# Query Builder Class

CodeIgniter gives you access to a Query Builder class. This pattern allows information to be retrieved, inserted, and updated in your database with minimal scripting. In some cases only one or two lines of code are necessary to perform a database action. CodeIgniter does not require that each database table be its own class file. It instead provides a more simplified interface.

Beyond simplicity, a major benefit to using the Query Builder features is that it allows you to create database independent applications, since the query syntax is generated by each database adapter. It also allows for safer queries, since the values are escaped automatically by the system.

**Note**

If you intend to write your own queries you can disable this class in your database config file, allowing the core database library and adapter to utilize fewer resources.

## Selecting Data

The following functions allow you to build SQL **SELECT** statements.

**$this->db->get()**

Runs the selection query and returns the result. Can be used by itself to retrieve all records from a table:

```
$query = $this->db->get('mytable');  // Produces: SELECT * FROM mytable
```

The second and third parameters enable you to set a limit and offset clause:

```
$query = $this->db->get('mytable', 10, 20);

// Executes: SELECT * FROM mytable LIMIT 20, 10
// (in MySQL. Other databases have slightly different syntax)
```

You'll notice that the above function is assigned to a variable named $query, which can be used to show the results:

```
$query = $this->db->get('mytable');

foreach ($query->result() as $row)
{
        echo $row->title;
}
```

Please visit the result functions page for a full discussion regarding result generation.

**$this->db->get_compiled_select()**

Compiles the selection query just like **$this->db->get()** but does not *run* the query. This method simply returns the SQL query as a string.

Example:

```
$sql = $this->db->get_compiled_select('mytable');
echo $sql;

// Prints string: SELECT * FROM mytable
```

The second parameter enables you to set whether or not the query builder query will be reset (by default it will be reset, just like when using *$this->db->get()*):

```php
echo $this->db->limit(10,20)->get_compiled_select('mytable', FALSE);

// Prints string: SELECT * FROM mytable LIMIT 20, 10
// (in MySQL. Other databases have slightly different syntax)

echo $this->db->select('title, content, date')->get_compiled_select();

// Prints string: SELECT title, content, date FROM mytable LIMIT 20, 10
```

The key thing to notice in the above example is that the second query did not utilize **$this->db->from()** and did not pass a table name into the first parameter. The reason for this outcome is because the query has not been executed using **$this->db->get()** which resets values or reset directly using **$this->db->reset_query()**.

### $this->db->get_where()

Identical to the above function except that it permits you to add a "where" clause in the second parameter, instead of using the db->where() function:

```php
$query = $this->db->get_where('mytable', array('id' => $id), $limit, $offset);
```

Please read the about the where function below for more information.

**Note**

get_where() was formerly known as getwhere(), which has been removed

### $this->db->select()

Permits you to write the SELECT portion of your query:

```
$this->db->select('title, content, date');
$query = $this->db->get('mytable');

// Executes: SELECT title, content, date FROM mytable
```

**Note**

If you are selecting all (*) from a table you do not need to use this function. When omitted, CodeIgniter assumes that you wish to select all fields and automatically adds 'SELECT *'.

`$this->db->select()` accepts an optional second parameter. If you set it to FALSE, CodeIgniter will not try to protect your field or table names. This is useful if you need a compound select statement where automatic escaping of fields may break them.

```
$this->db->select('(SELECT SUM(payments.amount) FROM payments WHERE payments.invoice_id
$query = $this->db->get('mytable');
```

### $this->db->select_max()

Writes a `SELECT MAX(field)` portion for your query. You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_max('age');
$query = $this->db->get('members');  // Produces: SELECT MAX(age) as age FROM members
```

```
$this->db->select_max('age', 'member_age');
$query = $this->db->get('members'); // Produces: SELECT MAX(age) as member_age FROM mem
```

### $this->db->select_min()

Writes a “SELECT MIN(field)” portion for your query. As with select_max(), You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_min('age');
$query = $this->db->get('members'); // Produces: SELECT MIN(age) as age FROM members
```

### $this->db->select_avg()

Writes a “SELECT AVG(field)” portion for your query. As with select_max(), You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_avg('age');
$query = $this->db->get('members'); // Produces: SELECT AVG(age) as age FROM members
```

### $this->db->select_sum()

Writes a “SELECT SUM(field)” portion for your query. As with select_max(), You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_sum('age');
$query = $this->db->get('members'); // Produces: SELECT SUM(age) as age FROM members
```

### $this->db->from()

Permits you to write the FROM portion of your query:

```
$this->db->select('title, content, date');
$this->db->from('mytable');
$query = $this->db->get();  // Produces: SELECT title, content, date FROM mytable
```

**Note**

As shown earlier, the FROM portion of your query can be specified in the $this->db->get() function, so use whichever method you prefer.

