unlike `PDOStatement::bindValue()`, the variable is bound as a reference and will only be evaluated at the time that `PDOStatement::execute()` is called.

Most parameters are input parameters, that is, parameters that are used in a read-only fashion to build up the query. PDO_ODBC supports the invocation of stored procedures that return data as output (OUT) parameters, as well as INOUT parameters that both pass an input value to the database and are updated with an output value after the CALL statement.

Parameters:

parameter

Parameter identifier. For a prepared statement using named placeholders, this will be a parameter name of the form *:name*. For a prepared statement using question mark placeholders, this will be the 1-indexed position of the parameter.

variable

Name of the PHP variable to bind to the SQL statement parameter.

data_type

Explicit data type for the parameter using the `PDO::PARAM_*` constants. To return an INOUT parameter from a stored procedure, use the bitwise OR operator to set the `PDO::PARAM_INPUT_OUTPUT` bits for the **data_type** parameter.

length

Length of the data type. To indicate that a parameter is an OUT parameter from a stored procedure, you must explicitly set the length.

driver_options

Examples:

Execute a prepared statement with named placeholders:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Execute a prepared statement with question mark placeholders:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Call a stored procedure with an INOUT parameter:

```
<?php
/* Call a stored procedure with an INOUT parameter */
$colour = 'red';
$sth = $dbh->prepare('CALL puree_fruit(?)');
$sth->bindParam(1, $colour, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 12);
$sth->execute();
print("After pureeing fruit, the colour is: $colour");
?>
```

PDOStatement::bindValue - Binds a value to a parameter

Syntax:

bool PDOStatement::bindValue(**mixed** *parameter*, **mixed** *value*, [**int** *data_type*])

Description:

Binds a value to a corresponding named or question mark placeholder in the SQL statement that was use to prepare the statement.

Parameters:

parameter

Parameter identifier. For a prepared statement using named placeholders, this will be a parameter name of the form *:name*. For a prepared statement using question mark placeholders, this will be the 1-indexed position of the parameter.

value

The value to bind to the parameter.

data_type

Explicit data type for the parameter using the PDO::PARAM_* constants.

Examples:

Execute a prepared statement with named placeholders:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindValue(':calories', $calories, PDO::PARAM_INT);
$sth->bindValue(':colour', $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

Execute a prepared statement with question mark placeholders:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindValue(1, $calories, PDO::PARAM_INT);
$sth->bindValue(2, $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

PDOStatement::closeCursor - Closes the cursor, enabling the statement to be executed again

Syntax:

bool PDOStatement::closeCursor()

Description:

`PDOStatement::closeCursor()` frees up the connection to the database so that other SQL statements may be issued, but leaves the statement in a state that enables it to be executed again.

Examples:

A PDOStatement::closeCursor() example:

In the following example, the *$stmt* PDOStatement object returns multiple rows but the application fetches only the first row, leaving the PDOStatement object representing a result set with unfetched rows. To ensure that the application will work with all databases and PDO drivers, the author inserts a call to `PDOStatement::closeCursor()` on *$stmt* before executing the *$otherStmt* PDOStatement object.

```
<?php
/* Create a PDOStatement object */
$stmt = $dbh->prepare('SELECT foo FROM bar');

/* Create a second PDOStatement object */
$stmt = $dbh->prepare('SELECT foobaz FROM foobar');

/* Execute the first statement */
$stmt->execute();

/* Fetch only the first row from the results */
$stmt->fetch();

/* The following call to closeCursor() may be required by some drivers */
$stmt->closeCursor();

/* Now we can execute the second statement */
$otherStmt->execute();
?>
```

PDOStatement::columnCount - Returns the number of columns in the result set

Syntax:

int PDOStatement::columnCount()

Description:

Use `PDOStatement::columnCount()` to return the number of columns in the result set represented by the PDOStatement object.

If the PDOStatement object was returned from `PDO::query()`, the column count is immediately available.

If the PDOStatement object was returned from `PDO::prepare()`, an accurate column count will not be available until you invoke `PDOStatement::execute()`.

Return Values:

Returns the number of columns in the result set represented by the PDOStatement object. If there is no result set, `PDOStatement::columnCount()` returns **0**.

Examples:

Counting columns:

This example demonstrates how `PDOStatement::columnCount()` operates with and without a result set.

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');

$sth = $dbh->prepare("SELECT name, colour FROM fruit");

/* Count the number of columns in the (non-existent) result set */
$colcount = $sth->columnCount();
print("Before execute(), result set has $colcount columns (should be 0)\n");

$sth->execute();
```

```
/* Count the number of columns in the result set */
$colcount = $sth->columnCount();
print("After execute(), result set has $colcount columns (should be 2)\n");

?>
```

The preceding example returns the following output:

```
Before execute(), result set has 0 columns (should be 0)
After execute(), result set has 2 columns (should be 2)
```

PDOStatement::errorCode - Fetch the SQLSTATE associated with the last operation on the statement handle

Syntax:

int PDOStatement::errorCode()

Description:

Return Values:

Identical to `PDO::errorCode()`, except that `PDOStatement::errorCode()` only retrieves error codes for operations that are performed with PDOStatement objects.

Examples:

Retrieving a SQLSTATE code:

