**General concepts of database API** [**https://drupal.org/node/310070**](https://drupal.org/node/310070)

**Static queries** [**https://drupal.org/node/310072**](https://drupal.org/node/310072)

**Dynamic queries** [**https://drupal.org/node/310075**](https://drupal.org/node/310075)

**Chaining in dynamic queries** [**https://drupal.org/node/1060924**](https://drupal.org/node/1060924)

**Drupal schema entity relationship** [**https://drupal.org/files/er\_db\_schema\_drupal\_7.png**](https://drupal.org/files/er_db_schema_drupal_7.png)

The Drupal 7 Database API provides a standard, vendor-agnostic abstraction layer for accessing database servers. **You should almost never be making database calls directly unless you are developing core APIs.**

The API is designed to preserve the syntax and power of SQL as much as possible, but also:

* To support multiple database servers easily;
* To allow developers to leverage more complex functionality, such as transactions;
* To provide a structured interface for the dynamic construction of queries;
* To enforce security checks and other good practices;
* To provide modules with a clean interface for intercepting and modifying a site's queries.

**The Drupal database layer is built on top of the PHP's PDO library**. PDO provides a unified, object-oriented API for accessing different databases but it **does not provide an abstraction for the different dialects of SQL used by different databases**.

**Drivers**

Because different databases require different sorts of interaction, the Drupal database layer requires a driver for each database type. A driver consists of a series of files located in includes/database/driver, where driver is a string representing the unique key for that driver. In most cases the driver key is the lowercase version of the database name, such as "mysql", "pgsql", or "mycustomdriver".

Each driver consists of several classes derived from parent classes in the core database system. These driver-specific classes may override whatever behavior is needed to properly support that database type. Driver-specific classes are always named for their parent class followed by an underscore followed by the driver name. For example, the MySQL specific version of InsertQuery is named InsertQuery\_mysql.

Connections

A connection is an object of class DatabaseConnection, which inherits from the PDO class. Every database to which Drupal connects has a single connection object associated with it. That connection object must be subclassed for each individual driver.

To access (and open if necessary) a connection object, use:

**$conn = Database::getConnection($target, $key);**

For more information on Target and Connection Key, please see the documentation page on Database configuration.

**To access the currently active connection**, use:

**$conn = Database::getConnection();**

This will get the default target of the active connection.

Note that in the vast majority of cases you will not need to request the connection object directly. Rather, the procedural wrappers will do so for you. The only reason you would ever need to access a connection object directly is if you are doing complex manipulation of more than one database and you do not want to change the active database.

**To set the active connection, use:**

**db\_set\_active($key);**

Queries

A query is an SQL statement that will be sent to a database connection. There are six types of queries supported by the database system: Static, Dynamic, Insert, Update, Delete, and Merge. Some queries are written as SQL string templates (prepared statements) while others use object-oriented query builders. A "query object" refers to an instance of a query builder for one of the various query types.

Statements

A statement object is the result of a Select query. It will always be of type DatabaseStatement, or possibly a subclass of DatabaseStatement. DatabaseStatement extends the PDOStatement class.

Drupal uses prepared statements for all queries. A prepared statement is a template for a query into which values will be inserted for execution. Think of a prepared statement as the SQL equivalent of a function, which is then called with parameters to use.

In normal PDO, one must explicitly prepare a statement object and then execute it with certain values bound to placeholders in the query. The statement can then be iterated as a result set. Effectively a statement and a result set are synonymous, but only after a statement has been executed.

Drupal does not expose the prepared statement directly. Instead, a module developer will use a query object or a one-off SQL string to execute a query and the statement object for that query is returned. The terms "statement object" and "result set object" are therefore more or less synonymous.

Database configuration

The primary means of defining a database connection is via the $databases array in settings.php. As its name suggests, $databases allows for the definition of multiple database connections. It also supports the definition of multiple targets. A database connection is not opened (the connection object is not created) until the first time some piece of code tries to run a query against that database.

**Connection key**

A connection key is a unique identifier for a given database connection. The connection key must be unique for a given site, and there must always be a connection of "default" that will be the primary Drupal database. On most sites, it will be the only connection defined.

**Target**

A given connection key must have one or more targets. A target is a database that may be used if available. A target of "default" must always be defined for each connection key. If the requested target is not defined, the system will silently fall back to "default".

The primary use of targets is for master/slave replication. The "default" target is the master SQL server. One or more "slave" targets may then be defined (note that in some situations, "slave" is the only valid alternative target, e.g. in static queries). Queries that are flagged to try and use a slave server if available will attempt to access the "slave" target. If one is available, that connection will be opened (if it is not already) and the query will run against the slave server. If not, the query will run against the "default" (master) server instead. That allows for a transparent fallback, so code can be written to take advantage of a slave server if it is available but will still run without one with no modification.

**$databases syntax**

The $databases array is a nested array of at least three levels. The first level defines the database keys. The second defines the database targets. The value of each target is the connection information for that key/target. Some examples should make that clearer.

$databases['default']['default'] = array(

'driver' => 'mysql',

'database' => 'drupaldb',

'username' => 'username',

'password' => 'secret',

'host' => 'localhost',

);

The above $databases array defines a single connection key ("default"), with a single target ("default"). That connection uses a MySQL database (the "driver" key) on localhost named "drupaldb" with a username of "username" and a password of "secret".

For a master/slave configuration, one would define the following:

$databases['default']['default'] = array(

'driver' => 'mysql',

'database' => 'drupaldb1',

'username' => 'username',

'password' => 'secret',

'host' => 'dbserver1',

);

$databases['default']['slave'][] = array(

'driver' => 'mysql',

'database' => 'drupaldb2',

'username' => 'username',

'password' => 'secret',

'host' => 'dbserver2',

);

$databases['default']['slave'][] = array(

'driver' => 'mysql',

'database' => 'drupaldb3',

'username' => 'username',

'password' => 'secret',

'host' => 'dbserver3',

);

This definition provides a single "default" server and two "slave" servers.

Below configuration defines a single main Drupal database and one additional database labelled "extra" that uses SQLite. Note that the SQLite connection information is structured differently than MySQL's. Each driver may have different configuration depending on what is appropriate for it.

$databases[**'default'**]['default'] = array(

'driver' => 'mysql',

'database' => 'drupaldb1',

'username' => 'username',

'password' => 'secret',

'host' => 'dbserver1',

);

$databases[**'extra'**]['default'] = array(

'driver' => 'sqlite',

'database' => 'files/extradb.sqlite',

);

Remember that no matter how many connections you define, Drupal will not open a connection to that database until it is actually used.