### $this->db->join()

Permits you to write the JOIN portion of your query:

```
$this->db->select('*');
$this->db->from('blogs');
$this->db->join('comments', 'comments.id = blogs.id');
$query = $this->db->get();

// Produces:
// SELECT * FROM blogs JOIN comments ON comments.id = blogs.id
```

Multiple function calls can be made if you need several joins in one query.

If you need a specific type of JOIN you can specify it via the third parameter of the function. Options are: left, right, outer, inner, left outer, and right outer.

```
$this->db->join('comments', 'comments.id = blogs.id', 'left');
// Produces: LEFT JOIN comments ON comments.id = blogs.id
```

## Looking for Specific Data

**$this->db->where()**

This function enables you to set **WHERE** clauses using one of four methods:

**Note**

All values passed to this function are escaped automatically, producing safer queries.

1. **Simple key/value method:**

    ```
    $this->db->where('name', $name); // Produces: WHERE name = 'Joe'
    ```

    Notice that the equal sign is added for you.

    If you use multiple function calls they will be chained together with AND between them:

```
$this->db->where('name', $name);
$this->db->where('title', $title);
$this->db->where('status', $status);
// WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

2. **Custom key/value method:**

You can include an operator in the first parameter in order to control the comparison:

```
$this->db->where('name !=', $name);
$this->db->where('id <', $id); // Produces: WHERE name != 'Joe' AND id < 45
```

3. **Associative array method:**

```
$array = array('name' => $name, 'title' => $title, 'status' => $status);
$this->db->where($array);
// Produces: WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

You can include your own operators using this method as well:

```
$array = array('name !=' => $name, 'id <' => $id, 'date >' => $date);
$this->db->where($array);
```

4. **Custom string:**

You can write your own clauses manually:

```
$where = "name='Joe' AND status='boss' OR status='active'";
$this->db->where($where);
```

`$this->db->where()` accepts an optional third parameter. If you set it to FALSE, CodeIgniter will not try to protect your field or table names.

```
$this->db->where('MATCH (field) AGAINST ("value")', NULL, FALSE);
```

### $this->db->or_where()

This function is identical to the one above, except that multiple instances are joined by OR:

```
$this->db->where('name !=', $name);
$this->db->or_where('id >', $id);  // Produces: WHERE name != 'Joe' OR id > 50
```

**Note**

or_where() was formerly known as orwhere(), which has been removed.

### $this->db->where_in()

Generates a WHERE field IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->where_in('username', $names);
// Produces: WHERE username IN ('Frank', 'Todd', 'James')
```

### $this->db->or_where_in()

Generates a WHERE field IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->or_where_in('username', $names);
// Produces: OR username IN ('Frank', 'Todd', 'James')
```

**$this->db->where_not_in()**

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->where_not_in('username', $names);
// Produces: WHERE username NOT IN ('Frank', 'Todd', 'James')
```

**$this->db->or_where_not_in()**

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->or_where_not_in('username', $names);
// Produces: OR username NOT IN ('Frank', 'Todd', 'James')
```

## Looking for Similar Data

**$this->db->like()**

This method enables you to generate **LIKE** clauses, useful for doing searches.

**Note**

All values passed to this method are escaped automatically.

1. **Simple key/value method:**

```
$this->db->like('title', 'match');
// Produces: WHERE `title` LIKE '%match%' ESCAPE '!'
```

If you use multiple method calls they will be chained together with AND between them:

```
$this->db->like('title', 'match');
$this->db->like('body', 'match');
// WHERE `title` LIKE '%match%' ESCAPE '!' AND  `body` LIKE '%match% ESCAPE '!'
```

If you want to control where the wildcard (%) is placed, you can use an optional third argument. Your options are 'before', 'after' and 'both' (which is the default).

```
$this->db->like('title', 'match', 'before');     // Produces: WHERE `title` LIKE
$this->db->like('title', 'match', 'after');      // Produces: WHERE `title` LIKE
$this->db->like('title', 'match', 'both');       // Produces: WHERE `title` LIKE
```