```
<?php
/* Provoke an error -- the BONES table does not exist */
$err = $dbh->prepare('SELECT skull FROM bones');
$err->execute();

echo "\nPDOStatement::errorCode(): ";
print $err->errorCode();
?>
```

The preceding example returns the following output:

```
PDOStatement::errorCode(): 42S02
```

PDOStatement::errorInfo - Fetch extended error information associated with the last operation on the statement handle

Syntax:

array PDOStatement::errorInfo()

Description:

Return Values:

`PDOStatement::errorInfo()` returns an array of error information about the last operation performed by this statement handle. The array consists of the following fields:

Element	Information
0	SQLSTATE error code (a five-character alphanumeric identifier defined in the ANSI SQL standard).
1	Driver-specific error code.
2	Driver-specific error message.

Examples:

Displaying errorInfo() fields for a PDO_ODBC connection to a DB2 database:

```
<?php
/* Provoke an error -- the BONES table does not exist */
$sth = $dbh->prepare('SELECT skull FROM bones');
$sth->execute();

echo "\nPDOStatement::errorInfo():\n";
$arr = $sth->errorInfo();
print_r($arr);
?>
```

The preceding example returns the following output:

```
PDOStatement::errorInfo():
Array
(
   [0] => 42S02
   [1] => -204
   [2] => [IBM][CLI Driver][DB2/LINUX] SQL0204N "DANIELS.BONES"
          is an undefined name.SQLSTATE=42704
)
```

PDOStatement::execute - Executes a prepared statement

Syntax:

bool PDOStatement::execute([**array** *input-parameters*])

Description:

Execute the prepared statement. If the prepared statement included parameter markers, you must either:

- call `PDOStatement::bindParam()` to bind PHP variables to the parameter markers: bound variables pass their value as input and receive the output value, if any, of their associated parameter markers
- or pass an array of input-only parameter values to the *input-parameters* argument

Examples:

Execute a prepared statement with bound variables:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
```

```
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

Execute a prepared statement with an array of insert values:

```
<?php
/* Execute a prepared statement by passing an array of insert values */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->execute(array(':calories' => $calories, ':colour' => $colour));
?>
```

Execute a prepared statement with question mark placeholders:

```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

PDOStatement::fetch - Fetches the next row from a result set

Syntax:

mixed PDOStatement::fetch([**int** *fetch_style*, [**int** *cursor_orientation*, [**int** *cursor_offset*]]])

Description:

Fetches a row from a result set associated with a PDOStatement object. The **fetch_style** parameter determines how PDO returns the row.

Parameters:

fetch_style
: Controls how the next row will be returned to the caller. This value must be one of the **PDO::FETCH_*** constants, defaulting to **PDO::FETCH_BOTH**.

  PDO::FETCH_ASSOC
  : returns an array indexed by column name as returned in your result set

  PDO::FETCH_BOTH (default)
  : returns an array indexed by both column name and 0-indexed column number as returned in your result set

  PDO::FETCH_BOUND
  : returns TRUE and assigns the values of the columns in your result set to the PHP variables to which they were bound with the `PDOStatement::bindParam()` method

PDO::FETCH_CLASS
returns a new instance of the requested class, mapping the columns of the result set to named properties in the class

PDO::FETCH_INTO
updates an existing instance of the requested class, mapping the columns of the result set to named properties in the class

PDO::FETCH_LAZY
combines **PDO::FETCH_BOTH** and **PDO::FETCH_OBJ**, creating the object variable names as they are accessed

PDO::FETCH_NUM
returns an array indexed by column number as returned in your result set, starting at column 0

PDO::FETCH_OBJ
returns an anonymous object with property names that correspond to the column names returned in your result set

cursor_orientation
For a PDOStatement object representing a scrollable cursor, this value determines which row will be returned to the caller. This value must be one of the **PDO::FETCH_ORI_*** constants, defaulting to **PDO::FETCH_ORI_NEXT**. To request a scrollable cursor for your PDOStatement object, you must set the **PDO::ATTR_CURSOR** attribute to **PDO::CURSOR_SCROLL** when you prepare the SQL statement with `PDO::prepare()`.

offset

For a PDOStatement object representing a scrollable cursor for which the *cursor_orientation* parameter is set to **PDO::FETCH_ORI_ABS**, this value specifies the absolute number of the row in the result set that shall be fetched.

For a PDOStatement object representing a scrollable cursor for which the *cursor_orientation* parameter is set to **PDO::FETCH_ORI_REL**, this value specifies the row to fetch relative to the cursor position before `PDOStatement::fetch()` was called.