# Static queries

**The most common SELECT queries in Drupal are static queries using the db\_query() function**. Static queries are passed to the database nearly verbatim.

db\_query() takes three arguments:

* $query: the query to run. Use placeholders where appropriate and denote all table names with curly braces.
* $args: an array of placeholder values to substitute into the query.
* $options: an array of options to control how the query operates (optional).

db\_query() Executes an arbitrary query string against the active database

Example:

$query = db\_query("SELECT nid, title FROM **{node}**");

$records = $query->fetchAll();

foreach ($records as $record) {

// Do something.

}

Example of Options

function [**db\_query**](https://api.drupal.org/api/drupal/includes%21database%21database.inc/function/db_query/7.x)($query, array $args = array(), array $options = array()) {

if (empty($options['target'])) {

$options['target'] = 'default';

}

return [**Database**](https://api.drupal.org/api/drupal/includes%21database%21database.inc/class/Database/7.x)::[**getConnection**](https://api.drupal.org/api/drupal/includes%21database%21database.inc/function/Database%3A%3AgetConnection/7.x)($options['target'])->[**query**](https://api.drupal.org/api/drupal/7.x/search/query)($query, $args, $options);

}

**Use this function for SELECT queries if it is just a simple query string. If the caller or other modules need to change the query, use db\_select() instead.**

**Do not use this function for INSERT, UPDATE, or DELETE queries. Those should be handled via db\_insert(), db\_update() and db\_delete() respectively.**

**Prefixing**

In static queries, all table names must be wrapped in {}. That flags them so that the database system can attach a prefix string to them if appropriate. Prefixing allows for running multiple sites from the same database or, in limited cases, for sharing selected tables between sites.

**Placeholders**

Placeholders mark where a literal will be inserted into a query for execution. By separating them out from the query itself, we allow the database to differentiate between SQL syntax and user-provided values, thus avoiding SQL injection.

$result = db\_query("SELECT nid, title FROM {node} WHERE created > **:created**", **array(**

**':created' => REQUEST\_TIME - 3600,**

**)**  
);

Note that **placeholders should not be escaped or quoted regardless** of their type. Because they are passed to the database server separately, the server is able to differentiate between the query string and the value on its own.

// **WRONG (quotes around the :type placeholder)**

$result = db\_query("SELECT title FROM {node} WHERE type = ':type'", array(

':type' => 'page',

));

// **CORRECT: (no quotes around the :type placeholder)**

$result = db\_query("SELECT title FROM {node} WHERE type = :type", array(

':type' => 'page',

));

Placeholders cannot be used for column and table names. Instead, if these are derived from unsafe input, they should be run through **db\_escape\_table().**

### Placeholder arrays

Drupal's database layer includes an extra feature of placeholders. If the value passed in for a placeholder is an array, it will be automatically expanded into a comma separated list as will the corresponding placeholder. That means developers do not need to worry about counting how many placeholders they will need.

**db\_query("SELECT \* FROM {node} WHERE nid IN (:nids)", array(':nids' => array(13, 42, 144)));**

db\_query("SELECT \* FROM {node} WHERE nid IN (:nids\_1, :nids\_2, :nids\_3)", array(

':nids\_1' => 13,

':nids\_2' => 42,

':nids\_3' => 144,

));

db\_query("SELECT \* FROM {node} WHERE nid IN (13, 42, 144)");

### Query options

The third parameter todb\_query() (and to the query method of the connection object) is an array of options that direct how the query will behave. There are typically only two directives that will be used by most queries. The other values are mostly for internal use.

The "**target**" key specifies the target to use. If not specified, it defaults to "default". At present, the only other valid value is "slave", to indicate that a query should run against a slave server if one exists.

The "**fetch**" key specifies how records returned from that query will be retrieved. Legal values include PDO::FETCH\_OBJ, PDO::FETCH\_ASSOC, PDO::FETCH\_NUM, PDO::FETCH\_BOTH, or a string representing the name of a class.

The following example will execute a query against a slave server if available and fetch records from the result set as an associative array.

**$result = db\_query("SELECT nid, title FROM {node}", array(), array(**

**'target' => 'slave',**

**'fetch' => PDO::FETCH\_ASSOC,**

**));**

**if ($result) {**

**while ($row = $result->fetchAssoc()) {**

**..**

**}**

**}**

# Fetching into a custom class

Queries can be fetched into objects based on custom classes. For example, if we have a class named ExampleClass the following query will return objects of the type exampleClass.

class ExampleClass {

function \_\_construct() {

// Do something

}

}

$result = db\_query("SELECT id, title FROM {example\_table}", array(), array(

'fetch' => 'ExampleClass',

));

If the class has a \_\_construct() method the objects will be created, the properties will be added to the object, and then the \_\_construct() method will be called.

If there is a **\_\_construct()** method on the object and that needs to be executed **before the properties** are added to the object the following example shows how to do this.

$result = db\_query("SELECT id, title FROM {example\_table}");

foreach ($result->fetchAll(PDO::FETCH\_CLASS | PDO::FETCH\_PROPS\_LATE, 'ExampleClass') as $record) {

// Do something

}

The arguments passed into fetchAll can be used in fetch the same way. PDO::FETCH\_CLASS tells fetchAll to take the returned result set and add the values as properties to the object of type ExampleClass (the second argument). PDO::FETCH\_PROPS\_LATE tells fetchAll to add the result set as properties to the object after \_\_construct() is called.

Dynamic queries

# Introduction to dynamic queries

Dynamic queries refer to queries that are built dynamically by Drupal rather than provided as an explicit query string. All Insert, Update, Delete, and Merge queries must be dynamic. Select queries may be either static or dynamic. Therefore, "dynamic query" generally refers to a dynamic Select query.

**Note: in 90% of select query use cases you will have a static query. If in a critical performance path, you should use db\_query() and friends instead of db\_select() for performance reasons**  
Drupal 7 no longer has [db\_rewrite\_sql](https://api.drupal.org/api/drupal/includes!database.inc/function/db_rewrite_sql/6) and you **must** use a dynamic query to accomplish the same thing. For example, whenever querying the node table you should make use of the "node\_access" tag, as such:

$query = db\_select('node', 'n')

->addTag('node\_access');

Dynamic select queries are started using the db\_select() function as follows:

$query = db\_select('users', 'u', $options);

In this case, "users" is the base table for the query; that is, the first table after the FROM statement. Note that it should not have brackets around it. The query builder will handle that automatically. The second parameter is the alias for the table. If not specified, the name of the table is used. The $options array is optional, and is identical to the $options array for static queries.

The value returned by the db\_select() call is an object of type SelectQuery.  This object has a whole list of methods such as fields(), joins() and group() which can be called to further define the query.

Let's say we want to create a dynamic query which is roughly equivalent to the following static query:

**$result = db\_query("SELECT uid, name, status, created, access FROM {users} u WHERE uid <> 0 LIMIT 50 OFFSET 0");**

The dynamic equivalent begins as follows:

**// Create an object of type SelectQuery**

**$query = db\_select('users', 'u');**

**// Add extra detail to this query object: a condition, fields and a range**

**$query->condition('u.uid', 0, '<>');**

**$query->fields('u', array('uid', 'name', 'status', 'created', 'access'));**

**$query->range(0, 50);**

Same as

**// Create an object of type SelectQuery and directly**

**// add extra detail to this query object: a condition, fields and a range**

**$query = db\_select('users', 'u')**

**->condition('u.uid', 0, '<>')**

**->fields('u', array('uid', 'name', 'status', 'created', 'access'))**

**->range(0, 50);**

## Executing the query

Once the query is built, call the execute() method to compile and run the query.

$result = $query->execute();

The execute() method will return a result set / statement object that is identical to that returned by db\_query(), and it may be iterated or fetched in the exact same way:

$result = $query->execute();

foreach ($result as $record) {

// Do something with each $record

}

## Debugging

To examine the SQL query that the query object uses at a particular point in its lifecycle, print the query object. To examine the arguments, look at the Array returned by the arguments() method:

**echo $query;**

**print\_r($query->\_\_toString());**

**print\_r($query->arguments());**

# Conditional clauses

A "conditional clause" is a portion of a query that restricts the rows matched by certain conditions. In SQL, that means the WHERE or HAVING portions of a SELECT, UPDATE, or DELETE query. In all dynamic queries in Drupal those are implemented using the same mechanism. Except where noted, everything that follows applies to all three of those query types.

**Query:**

SELECT FROM {mytable} WHERE (a = 1 AND b = 'foo' OR (c = 'bar'))

**Conditional object**

Drupal represents each conditional fragment as an instance of class QueryConditional. A conditional object is an instance of that class.

WHERE (a = 1 AND b = 'foo' OR (c = 'bar'))

**Conjunction**

Every conditional clause consists of one or more conditional fragments joined by a conjunction. A conjunction is a term such as AND or OR that joins the two statements together.

AND, OR

**Conditional fragment**

A conditional fragment is a self-contained portion of a conditional clause.

(a = 1 AND b = 'foo' OR (c = 'bar'))

(c = 'bar')

## API

There are two main methods that apply for all conditional objects:

**$query->condition($field, $value = NULL, $operator = '=')**

The condition() method allows for adding a standard $field $value $operator format of conditional fragment. That includes any case where the condition is a binary comparison such as **=, <, >=, LIKE,** etc. If no operator is specified, = is assumed. That means the most common case would be condition('myfield', $value), which results in a conditional fragment of myfield = :value, where :value will be replaced with $value when the query is run.

**$query->where($snippet, $args = array())**

The where() method allows for the addition of arbitrary SQL as a conditional fragment. $snippet may contain any legal SQL fragment, and if it has variable content it must be added using a named placeholder. The $args array is an array of placeholders and values that will be substituted into the snippet. It is up to the developer to ensure that the snippet is valid SQL. No database-specific modifications are made to the snippet.

The condition() method is preferred in most cases, unless the $field $value $operator format is not appropriate, such as happens when you have more complex things like expressions, or a condition on two fields instead of a field and a value. Both methods return the corresponding conditional object, so they may be chained indefinitely.

Unlike Update and Delete queries, Select queries have two types of conditionals: The WHERE clause and the HAVING clause. The Having clause behaves identically to the WHERE clause, except that it uses methods havingCondition() and having() instead of condition() and where().

**$query->havingCondition($field, $value = NULL, $operator = '=')**

**$query->having($snippet, $args = array())**

### Array operators

Some operators are intended to work on an array for the value parameter. The most common of these are IN and BETWEEN. If the operator is IN, then the $value is assumed to be an array of values that the field may equal. Thus, the following call will evaluate this way:

$query->condition('myfield', array(1, 2, 3), 'IN');

// Becomes: myfield IN (:db\_placeholder\_1, :db\_placeholder\_2, :db\_placeholder\_3)

If the operator is BETWEEN, then $value is assumed to be a 2-element array of the values the field must be between. For example:

$query->condition('myfield', array(5, 10), 'BETWEEN');

// Becomes: myfield BETWEEN :db\_placeholder\_1 AND :db\_placeholder\_2

NOT IN is also supported:

$query->condition('myfield', array(1, 2, 3), 'NOT IN');

// myfield NOT IN (:db\_placeholder\_1, :db\_placeholder\_2, :db\_placeholder\_3)

**Using the <> operator will result in an error.**

### Nested conditionals

The first parameter of condition() can also accept another conditional object. That inner conditional object will be incorporated into the outer conditional, surrounded by parentheses. The inner object may also use a different conjunction than the outer object. That way, one can build complex nested conditional structures by creating and building up conditional objects "bottom up".

The db\_condition() helper function will return a new conditional object. It takes a single parameter that is the conjunction that object will use. In general, the helper methods db\_and(), db\_or(), and db\_xor() will cover almost any expected case. That allows conditionals to be inserted inline in a query for a very compact syntax.

$query

->condition('field1', array(1, 2), 'IN')

->condition(db\_or()->condition('field2', 5)->condition('field3', 6))

// Results in:

// (field1 IN (:db\_placeholder\_1, :db\_placeholder\_2) AND (field2 = :db\_placeholder3 OR field3 = :db\_placeholder\_4))

### Null values

To filter a database field on whether the value is or is not NULL, use the following methods:

$query->isNull('myfield');

// Results in (myfield IS NULL)

$query->isNotNull('myfield');

// Results in (myfield IS NOT NULL)

Both methods may be chained and combined with condition() and where() as desired.

**Note: Although it is possible in Drupal 7 to check for NULL values using condition('field', NULL),**

### Subselects

condition() also supports subselects as the $value. To use a subselect, first construct a SelectQuery object created by db\_select(). Then, instead of executing the Select query pass it into the value parameter of the condition() method of another query. It will automatically get integrated into the main query when it is executed.