Examples:

Fetching rows using different fetch styles:

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Exercise PDOStatement::fetch styles */
print("PDO::FETCH_ASSOC: ");
print("Return next row as an array indexed by column name\n");
$result = $sth->fetch(PDO::FETCH_ASSOC);
print_r($result);
print("\n");

print("PDO::FETCH_BOTH: ");
print("Return next row as an array indexed by both column name and number\n");
$result = $sth->fetch(PDO::FETCH_BOTH);
print_r($result);
print("\n");

print("PDO::FETCH_LAZY: ");
print("Return next row as an anonymous object with column names as properties\n");
```

```
$result = $sth->fetch(PDO::FETCH_LAZY);
print_r($result);
print("\n");

print("PDO::FETCH_OBJ: ");
print("Return next row as an anonymous object with column names as properties\n");
$result = $sth->fetch(PDO::FETCH_OBJ);
print $result->NAME;
print("\n");
?>
```

The preceding example returns the following output:

```
PDO::FETCH_ASSOC: Return next row as an array indexed by column name
Array
(
    [NAME] => apple
    [COLOUR] => red
)

PDO::FETCH_BOTH: Return next row as an array indexed by both column name and number
Array
(
    [NAME] => banana
    [0] => banana
    [COLOUR] => yellow
    [1] => yellow
)

PDO::FETCH_LAZY: Return next row as an anonymous object with column names as
properties
PDORow Object
(
    [NAME] => orange
    [COLOUR] => orange
)

PDO::FETCH_OBJ: Return next row as an anonymous object with column names as
properties
kiwi
```

Fetching rows with a scrollable cursor:

```
<?php
function readDataForwards($dbh) {
  $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY BET';
  try {
    $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR, PDO::CURSOR_SCROLL));
    $stmt->execute();
    while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT)) {
      $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
      print $data;
    }
    $stmt = null;
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}
function readDataBackwards($dbh) {
  $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY bet';
  try {
    $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR, PDO::CURSOR_SCROLL));
    $stmt->execute();
    $row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_LAST);
    do {
      $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
      print $data;
```

```
    } while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_PRIOR));
    $stmt = null;
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}

print "Reading forwards:\n";
readDataForwards($conn);

print "Reading backwards:\n";
readDataBackwards($conn);
?>
```

The preceding example returns the following output:

```
Reading forwards:
21    10    5
16    0     5
19    20    10

Reading backwards:
19    20    10
16    0     5
21    10    5
```

PDOStatement::fetchAll - Returns an array containing all of the result set rows

Syntax:

array PDOStatement::fetchAll([int *fetch_style*, **[int** *column_index*]])

Description:

Returns an array containing all of the result set rows.

Parameters:

fetch_style

Controls the contents of the returned array as documented in `PDOStatement::fetch()`. Defaults to **PDO::FETCH_BOTH**.

To return an array consisting of all values of a single column from the result set, specify **PDO::FETCH_COLUMN**. You can specify which column you want with the **column-index** parameter.

To fetch only the unique values of a single column from the result set, bitwise-OR **PDO::FETCH_COLUMN** with **PDO::FETCH_UNIQUE**.

To return an associative array grouped by the values of a specified column, bitwise-OR **PDO::FETCH_COLUMN** with **PDO::FETCH_GROUP**.

column_index

Returns the indicated 0-indexed column when the value of **fetch_style** is **PDO::FETCH_COLUMN**. Defaults to **0**.

Return Values:

PDOStatement::fetchAll - Returns an array containing all of the result set rows

PDOStatement::fetchAll returns an array containing all of the remaining rows in the result set. The array represents each row as either an array of column values or an object with properties corresponding to each column name.

Using this method to fetch large result sets will result in a heavy demand on system and possibly network resources. Rather than retrieving all of the data and manipulating it in PHP, consider using the database management system to manipulate the result sets. For example, use the WHERE and SORT BY clauses in SQL to restrict results before retrieving and processing them with PHP.

Examples:

Fetch all remaining rows in a result set:

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Fetch all of the remaining rows in the result set */
print("Fetch all of the remaining rows in the result set:\n");
$result = $sth->fetchAll();
print_r($result);
?>
```

The preceding example returns the following output:

```
Fetch all of the remaining rows in the result set:
Array
(
    [0] => Array
        (
            [NAME] => pear
            [0] => pear
            [COLOUR] => green
            [1] => green
        )

    [1] => Array
        (
            [NAME] => watermelon
            [0] => watermelon
            [COLOUR] => pink
            [1] => pink
        )

)
```

Fetching all values of a single column from a result set:

The following example demonstrates how to return all of the values of a single column from a result set, even though the SQL statement itself may return multiple columns per row.

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Fetch all of the values of the first column */
$result = $sth->fetchAll(PDO::FETCH_COLUMN, 0);
var_dump($result);
?>
```

The preceding example returns the following output:

```
Array(3)
(
    [0] =>
    string(5) => apple
    [1] =>
    string(4) => pear
    [2] =>
    string(10) => watermelon
)
```

Grouping all values by a single column:

The following example demonstrates how to return an associative array grouped by the values of the specified column in the result set. The array contains three keys: values **apple** and **pear** are returned as arrays that contain two different colors, while **watermelon** is returned as an array that contains only one color.

```
<?php
$insert = $dbh->prepare("INSERT INTO fruit(name, colour) VALUES (?, ?)");
$insert->execute('apple', 'green');
$insert->execute('pear', 'yellow');

$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Group values by the first column */
var_dump($sth->fetchAll(PDO::FETCH_COLUMN|PDO::FETCH_GROUP));
?>
```

The preceding example returns the following output:

```
array(3) {
  ["apple"]=>
  array(2) {
    [0]=>
    string(5) "green"
    [1]=>
    string(3) "red"
  }
  ["pear"]=>
  array(2) {
    [0]=>
    string(5) "green"
    [1]=>
    string(6) "yellow"
  }
  ["watermelon"]=>
  array(1) {
    [0]=>
    string(5) "green"
  }
}
```

PDOStatement::fetchColumn - Returns a single column from the next row of a result set

Syntax:

string PDOStatement::fetchColumn([**int** *column_number*])

Description:

Returns a single column from the next row of a result set.

Parameters:

column_number

0-indexed number of the column you wish to retrieve from the row. If no value is supplied, `PDOStatement::fetchColumn()` fetches the first column.

Return Values:

`PDOStatement::fetchColumn()` returns a single column in the next row of a result set.

There is no way to return another column from the same row if you use `PDOStatement::fetchColumn()` to retrieve data.

Examples:

Return first column of the next row:

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* Fetch the first column from the next row in the result set */
print("Fetch the first column from the next row in the result set:\n");
$result = $sth->fetchColumn();
print("name = $result\n");

print("Fetch the second column from the next row in the result set:\n");
$result = $sth->fetchColumn(1);
print("colour = $result\n");
?>
```

The preceding example returns the following output:

```
Fetch the first column from the next row in the result set:
name = lemon
Fetch the second column from the next row in the result set:
colour = red
```

PDOStatement::getAttribute - Retrieve a statement attribute

Syntax:

mixed PDOStatement::getAttribute(int *attribute*)

Description:

Gets an attribute of the statement. Currently, the only attribute that can be retrieved is **PDO::ATTR_CURSOR_NAME**, which returns the name of the cursor corresponding to this PDOStatement object.

PDOStatement::getColumnMeta - Returns metadata for a column in a result set

Syntax:

mixed PDOStatement::getColumnMeta(int *column*)

Description:

PDOStatement::getColumnMeta - Returns metadata for a column in a result set

Retrieves the metadata for a 0-indexed column in a result set as an associative array.

Parameters:

column

The 0-indexed column in the result set.

Return Values:

Returns an associative array containing the following values representing the metadata for a single column:

Table 11. Column metadata

Name	Value
native_type	The PHP native type used to represent the column value.
driver:decl_type	The SQL type used to represent the column value in the database. If the column in the result set is the result of a function, this value is not returned by `PDOStatement::getColumnMeta()`.
flags	Any flags set for this column.
name	The name of this column as returned by the database.
len	The length of this column. Normally **-1** for types other than floating point decimals.
precision	The numeric precision of this column. Normally **0** for types other than floating point decimals.
pdo_type	The type of this column as represented by the **PDO::PARAM_*** constants.

Returns FALSE if the requested column does not exist in the result set, or if no result set exists.

Examples:

Retrieving column metadata:

The following example shows the results of retrieving the metadata for a single column generated by a function (COUNT).

```
<?php
$select = $DB->query('SELECT COUNT(*) FROM fruit');
$meta = $select->getColumnMeta(0);
var_dump($meta);
?>
```

The preceding example returns the following output:

```
array(6) {
  ["native_type"]=>
  string(7) "integer"
  ["flags"]=>
  array(0) {
  }
```

```
  ["name"]=>
  string(8) "COUNT(*)"
  ["len"]=>
  int(-1)
  ["precision"]=>
  int(0)
  ["pdo_type"]=>
  int(2)
}
```

PDOStatement::nextRowset - Advances to the next result set in a statement handle associated with multiple result sets

Syntax:

bool PDOStatement::nextRowset()

Description:

Some database management systems, such as DB2, support stored procedures that return more than one result set (also known as a result set). `PDOStatement::nextRowSet()` enables you to access the second and subsequent result sets associated with a PDOStatement object. Each result set can have a different set of columns from the preceding result set.

Return Values:

Returns TRUE on success or FALSE on failure.

Examples:

Fetching multiple result sets returned from a stored procedure:

The following example shows how to call a stored procedure, MULTIPLE_RESULTS, that returns three result sets. We use a do-while loop to loop over the `PDOStatement::nextRowset()` method, which returns false and terminates the loop when no more result sets can be returned.

```
<?php
$sql = 'CALL multiple_results()';
$stmt = $conn->query($sql);
$i = 1;
do {
    $rowset = $stmt->fetchAll(PDO::FETCH_NUM);
    if ($rowset) {
        printResultSet($rowset, $i);
    }
    $i++;
} while ($stmt->nextRowset());

function printResultSet(&$rowset, $i) {
    print "Result set $i:\n";
    foreach ($rowset as $row) {
        foreach ($row as $col) {
            print $col . "\t";
        }
        print "\n";
    }
    print "\n";
}
?>
```

The preceding example returns the following output:

```
Result set 1:
apple    red
banana   yellow

Result set 2:
orange   orange    150
banana   yellow    175

Result set 3:
lime     green
apple    red
banana   yellow
```

PDOStatement::rowCount - Returns the number of rows affected by the last SQL statement

Syntax:

int PDOStatement::rowCount()

Description:

`PDOStatement::rowCount()` returns the number of rows affected by the last DELETE, INSERT, or UPDATE statement executed by the corresponding **PDOStatement** object.