Subselects are generally useful only in two cases: Where the subselect results in only a single row and value returned and the operator is =, <, >, <=, or >=; or when the subselect returns a single column of information and the operator is IN

**Note: Currently it is only possible to use subselect conditions with the IN operator because with the other operators the sub-query is not wrapped in parentheses and so results in a syntax error.**

db\_delete('sessions')

->condition('timestamp', REQUEST\_TIME - $lifetime, '<')

->execute();

// DELETE FROM {sessions} WHERE (timestamp < 1228713473)

db\_update('sessions')

->fields(array(

'sid' => session\_id()

))

->condition('sid', $old\_session\_id)

->execute();

// UPDATE {sessions} SET sid = 'abcde' WHERE (sid = 'fghij');

// From taxonomy\_term\_save():

$or = db\_or()->condition('tid1', 5)->condition('tid2', 6);

db\_delete('term\_relation')->condition($or)->execute();

// DELETE FROM {term\_relation} WHERE ((tid1 = 5 OR tid2 = 6))

# Count queries

Any query may have a corresponding "count query". The count query returns the number of rows in the original query. To obtain a count query, use the countQuery() method.

$count\_query = $query->countQuery();

$count\_query is now a new Dynamic Select query with no ordering restrictions that when executed will return a result set with only one value, the number of records that would be matched by the original query. Because PHP supports chaining methods on returned objects, the following idiom is a common approach:

$num\_rows = $query->countQuery()->execute()->fetchField();

## Distinct

Some SQL queries may produce duplicate results. In such cases, duplicate rows may be filtered out using the "DISTINCT" keyword in a static query. In a dynamic query, use the distinct() method.

// Force filtering of duplicate records in the result set.

$query->distinct()

Note that DISTINCT can introduce a performance penalty, so do not use it unless there is no other way to restrict the result set to avoid duplicates.

## Expressions

The Select query builder supports the use of expressions in the field list. Examples of expressions include "twice the age field", "a count of all name fields", and a substring of the title field. Be aware that many expressions may use SQL functions, and not all SQL functions are standardized across all databases. It is up to the module developer to ensure that only cross-database compatible expressions are used. (Refer to this list: <http://drupal.org/node/773090>)

To add an expression to a query, use the addExpression() method.

$count\_alias = $query->addExpression('COUNT(uid)', 'uid\_count');

$count\_alias = $query->addExpression('created - :offset', 'timestamp', array(':offset' => 3600));

The first line above will add "COUNT(uid) AS uid\_count" to the query. The second parameter is the alias for the field.

In the rare case that alias is already in use, a new one will be generated and the return value of addExpression() will be the alias used. If no alias is specified, a default of "expression" (or expression\_2, expression\_3, etc.) will be generated.

The optional third parameter is an associative array of placeholder values to use as part of the expression.

# Extenders

# Select queries support the concept of "extenders". An extender is a way to add functionality to a Select query at runtime. That functionality could be additional methods or altering the behavior of existing methods.

For those familiar with object-oriented design patterns, extenders are an implementation of the Decorator Pattern. They attach additional responsibilities to an object dynamically by providing a flexible alternative to subclassing for extending functionality.

## Using an Extender

To use an extender, you must first have a query object. From the query object, the **extend() method** will return a new object that should be used in place of the query object. For example:

$query = $query->extend('PagerDefault');

The above line takes a select query, creates a new PagerDefault query object that contains the original select query, and returns the new object. $query may now be used as if it was the original query object but with additional methods now available.

Note that $query is not altered in place. **The new object is returned from extend(), and if it is not saved to a variable it will be lost**. For example, the following will not do what you expect:

$query = db\_select('node', 'n');

$query

->fields('n', array('nid', 'title'))

->extend('PagerDefault') // This line returns a new PagerDefault object.

->limit(5); // This line works, because the PagerDefault object is what is called.

// The return from extend() was never saved to a variable, so $query is still just the Select object.

$query->orderBy('title');

// This line executes the Select object, not the extender. The extender no longer exists.

$result = $query->execute();

To avoid this problem, the recommended convention for extending a Select query is to do so when the query is first declared.

$query = db\_select('node', 'n')->extend('PagerDefault')->extend('TableSort');

$query->fields(...);

//

That ensures that $query is the fully extended object right from the beginning.

## Creating new extenders

An extender is simply a class that implements the SelectQueryInterface, and takes two parameters in its constructor: A select query (or rather, another object that implements SelectQueryInterface) and a DatabaseConnection object. It must then re-implement the methods of SelectQueryInterface and pass them through to query object specified in the constructor, returning itself where appropriate.

class ExampleExtender extends SelectQueryExtender {

/\*\*

\* Override the normal orderBy behavior.

\*/

public function orderBy($field, $direction = 'ASC') {

return $this;

}

/\*\*

\* Add a new method of our own.

\*/

public function orderByForReal($field, $direction = 'ASC') {

$this->query->orderBy($field, $direction);

return $this;

}

}

The example above overrides the orderBy() method of a query to do nothing, but adds another method, orderByForReal(), that implements actual ordering behavior

Note that in both methods, the $this being returned is the extender object itself. That ensures that the extender doesn't "get lost" by returning the query object.

Any module may declare an Extender. Core ships with two that are generally useful: PagerDefault and TableSort.

# Fields

## Add a field

To add a field to the Select query, use the addField() method:

$title\_field = $query->addField('n', 'title', 'my\_title');

Table / table alias then field name then field alias

The above code will instruct the query to select the "title" field of the table with alias "n", and give it an alias of "my\_title". If no alias is specified, one will be generated automatically.

In this example, that would be "title". If that alias already exists, the alias will be the table name and field name. In this example, that would be "n\_title". If that alias already exists, a counter will be added to the alias until an unused alias is found, such as "n\_title\_2".

Note that if you are creating and populating the query yourself and do not specify an alias and the default alias is not available, there is almost certainly a bug in your code. If you are writing a hook\_query\_alter() implementation, however, you cannot know with certainty what aliases are already in use so **you should always use the generated alias**.

## Add multiple fields

To select multiple fields, call addField() multiple times in the order desired. Note that in most cases the order of fields should not matter, and if it does then there is likely a flaw in the business logic of the module.

As an alternate shorthand, you can use the fields() method to add multiple fields at once.

$query->fields('n', array('nid', 'title', 'created', 'uid'));

The above method is equivalent to calling addField() four times, once for each field. However, fields() does not support specifying an alias for a field. It also returns the query object itself so that the method may be chained rather than returning any generated aliases. If you need to know the generated alias, either use addField() or use getFields() to access the raw internal fields structure.

Calling fields() with no field list will result in a "SELECT \*" query.

$query->fields('n');

That will result in "n.\*" being included in the field list of the query. Note that no aliases will be generated. If a table using SELECT \* contains a field that is also specified directly from another table, it is possible for a field name collision to occur in the result set. In that case, the result set will only contain one of the fields with the common name. For that reason the SELECT \* usage is discouraged.

## Return only one field using fetchField

Use the method fetchField to return only one field with the query, e.g. as follows. (slightly silly example)

$query = db\_select('node', 'n');

$query->condition('n.nid', 123);

$query->addField('n', 'title');

$result = $query->execute();

return $result->fetchField();

# Grouping

To group by a given field, use the groupBy() method.

$query->groupBy('uid');

The above code will instruct the query to group by the uid field. Note that the field name here should be the alias created by the addField() or addExpression() methods, so in most cases you will want to use the return value from those methods here to ensure the correct alias is used.

To obtain the count of rows that have been grouped by a field such as the uid, you could run the following

$query->addExpression('count(uid)', 'uid\_node\_count');

To group by multiple fields, simply call groupBy() multiple times in the order desired.

## Having

It's possible to add a condition on aggregated values.

$query->having('COUNT(uid) >= :matches', array(':matches' => $limit));

This example would find instances where the count of uids is greater than or equal to the $limit. Note that the first parameter to having is not filtered before sending it to the database, so user supplied values should be passed in via the second parameter.

## Examples of groupBy and having

The following code counts the number of nodes per uid:

$query = db\_select('node', 'n')

->fields('n', array('uid'));

$query->addExpression('count(uid)', 'uid\_node\_count');

$query->groupBy("n.uid");

$query->execute();

The next block of code takes the previous example of counting nodes per uid and limits the results to uids that have at least 2 records.

$query = db\_select('node', 'n')

->fields('n',array('uid'));

$query->addExpression('count(uid)', 'uid\_node\_count');

$query->groupBy("n.uid");

$query->having('COUNT(uid) >= :matches', array(':matches' => 2));

$results = $query->execute();

## Joins

To join against another table, use the join(), innerJoin(), leftJoin(), or rightJoin() methods, like so:

$query = db\_select('node', 'n');

$table\_alias = $query->join('users', 'u', 'n.uid = u.uid AND u.uid = :uid', array(':uid' => 5));

The above directive will add an INNER JOIN (the default join type) against the "user" table, which will get an alias of "u". The join will be ON the condition " n.uid = u.uid AND u.uid = :uid", where :uid has a value of 5

$query = db\_select('node', 'n');

$myselect = db\_select('mytable')

->fields('mytable')

->condition('myfield', 'myvalue');

$alias = $query->join($myselect, 'myalias', 'n.nid = myalias.nid');

$query = db\_select('node', 'n');

$query->join('field\_data\_body', 'b', 'n.nid = b.entity\_id');

$query

->fields('n', array('nid', 'title'))

->condition('n.type', 'page')

->condition('n.status', '1')

->orderBy('n.created', 'DESC')

->addTag('node\_access');

## Ordering

To add an order by clause to a dynamic query, use the chainable orderBy() method:

$query->orderBy('title', 'DESC')

->orderBy('node.created', 'ASC');

The above code will instruct the query to sort by the title field in descending order and then the created field in ascending order. The second parameter may be either "ASC" or "DESC" for ascending or descending, respectively, and defaults to "ASC". The field name here should be the alias created by the addField() or addExpression() methods, so you may want to use the return value from those methods here to guarantee the correct alias is used. To order by multiple fields, simply call orderBy() multiple times in the order desired.

## Random ordering

Random ordering of queries requires slightly different syntax on different databases. Therefore, that is best handled by a dynamic query.

To indicate that a given query should order randomly, call the orderRandom() method on it.

$query->orderRandom();

Note that orderRandom() is chainable, and stackable with orderBy(). That is, it is safe to do something like the following:

$query->orderBy('term')->orderRandom()->execute();

The above would order first by the "term" field of the query and then, for records that have the same term, order randomly.

# Query alteration (tagging)

An important feature of dynamic select queries is the ability of other modules to alter them on the fly. That allows other modules to inject their own restrictions into the query, either to alter a module's behavior or to apply runtime restrictions on the query, such as node access restrictions. There are three components to query alteration: **tagging, meta data, and hook\_query\_alter().**

Only queries that have been tagged by the original creator can be altered later.

## Tagging

Any dynamic Select query may be "tagged" with one or more strings. These tags serve to identify the type of query it is, which in turn allows alter hooks to determine if they need to take action. Tags should be an alphanumeric lowercase string, following the same rules as a PHP variable. (That is, letters, numbers, and underscores only and must begin with a letter.) To add a tag to a query, use the addTag() method:

$query->addTag('node\_access');

To determine if a given query object has been tagged with a given tag, there are three methods available:

// TRUE if this query object has this tag.

$query->hasTag('example');

// TRUE if this query object has every single one of the specified tags.

$query->hasAllTags('example1', 'example2');

// TRUE if this query object has at least one of the specified tags.

$query->hasAnyTag('example1', 'example2');

**node\_access**

This query should have node access restrictions placed on it; all queries that retrieve a list of nodes (or node IDs) for display to users should have this tag. However, note that when the Node module alters queries with this tag, it does not check the published/unpublished status of nodes, so your base query is responsible for ensuring that unpublished nodes are not displayed to inappropriate users.

**entity\_field\_access**

This query should have entity field access restrictions placed on it.

**translatable**

This query should have translatable columns.

**term\_access**

This query should have taxonomy term-based restrictions placed on it; all queries that retrieve a list of taxonomy terms for display to users should have this tag.

**views**

This query is generated by the views module.

**views\_*<view\_name>***

The name of the view creating this query.

## Meta data

Queries may also have meta data attached to them to provide additional context to alter hooks. Meta data may be any PHP variable, and is keyed by a string.

$node = node\_load($nid);

// ... Create a $query object here.

$query->addMetaData('node', $node);

Meta data has no intrinsic meaning, and on its own has no effect on the query object. It exists only to provide additional information to alter hooks, and generally only apply when the query has certain tags.

To access a given piece of meta data on a query, use the getMetaData() method.

$node = $query->getMetaData('node');

If no meta data has been assigned with that key, NULL will be returned.

## hook\_query\_alter()

**Neither tagging nor meta data do anything on their own. Both exist solely to provide information to**[**hook\_query\_alter()**](https://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook_query_alter/7)**, which can take virtually any action on a Select query.**

All Dynamic Select query objects are passed through [hook\_query\_alter()](https://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook_query_alter/7) by the execute() method, immediately before the query string is compiled. That gives modules the opportunity to manipulate the query as desired. hook\_query\_alter() accepts a single parameter: the Select query object itself.

/\*\*

\* Implementation of hook\_query\_alter().

\*/

function example\_query\_alter(QueryAlterableInterface $query) {

// ...