If the last SQL statement executed by the associated **PDOStatement** was a SELECT statement issued with a scrollable cursor, DB2 returns the number of rows returned by that statement. However, scrollable cursors require more system resources on the database management system and are not recommended for general usage.

Examples:

Return the number of deleted rows:

`PDOStatement::rowCount()` returns the number of rows affected by a DELETE, INSERT, or UPDATE statement.

```
<?php
/* Delete all rows from the FRUIT table */
$del = $dbh->prepare('DELETE FROM fruit');
$del->execute();

/* Return number of rows that were deleted */
print("Return number of rows that were deleted:\n");
$count = $del->rowCount();
print("Deleted $count rows.\n");
?>
```

The preceding example returns the following output:

```
Deleted 9 rows.
```

Counting rows returned by a SELECT statement:

For most databases, `PDOStatement::rowCount()` does not return the number of rows affected by a SELECT statement. Instead, use `PDO::query()` to issue a SELECT COUNT(*) statement with the same predicates as your intended SELECT

statement, then use PDOStatement::fetchColumn() to retrieve the number of rows that will be returned. Your application can then perform the correct action.

```
<?php
$sql = "SELECT COUNT(*) FROM fruit WHERE calories > 100";
if ($res = $conn->query($sql)) {

    /* Check the number of rows that match the SELECT statement */
    if ($res->fetchColumn() > 0) {

        /* Issue the real SELECT statement and work with the results */
         $sql = "SELECT name FROM fruit WHERE calories > 100";
         foreach ($conn->query($sql) as $row) {
             print "Name: " .  $row['NAME'] . "\n";
         }
    }
    /* No rows matched -- do something else */
    else {
        print "No rows matched the query.";
    }
}

$res = null;
$conn = null;
?>
```

The preceding example returns the following output:

```
apple
banana
orange
pear
```

PDOStatement::setAttribute - Set a statement attribute

Syntax:

bool PDOStatement::setAttribute(int *attribute*, **mixed** *value*)

Description:

Sets an attribute on the statement. Currently, you can only set the **PDO::ATTR_CURSOR_NAME** attribute to set the name of the cursor associated with this PDOStatement object.

PDOStatement::setFetchMode - Set the default fetch mode for this statement

Syntax:

bool PDOStatement::setFetchMode(int *mode*)

Description:

Parameters:

mode

The fetch mode must be one of the **PDO::FETCH_*** constants.

Return Values:

Returns **1** on success or FALSE on failure.

Examples:

Setting the fetch mode:

The following example demonstrates how `PDOStatement::setFetchMode()` changes the default fetch mode for a PDOStatement object.

```
<?php
$sql = 'SELECT name, colour, calories FROM fruit';
try {
  $stmt = $dbh->query($sql);
  $result = $stmt->setFetchMode(PDO::FETCH_NUM);
  while ($row = $stmt->fetch()) {
    print $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
  }
}
catch (PDOException $e) {
  print $e->getMessage();
}
?>
```

The preceding example returns the following output:

```
apple   red     150
banana  yellow  250
orange  orange  300
kiwi    brown   75
lemon   yellow  25
pear    green   150
watermelon      pink    90
```

Part 2. Developing Perl Applications

Chapter 6. Developing Perl Applications

Programming Considerations for Perl

Perl is a popular programming language that is freely available for many operating systems. Using the DBD::DB2 driver available from http://www.ibm.com/software/data/db2/perl with the Perl Database Interface (DBI) Module available from http://www.perl.com, you can create DB2 applications using Perl.

Because Perl is an interpreted language and the Perl DBI Module uses dynamic SQL, Perl is an ideal language for quickly creating and revising prototypes of DB2 applications. The Perl DBI Module uses an interface that is quite similar to the CLI and JDBC interfaces, which makes it easy for you to port your Perl prototypes to CLI and JDBC.

Most database vendors provide a database driver for the Perl DBI Module, which means that you can also use Perl to create applications that access data from many different database servers. For example, you can write a Perl DB2 application that connects to an Oracle database using the DBD::Oracle database driver, fetch data from the Oracle database, and insert the data into a DB2 database using the DBD::DB2 database driver.

Related concepts:

- "Database Connections in Perl" on page 134
- "Example of a Perl Program" on page 136

Related tasks:

- "Building Perl applications" on page 136

Perl DBI

DB2 supports the Perl Database Interface (DBI) specification for data access through the DBD::DB2 driver. The DB2 Perl DBI website is located at:

```
http://www.ibm.com/software/data/db2/perl/
```

and contains the latest DBD::DB2 driver, and related information.

Perl is an interpreted language and the Perl DBI Module uses dynamic SQL. These properties make Perl an ideal language for quickly creating and revising DB2 applications. The Perl DBI Module uses an interface that is quite similar to the CLI and JDBC interfaces, which makes it easy to port Perl applications to CLI and JDBC, or to port CLI and JDBC applications to Perl.

Related concepts:

- "Programming Considerations for Perl" on page 133

Database Connections in Perl

To enable Perl to load the DBI module, you must include the following line in your DB2 application:

```
use DBI;
```

The DBI module automatically loads the DBD::DB2 driver when you create a *database handle* using the **DBI->connect** statement with the following syntax:

```
my $dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password);
```

where:

$dbhandle
represents the database handle returned by the connect statement

dbalias
represents a DB2 alias cataloged in your DB2 database directory

$userID
represents the user ID used to connect to the database

$password
represents the password for the user ID used to connect to the database

Fetching Results in Perl

Because the Perl DBI Module only supports dynamic SQL, you do not use host variables in your Perl DB2 applications.

Procedure:

To return results from an SQL query, perform the following steps:

1. Create a database handle by connecting to the database with the `DBI->connect` statement.
2. Create a statement handle from the database handle. For example, you can call **prepare** with an SQL statement as a string argument to return statement handle *$sth* from the database handle, as demonstrated in the following Perl statement:

   ```
   my $sth = $dbhandle->prepare(
      'SELECT firstnme, lastname
         FROM employee '
      );
   ```

3. Execute the SQL statement by calling **execute** on the statement handle. A successful call to **execute** associates a result set with the statement handle. For example, you can execute the statement prepared in the previous example using the following Perl statement:

   ```
   #Note: $rc represents the return code for the execute call
   my $rc = $sth->execute();
   ```

4. Fetch a row from the result set associated with the statement handle with a call to **fetchrow()**. The Perl DBI returns a row as an array with one value per column. For example, you can return all of the rows from the statement handle in the previous example using the following Perl statement:

   ```
   while (($firstnme, $lastname) = $sth->fetchrow()) {
      print "$firstnme $lastname\n";
   }
   ```

Related concepts:

- "Database Connections in Perl" on page 134

Parameter Markers in Perl

To enable you to execute a prepared statement using different input values for specified fields, the Perl DBI module enables you to prepare and execute a statement using parameter markers. To include a parameter marker in an SQL statement, use the question mark (?) character.

The following Perl code creates a statement handle that accepts a parameter marker for the WHERE clause of a SELECT statement. The code then executes the statement twice using the input values 25000 and 35000 to replace the parameter marker.

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee
      WHERE salary > ?'
   );

my $rc = $sth->execute(25000);

⋮
my $rc = $sth->execute(35000);
```

SQLSTATE and SQLCODE Variables in Perl

To return the SQLSTATE associated with a Perl DBI database handle or statement handle, call the **state** method. For example, to return the SQLSTATE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlstate = $dbhandle->state;
```

To return the SQLCODE associated with a Perl DBI database handle or statement handle, call the **err** method. To return the message for an SQLCODE associated with a Perl DBI database handle or statement handle, call the **errstr** method. For example, to return the SQLCODE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlcode = $dbhandle->err;
```

Perl Restrictions

The Perl DBI module supports only dynamic SQL. When you need to execute a statement multiple times, you can improve the performance of your Perl DB2 applications by issuing a **prepare** call to prepare the statement.

Perl does not support multiple-thread database access.

For current information on the restrictions of the version of the DBD::DB2 driver that you install on your workstation, refer to the CAVEATS file in the DBD::DB2 driver package.

Related concepts:

- "Programming Considerations for Perl" on page 133

Example of a Perl Program

Following is an example of an application written in Perl:

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]\n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname\n";
}

# check for problems which may have terminated the fetch early
warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;
```

Related concepts:

- "Programming Considerations for Perl" on page 133

Related tasks:

- "Building Perl applications" on page 136

Related reference:

- "Perl Samples" in *Samples Topics*

Building Perl applications

DB2 supports database access for client applications written in Perl 5.8 or later. At the time of printing, release 0.78 of the DB2 driver (DBD::DB2) for the Perl Database Interface (Perl DBI) Version 1.41 or later is supported and available for AIX, HP-UX, Linux, Solaris and Windows. For information on how to obtain the latest driver, visit http://www.ibm.com/db2/perl.

DB2 provides Perl sample programs located on UNIX in the `sqllib/samples/perl` directory, and on Windows in the `sqllib\samples\perl` directory.

Procedure:

To run the perl interpreter on a DB2 Perl program on the command line, enter the interpreter name and the program name (including extension):

- If connecting locally on the server:

```
perl dbauth.pl
```

- If connecting from a remote client:

```
perl dbauth.pl sample <userid> <password>
```

Some programs require support files to be run. The `tbsel` sample program requires several tables created by the `tbselcreate.db2` CLP script. The `tbselinit` script (UNIX), or the `tbselinit.bat` batch file (Windows), first calls `tbseldrop.db2` to drop the tables if they exist, and then calls `tbselcreate.db2` to create them. So to run the program, you would enter the following commands:

- If connecting locally on the server:

```
tbselinit
perl tbsel.pl
```

- If connecting from a remote client:

```
tbselinit
perl tbsel.pl sample <userid> <password>
```

Note: For a remote client, you need to modify the connect statement in the `tbselinit` or `tbselinit.bat` file to hardcode your user ID and password:
`db2 connect to sample user <userid> using <password>`

Calling routines

DB2 client applications can access routines (stored procedures and user-defined functions) that are created by supported host languages or by SQL procedures. For example, the sample program `spclient.pl` can access the SQL procedures `spserver` shared library, if it exists in the database.