}

There is also a tag-specific alter hook, hook\_query\_TAG\_NAME\_alter(), that is called for every tag on a given Select query, after the generic one has been called. The following example is called for queries that have the tag 'sort\_by\_weight':

/\*\*

\* implements hook query alter to allow ordering by weight

\* @param QueryAlterableInterface $query

\*/

function MYMODULE\_query\_sort\_by\_weight\_alter(QueryAlterableInterface $query) {

$query->join('weight\_weights', 'w', 'node.nid = w.entity\_id');

$query->fields('w', array('weight'));

$query->orderBy('w.weight', 'ASC');

}

There are two important observations to be made regarding hook\_query\_alter().

* The $query parameter is not passed by reference. Because it is an object, the object will not be duplicated anyway due to the way PHP 5 and later handles objects. Passing by reference is therefore unnecessary. The alter hook also has no return value.
* The parameter type is explicitly specified as QueryAlterableInterface. While not strictly necessary, explicitly specifying the parameter type provides slightly better runtime protection against passing in the wrong type of variable. The type is also specified as QueryAlterableInterface rather than simply SelectQuery to better provide forward compatibility.

The alter hook may take any action on the query object it wishes, except executing the query again as that would result in an infinite loop. The alter hook may make use of the tags and meta data associated with a query to determine what if any action to take. Module developers may either call additional methods on the query object as listed above to add additional fields, joins, conditionals, etc. to the query, or may request access to the query object's internal data structures to manipulate them directly. The former is preferred for adding new information to a query while the latter allows the alter hook to remove information from a query or manipulate instructions already queued up.

$fields =& $query->getFields();

$expressions =& $query->getExpressions();

$tables =& $query->getTables();

$order =& $query->getOrderBy();

$where =& $query->conditions();

$having =& $query->havingConditions();

It is important to note that all of the above must be returned by reference (=&) so that the alter hook is accessing the same data structure as the object. All of the above methods return an array, the general structure of which is documented in the inline documentation of SelectQuery in includes/database/select.inc.

For example, to remove an existing sort order:

**$order =& $query->getOrderBy();**

**unset($order['n.nid']);**

# Ranges and limits

Queries may also be restricted to a certain subset of the records found. In general this is known as a "range query". In MySQL, this is implemented using the LIMIT clause. To limit the range of a query, use the range() method, which takes two arguments: the first specifies offset and the second specifies limit

In most cases we want "the first *n* records". To do that, pass 0 as the first argument and *n* as the second.

// Limit the result to 10 records

// where 0 is offset and 10 is limit

$query->range(0, 10);

The following example will instruct the result set to start at the 6th record found (the count starts at 0) rather than the first, and to return only 10 records.

$query->range(5, 10);

Calling the range() method a second time will overwrite previous values. Calling it with no parameters will remove all range restrictions on the query.

## Table sorting

To produce a result table which can be sorted by any column, use the TableSort extender and then add the table header. Note that an extender does return a new query object that you need to use from that point on.

$query = $query

->extend('TableSort')

->orderByHeader($header);

## Unions

To combine the results from multiple SELECT queries into a single result you use UNION like this:

$table2 = db\_select('table2', 't2')

->fields('t2', array('column3', 'column4'));

$query = db\_select('table1', 't1')

->fields('t1', array('column1', 'column2'))

->union($table2);

To order the combined result the individual SELECT queries must be parenthesized which currently is not supported (see #1145076: UNION queries don't support ORDER BY clauses). A workaround is to put the UNIONed queries into a subquery and applying the ORDER BY clause to the combined result:

$table1 = db\_select('table1', 't1')

->fields('t1', array('column1', 'column2'));

$table2 = db\_select('table2', 't2')

->fields('t2', array('column3', 'column4'));

$query = Database::getConnection()

->select($table1->union($table2))

->fields(NULL, array('column1', 'column2'))

->orderBy('column1')

->orderBy('column2');

---End of Dynamic query

# Result sets

A Select query will always return a result set object of zero or more records. There are several ways to then retrieve data from that result set, depending on the use case.

The most common case is to iterate over the result set with a foreach() loop.  
  
$result = db\_query("SELECT nid, title FROM {node}");

foreach ($result as $record) {

// Do something with each $record

$node = node\_load($record->nid);

}

Depending on what the results are needed for, however, there are a number of other ways to retrieve the records.

To explicitly fetch the next record, use:

$record = $result->fetch(); // Use the default fetch mode.

$record = $result->fetchObject(); // Fetch as a stdClass object.

$record = $result->fetchAssoc(); // Fetch as an associative array.

If there is no next record, FALSE will be returned. fetch() should generally be avoided in favor of fetchObject() and fetchAssoc(), as the latter are more self-documenting. If you need to use some other PDO-supported fetch mode, then use fetch().

To fetch just a single field out of the result set, use:

$record = $result->fetchField($column\_index);

The default value of $column\_index is 0, for the first field.

To count the number of rows returned from a DELETE, INSERT or UPDATE statement use:

$number\_of\_rows = $result->rowCount();

To count the number of rows returned from a SELECT statement use:

$number\_of\_rows = db\_select('node')->countQuery()->execute()->fetchField();

To fetch all records at once into a single array, use one of the following:

// Retrieve all records into an indexed array of stdClass objects.

$result->fetchAll();

// Retrieve all records into an associative array keyed by the field in the result specified.

$result->fetchAllAssoc($field);

// Retrieve a 2-column result set as an associative array of field 0 => field 1.

$result->fetchAllKeyed();

// You can also specify which two fields to use by specifying the column numbers for each field

$result->fetchAllKeyed(0,2); // would be field 0 => field 2

$result->fetchAllKeyed(1,0); // would be field 1 => field 0

// If you need an array where keys and values contain the same field (e.g. for creating a 'checkboxes' form element), the following is a perfectly valid method:

$result->fetchAllKeyed(0,0); // would be field 0 => field 0, e.g. [article] => [article]

// Retrieve a 1-column result set as one single array.

$result->fetchCol();

// Column number can be specified otherwise defaults to first column

$result->fetchCol($column\_index);

// Get an associative array of nids to titles.

$nodes = db\_query("SELECT nid, title FROM {node}")->fetchAllKeyed();

// Get a single record out of the database.

$node = db\_query("SELECT \* FROM {node} WHERE nid = :nid", array(':nid' => $nid))->fetchObject();

// Get a single value out of the database.