Note: To build a host language routine, you must have the appropriate compiler set up on the server. SQL procedures do not require a compiler. The shared library can only be built on the server, and not from a remote client.

To demonstrate calling SQL procedures, go to the `samples/sqlproc` directory (UNIX) or the `samples\sqlproc` directory (Windows) on the server, and run the following commands to create and catalog the SQL procedures in the `spserver` library:

```
db2 connect to sample
db2 -td@ -vf spserver.db2
```

Next, come back to the `perl` samples directory (this can be on a remote client machine), and run the Perl interpreter on the client program to access the `spserver` shared library:

- If connecting locally on the server, enter the following command:

```
perl spclient
```

- If connecting from a remote client, enter the following command:

```
perl spclient sample <userid> <password>
```

Related concepts:

- "Perl DBI" on page 133
- "Programming Considerations for Perl" on page 133

Related reference:

- "Perl Samples" in *Samples Topics*

Part 3. Appendixes

Appendix A. DB2 Database technical information

Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF CD)
  - printed books
- Command line help
  - Command help
  - Message help
- Sample programs

IBM periodically makes documentation updates available. If you access the online version on the DB2 Information Center at ibm.com®, you do not need to install documentation updates because this version is kept up-to-date by IBM. If you have installed the DB2 Information Center, it is recommended that you install the documentation updates. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* or downloaded from Passport Advantage as new information becomes available.

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and Redbooks™ online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how we can improve the DB2 documentation, send an e-mail to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this e-mail address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

Related concepts:

- "Features of the DB2 Information Center" in *Online DB2 Information Center*
- "Sample files" in *Samples Topics*

Related tasks:

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 147

Related reference:

- "DB2 technical library in hardcopy or PDF format" on page 142

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order. DB2 Version 9 manuals in PDF format can be downloaded from www.ibm.com/software/data/db2/udb/support/manualsv9.html.

Although the tables identify books available in print, the books might not be available in your country or region.

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect™ or other DB2 products.

Table 12. DB2 technical information

Name	Form Number	Available in print
Administration Guide: Implementation	SC10-4221	Yes
Administration Guide: Planning	SC10-4223	Yes
Administrative API Reference	SC10-4231	Yes
Administrative SQL Routines and Views	SC10-4293	No
Call Level Interface Guide and Reference, Volume 1	SC10-4224	Yes
Call Level Interface Guide and Reference, Volume 2	SC10-4225	Yes
Command Reference	SC10-4226	No
Data Movement Utilities Guide and Reference	SC10-4227	Yes
Data Recovery and High Availability Guide and Reference	SC10-4228	Yes
Developing ADO.NET and OLE DB Applications	SC10-4230	Yes
Developing Embedded SQL Applications	SC10-4232	Yes

Table 12. DB2 technical information (continued)

Name	Form Number	Available in print
Developing SQL and External Routines	SC10-4373	No
Developing Java Applications	SC10-4233	Yes
Developing Perl and PHP Applications	SC10-4234	No
Getting Started with Database Application Development	SC10-4252	Yes
Getting started with DB2 installation and administration on Linux and Windows	GC10-4247	Yes
Message Reference Volume 1	SC10-4238	No
Message Reference Volume 2	SC10-4239	No
Migration Guide	GC10-4237	Yes
Net Search Extender Administration and User's Guide **Note:** HTML for this document is not installed from the HTML documentation CD.	SH12-6842	Yes
Performance Guide	SC10-4222	Yes
Query Patroller Administration and User's Guide	GC10-4241	Yes
Quick Beginnings for DB2 Clients	GC10-4242	No
Quick Beginnings for DB2 Servers	GC10-4246	Yes
Spatial Extender and Geodetic Data Management Feature User's Guide and Reference	SC18-9749	Yes
SQL Guide	SC10-4248	Yes
SQL Reference, Volume 1	SC10-4249	Yes
SQL Reference, Volume 2	SC10-4250	Yes
System Monitor Guide and Reference	SC10-4251	Yes
Troubleshooting Guide	GC10-4240	No
Visual Explain Tutorial	SC10-4319	No
What's New	SC10-4253	Yes
XML Extender Administration and Programming	SC18-9750	Yes
XML Guide	SC10-4254	Yes
XQuery Reference	SC18-9796	Yes

Table 13. DB2 Connect-specific technical information

Name	Form Number	Available in print
DB2 Connect User's Guide	SC10-4229	Yes

Table 13. DB2 Connect-specific technical information (continued)

Name	Form Number	Available in print
Quick Beginnings for DB2 Connect Personal Edition	GC10-4244	Yes
Quick Beginnings for DB2 Connect Servers	GC10-4243	Yes

Table 14. WebSphere® Information Integration technical information

Name	Form Number	Available in print
WebSphere Information Integration: Administration Guide for Federated Systems	SC19-1020	Yes
WebSphere Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1018	Yes
WebSphere Information Integration: Configuration Guide for Federated Data Sources	SC19-1034	No
WebSphere Information Integration: SQL Replication Guide and Reference	SC19-1030	Yes

Note: The DB2 Release Notes provide additional information specific to your product's release and fix pack level. For more information, see the related links.

Related concepts:

- "Overview of the DB2 technical information" on page 141
- "About the Release Notes" in *Release Notes*

Related tasks:

- "Ordering printed DB2 books" on page 144

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* CD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation CD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation CD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Procedure:

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at http://www.ibm.com/shop/publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
  - Locate the contact information for your local representative from one of the following Web sites:
    - The IBM directory of world wide contacts at www.ibm.com/planetwide
    - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
  - When you call, specify that you want to order a DB2 publication.
  - Provide your representative with the titles and form numbers of the books that you want to order.

Related concepts:

- "Overview of the DB2 technical information" on page 141

Related reference:

- "DB2 technical library in hardcopy or PDF format" on page 142

Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure:

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Related tasks:

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*

Accessing different versions of the DB2 Information Center

For DB2 Version 9 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

Related tasks:

- "Setting up access to DB2 contextual help and documentation" in *Administration Guide: Implementation*

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

Procedure:

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in a Firefox or Mozilla browser:

1. Select the **Tools** —> **Options** —> **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

Related concepts:

- "Overview of the DB2 technical information" on page 141

Updating the DB2 Information Center installed on your computer or intranet server

If you have a locally-installed DB2 Information Center, updated topics can be available for download. The 'Last updated' value found at the bottom of most topics indicates the current level for that topic.

To determine if there is an update available for the entire DB2 Information Center, look for the 'Last updated' value on the Information Center home page. Compare the value in your locally installed home page to the date of the most recent downloadable update at http://www.ibm.com/software/data/db2/udb/support/icupdate.html. You can then update your locally-installed Information Center if a more recent downloadable update is available.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to download and apply updates.
2. Use the Update feature to determine if update packages are available from IBM.

   Note: Updates are also available on CD. For details on how to configure your Information Center to install updates from CD, see the related links.

   If update packages are available, use the Update feature to download the packages. (The Update feature is only available in stand-alone mode.)
3. Stop the stand-alone Information Center, and restart the DB2 Information Center service on your computer.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center service.
   - On Windows, click **Start → Control Panel → Administrative Tools → Services**. Then right-click on **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv9 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `C:\Program Files\IBM\DB2 Information Center\Version 9` directory.
     c. Run the `help_start.bat` file using the fully qualified path for the DB2 Information Center:

        ```
        <DB2 Information Center dir>\doc\bin\help_start.bat
        ```
   - On Linux:

a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `/opt/ibm/db2ic/V9` directory.
b. Run the `help_start` script using the fully qualified path for the DB2 Information Center:

```
<DB2 Information Center dir>/doc/bin/help_start
```

The systems default Web browser launches to display the stand-alone Information Center.

3. Click the Update button (). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the download process, check the selections you want to download, then click **Install Updates**.
5. After the download and installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center.
   - On Windows, run the `help_end.bat` file using the fully qualified path for the DB2 Information Center:

     ```
     <DB2 Information Center dir>\doc\bin\help_end.bat
     ```

     Note: The help_end batch file contains the commands required to safely terminate the processes that were started with the help_start batch file. Do not use `Ctrl-C` or any other method to terminate `help_start.bat`.
   - On Linux, run the `help_end` script using the fully qualified path for the DB2 Information Center:

     ```
     <DB2 Information Center dir>/doc/bin/help_end
     ```

     Note: The help_end script contains the commands required to safely terminate the processes that were started with the help_start script. Do not use any other method to terminate the `help_start` script.
7. Restart the DB2 Information Center service.
   - On Windows, click **Start → Control Panel → Administrative Tools → Services**. Then right-click on **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv9 start
     ```

The updated DB2 Information Center displays the new and updated topics.

Related concepts:

- "DB2 Information Center installation options" in *Quick Beginnings for DB2 Servers*

Related tasks:

- "Installing the DB2 Information Center using the DB2 Setup wizard (Linux)" in *Quick Beginnings for DB2 Servers*
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" in *Quick Beginnings for DB2 Servers*

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

Before you begin:

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials:

To view the tutorial, click on the title.

Native XML data store
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

Related concepts:

- "Visual Explain overview" in *Administration Guide: Implementation*

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

DB2 documentation
Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

DB2 Technical Support Web site
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/support.html

Related concepts:

- "Introduction to problem determination" in *Troubleshooting Guide*
- "Overview of the DB2 technical information" on page 141

Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal use: You may reproduce these Publications for your personal, non commercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these Publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Appendix B. Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

PHP Documentation Group copyright

This documentation incorporates text which is copyright 1997-2005 by the PHP Documentation Group. The text was taken by permission from the PDO and ibm_db2 sections of the PHP Manual (http://php.net/manual/) under the following terms:

- Copyright (c) 1997-2005 by the PHP Documentation Group. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).
- Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.
- Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Trademarks

Company, product, or service names identified in the documents of the DB2 Version 9 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft®, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel®, Itanium®, Pentium®, and Xeon® are trademarks of Intel Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about DB2 products, go to http://www.ibm.com/software/data/db2/.

Printed in USA

SC10-4234-00