$title = db\_query("SELECT title FROM {node} WHERE nid = :nid", array(':nid' => $nid))->fetchField();

If what you want is a simple array like array(1, 2, 3, 4, 5) you will have to settle for something more like array(1=>1, 2=>2, 3=>3, 4=>4, 5=>5). You can get this by using

$nids = db\_query("SELECT nid FROM {node}")->fetchAllKeyed(0,0);

# Insert queries

Insert queries must always use a query builder object. Certain databases require special handling for LOB (Large OBject, such as TEXT in MySQL) and BLOB (Binary Large OBject) fields, so a layer of abstraction is required to allow individual database drivers to implement whatever special handling they require.

Insert queries are started using the db\_insert() function as follows:

$query = db\_insert('node', $options);

That creates an insert query object that will insert one or more records to the node table. Note that braces are not required around the table name as the query builder will handle that automatically.

The insert query object uses a fluent API. That is, all methods (except execute()) return the query object itself allowing method calls to be chained. In many cases, that means the query object will not need to be saved to a variable at all.

The insert query object supports a number of different usage patterns to support different needs. In general, the workflow consists of specifying the fields that the query will insert into, specifying the values the query will insert for those fields, and executing the query. The most common recommended usage patterns are listed below.

## Compact form

$nid = db\_insert('node')

->fields(array(

'title' => 'Example',

'uid' => 1,

'created' => REQUEST\_TIME,

))

->execute();

This will result in the equivalent of the following query:

INSERT INTO {node} (title, uid, created) VALUES ('Example', 1, 1221717405);

$nid = db\_insert('node')

->fields(array('title', 'uid', 'created'))

->values(array(

'title' => 'Example',

'uid' => 1,

'created' => REQUEST\_TIME,

))

->execute();

## Multi-insert form

The Insert query object may also take multiple value sets. That is, values() may be called multiple times to enqueue several insert statements together.

$values = array(

array(

'title' => 'Example',

'uid' => 1,

'created' => REQUEST\_TIME,

),

array(

'title' => 'Example 2',

'uid' => 1,

'created' => REQUEST\_TIME,

),

array(

'title' => 'Example 3',

'uid' => 2,

'created' => REQUEST\_TIME,

),

);

$query = db\_insert('node')->fields(array('title', 'uid', 'created'));

foreach ($values as $record) {

$query->values($record);

}

$query->execute();

The above example will execute three insert statements together as a single unit, using the most efficient method for the particular database driver in use.

## Inserting based on the results of a select query

If you want to populate a table with results from other tables, you either need to SELECT from the source tables, iterate over the data in PHP and insert it into the new table, or you can do an INSERT INTO...SELECT FROM query in which every record which is returned from the SELECT query gets fed into the INSERT query.

// Build the SELECT query.

$query = db\_select('node', 'n');

// Join to the users table.

$query->join('users', 'u', 'n.uid = u.uid');

// Add the fields we want.

$query->addField('n','nid');

$query->addField('u','name');

// Add a condition to only get page nodes.

$query->condition('type', 'page');

// Perform the insert.

db\_insert('mytable')

->from($query)

->execute();

## Default values

To explicitly tell the database to use the default value for a given field, there is a useDefaults() method.

$query->useDefaults(array('field1', 'field2'));

## db\_insert or db\_query

What are the difference between db\_insert and db\_query?

db\_insert has each column specified as a separate entry in the fields array and the code can clean each column value. db\_query has an SQL string with no way of checking individual columns. If you use db\_query with placeholders, the code can check the column values but placeholders are just an option, there is no way to ensure your SQL does not contain values not passed through placeholders.

**db\_insert** passes the request through a set of hooks to let other modules check and modify your requests. **This is the right way to work with other modules**. db\_query is slightly faster because db\_query does not pass the request through the hooks. You might save processing time but your code will not let other modules help your code.

db\_insert is more likely to work with other databases and future versions of Drupal.

# Update queries

Update queries must always use a query builder object. Certain databases require special handling for LOB (Large OBject, such as TEXT on MySQL) and BLOB (Binary Large OBject) fields, so a layer of abstraction is required to allow individual database drivers to implement whatever special handling they require.

Update queries are started using the db\_update() function as follows:

$query = db\_update('node', $options);

This creates an update query object that will modify one or more records to the node table. Note that braces are not required around the table name as the query builder will handle that automatically.

The update query object uses a fluent API. That is, all methods (except execute()) return the query object itself allowing method calls to be chained. In many cases, that means the query object will not need to be saved to a variable at all.

Update queries are conceptually simple, consisting of a set of key/value pairs to set and a WHERE clause. The full structure of the WHERE clause is detailed in the section on Conditional clauses, and will only be touched on here.

/\* This is a horrible example as node.status is pulled from node\_revision.status table as well, updating it here will do nothing. \*/

$num\_updated = db\_update('node')

->fields(array(

'uid' => 5,

'status' => 1,

))

->condition('created', REQUEST\_TIME - 3600, '>=')

->execute();

The above query will update all records in the node table created within the last hour and set their uid field to 5 and status field to 1.

The above example is equivalent to the following query:

UPDATE {node} SET uid=5, status=1 WHERE created >= 1221717405;

The execute() method will return the number of rows affected by the query.

$query = db\_update('mytable');

// Conditions etc.

$affected\_rows = $query->execute();

$query = db\_update('block')

->condition('module', 'my\_module')

->where(

'SUBSTR(delta, 1, 14) <> :module\_key',

array('module\_key' => 'my\_module-key\_')

)

->expression('delta', "REPLACE(delta, 'my\_module-other\_', 'my\_module-thing\_')")

->execute();

$query = db\_update('block')

->condition('module', 'my\_module')

->condition('SUBSTR(delta, 1, 14)', 'my\_module-key\_', '<>') // causes error.

->expression('delta', "REPLACE(delta, 'my\_module-other\_', 'my\_module-thing\_')")

->execute();

# Delete queries

Delete queries must always use a query builder object. They are started using the db\_delete() function as follows:

$query = db\_delete('node', $options);

That creates a delete query object that will delete records from the node table. Note that braces are not required around the table name as the query builder will handle that automatically.

The delete query object uses a fluent API. That is, all methods (except execute()) return the query object itself allowing method calls to be chained. In many cases, this means the query object will not need to be saved to a variable at all.

Delete queries are conceptually very simple, consisting of only a WHERE clause. The full structure of the WHERE clause is detailed in the section on [Conditional clauses](http://drupal.org/node/310086), and will only be touched on here.

A full Delete query will take the following form:

$num\_deleted = db\_delete('node')

->condition('nid', 5)

->execute();

The above query will delete all rows from the {node} table where the nid column is 5. It is equivalent to the following query:

DELETE FROM {node} WHERE nid=5;

The execute() method will return the number of records that were deleted as a result of the query.

# Merge queries using db\_merge

he Merge query builder in Drupal abstracts the concept of a Merge query out into a structured object that can be compiled down to the appropriate syntax for each database. These are sometimes called "UPSERT" queries, a combination of UPDATE and INSERT.

In the general sense, a Merge query is a combination of an Insert query and an Update query. If a given condition is met, such as a row with a given primary key already existing, then an Update query is run. If not, an Insert query is run. In the most common case, it is equivalent to:

if (db\_query("SELECT COUNT(\*) FROM {example} WHERE id = :id", array(':id' => $id))->fetchField()) {

// Run an update using WHERE id = $id

}

else {

// Run an insert, inserting $id for id

}

db\_merge('example')

->key(array('name' => $name))

->fields(array(

'field1' => $value1,

'field2' => $value2,

))

->execute();

In the above example, we instruct the query to operate on the "example" table. We then specify one key field, 'name', with a value of $name. We then specify an array of values to set.

If a row already exists in which the field "name" has the value $name, then fields field1 and field2 will be set to the corresponding values in that existing row. If such a row does not exist, one will be created in which name has the value $name, field1 has the value $value1, and field2 has the value $value2. Thus at the end of the query, the end result is the same regardless of whether or not the row already existed.

## Conditional set

In some cases, you may want to set values differently depending on whether or not the record, as identified by the key() fields, already existed. There are two ways to do that.

db\_merge('example')

->insertFields(array(

'field1' => $value1,

'field2' => $value2,

))

->updateFields(array(

'field1' => $alternate1,

))

->key(array('name' => $name))

->execute();

The above example will behave the same as the first, except that if the record already exists and we are updating it, field1 will be set to $alternate1 instead of $value1 and field2 will not be affected. The updateFields() method accepts either a single associative array of values or two parallel numeric arrays, one of fields, one of values, that must be in the same order.

db\_merge('example')

->key(array('name' => $name))

->fields(array(

'field1' => $value1,

'field2' => $value2,

))

->expression('field1', 'field1 + :inc', array(':inc' => 1))

->execute();

In this example, if the record already exists then field1 will be set to its current value plus 1. That makes it very useful for "counter queries", where you want to increment some counter in the database every time a certain event happens. field2 will still be set to the same value regardless of whether the record exists or not.

Note that expression() may be called multiple times, once for each field that should be set to an expression if the record already exists. The first parameter is the field, the second is an SQL fragment indicating the expression the field should be set to, and the optional third parameter is an array of placeholder values to insert into the expression.

There is also no requirement that a field used in expression() be already present in fields().

## Precedence

Given the above API it is quite possible to define queries that do not logically make sense, say if a field is set to both be ignored and to be set to an expression if the record already exists. To minimize potential errors, the following rules apply:

* If a field is set to an expression(), that takes priority over updateFields().
* If values are specified in updateFields(), only those fields will be altered if the record already exists. Fields not specified in updateFields() will not be affected.

Note that it may still be possible to define queries that do not make sense. It is up to the developer to ensure that a nonsensical query is not specified as the behavior in that case is undefined.

# Error handling

The Database API throws exceptions on error, which can be picked up by wrapping your database operations in try {} catch() {} blocks, as shown in this example:

// The transaction opens here.

$txn = db\_transaction();

try {

$id = db\_insert('example')

->fields(array(

'field1' => 'mystring',

'field2' => 5,

))

->execute();

my\_other\_function($id);

return $id;

}

catch (Exception $e) {

// Something went wrong somewhere, so roll back now.

$txn->rollback();

// Log the exception to watchdog.

watchdog\_exception('type', $e);

}

# Transactions

Drupal also supports transactions, including a transparent fallback for databases that do not support transactions. However, transactions can get quite complicated when you try and start two transactions at the same time. The behavior in that case also varies between databases.

To start a new transaction, simply call $transaction = db\_transaction(); in your own code. The transaction will remain open for as long as the variable $transaction remains in scope. When $transaction is destroyed, the transaction will be committed. If your transaction is nested inside of another then Drupal will track each transaction and only commit the outer-most transaction when the last transaction object goes out out of scope, that is, all relevant queries completed successfully.

**You must assign the return value of db\_transaction() to a variable**

If you call the function without assigning the return value to a variable, your transaction will commit instantly, making it useless

function my\_transaction\_function() {

// The transaction opens here.

**$transaction = db\_transaction();**

try {

$id = db\_insert('example')

->fields(array(

'field1' => 'mystring',

'field2' => 5,

))

->execute();

my\_other\_function($id);

return $id;

}

catch (Exception $e) {

**$transaction->rollback();**

watchdog\_exception('my\_type', $e);

}

**// $transaction goes out of scope here. Unless it was rolled back, it**

**// gets automatically commited here.**

}

function my\_other\_function($id) {

// The transaction is still open here.

if ($id % 2 == 0) {

db\_update('example')

->condition('id', $id)

->fields(array('field2' => 10))

->execute();

}

}

# Chaining

There is a lot of talk about chaining database api functions together, for example:

<?php

$result = db\_select('mytable')

->fields('mytable')

->condition('myfield', 'myvalue')

->execute();

?>

For a function to be chain-able its return value must be the query object itself for a function acting on the query object. You may also append a result set function after execute() such as fetchField() as in this example:

**Functions that can be chained:**

* addMetaData()
* addTag()
* comment()
* condition()
* countQuery()
* distinct()
* exists()
* fields()
* forUpdate()
* groupBy()
* having()
* havingCondition()
* isNotNull()
* isNull()
* notExists()
* orderBy()
* orderRandom()
* range()
* union()
* where()
* execute()

**Functions that cannot be chained:**

* addExpression()
* addField()
* addJoin()
* extend()
* innerJoin()
* join()
* leftJoin()
* rightJoin()

# Functions and operators

Drupal's database layer does not provide cross-database abstraction of SQL functions. For portability across supported database engines your code should only use functions known to be part of the ANSI standard and supported across all databases that Drupal supports. The following is a still-incomplete list. The form used here is recommended as other syntax variants may not work on all databases.

Note that the database layer does not whitelist operators, so you may pass a non-standard function, such as REPLACE(), and it will work for databases that support the syntax.

**Logical operators**

AND

OR

NOT

**Comparison operators**<  
>  
<=  
>=  
<>  
LIKE

## String functions and operators

### CONCAT(string1, string2)

### SUBSTRING(string, from, length)

### SUBSTRING\_INDEX(string, delimiter, count)

### LENGTH(string)

## Mathematical functions and operators

### GREATEST(num1, num2)

### POW(num1, num2)

### LOG(base, value)

## Aggregation functions

### COUNT(expression)

### SUM(expression)

### AVG(expression)

### MIN(expression)

### MAX(expression)