2. **Associative array method:**

```
$array = array('title' => $match, 'page1' => $match, 'page2' => $match);
$this->db->like($array);
```

```
// WHERE `title` LIKE '%match%' ESCAPE '!' AND  `page1` LIKE '%match%' ESCAPE '!
```

### $this->db->or_like()

This method is identical to the one above, except that multiple instances are joined by OR:

```
$this->db->like('title', 'match'); $this->db->or_like('body', $match);
// WHERE `title` LIKE '%match%' ESCAPE '!' OR  `body` LIKE '%match%' ESCAPE '!'
```

**Note**

`or_like()` was formerly known as `orlike()`, which has been removed.

### $this->db->not_like()

This method is identical to `like()`, except that it generates NOT LIKE statements:

```
$this->db->not_like('title', 'match');  // WHERE `title` NOT LIKE '%match% ESCAPE '!'
```

### $this->db->or_not_like()

This method is identical to `not_like()`, except that multiple instances are joined by OR:

```
$this->db->like('title', 'match');
$this->db->or_not_like('body', 'match');
```

```
// WHERE `title` LIKE '%match% OR  `body` NOT LIKE '%match%' ESCAPE '!'
```

### $this->db->group_by()

Permits you to write the GROUP BY portion of your query:

```
$this->db->group_by("title"); // Produces: GROUP BY title
```

You can also pass an array of multiple values as well:

```
$this->db->group_by(array("title", "date"));  // Produces: GROUP BY title, date
```

**Note**

group_by() was formerly known as groupby(), which has been removed.

### $this->db->distinct()

Adds the “DISTINCT” keyword to a query

```
$this->db->distinct();
$this->db->get('table'); // Produces: SELECT DISTINCT * FROM table
```

### $this->db->having()

Permits you to write the HAVING portion of your query. There are 2 possible syntaxes, 1 argument or 2:

```
$this->db->having('user_id = 45');  // Produces: HAVING user_id = 45
$this->db->having('user_id',  45);  // Produces: HAVING user_id = 45
```

You can also pass an array of multiple values as well:

```
$this->db->having(array('title =' => 'My Title', 'id <' => $id));
// Produces: HAVING title = 'My Title', id < 45
```

If you are using a database that CodeIgniter escapes queries for, you can prevent escaping content by passing an optional third argument, and setting it to FALSE.

```
$this->db->having('user_id',  45);  // Produces: HAVING `user_id` = 45 in some database
$this->db->having('user_id',  45, FALSE);  // Produces: HAVING user_id = 45
```

**$this->db->or_having()**

Identical to having(), only separates multiple clauses with “OR”.

## Ordering results

**$this->db->order_by()**

Lets you set an ORDER BY clause.

The first parameter contains the name of the column you would like to order by.

The second parameter lets you set the direction of the result. Options are **ASC**, **DESC** AND **RANDOM**.

```
$this->db->order_by('title', 'DESC');
// Produces: ORDER BY `title` DESC
```

You can also pass your own string in the first parameter:

```
$this->db->order_by('title DESC, name ASC');
// Produces: ORDER BY `title` DESC, `name` ASC
```

Or multiple function calls can be made if you need multiple fields.

```
$this->db->order_by('title', 'DESC');
$this->db->order_by('name', 'ASC');
// Produces: ORDER BY `title` DESC, `name` ASC
```

If you choose the **RANDOM** direction option, then the first parameters will be ignored, unless you specify a numeric seed value.

```
$this->db->order_by('title', 'RANDOM');
// Produces: ORDER BY RAND()
```

```
$this->db->order_by(42, 'RANDOM');
// Produces: ORDER BY RAND(42)
```

**Note**

order_by() was formerly known as orderby(), which has been removed.

**Note**

Random ordering is not currently supported in Oracle and will default to ASC instead.

## Limiting or Counting Results

### $this->db->limit()

Lets you limit the number of rows you would like returned by the query:

```
$this->db->limit(10);  // Produces: LIMIT 10
```

The second parameter lets you set a result offset.

```
$this->db->limit(10, 20);  // Produces: LIMIT 20, 10 (in MySQL.  Other databases have s
```

### $this->db->count_all_results()

Permits you to determine the number of rows in a particular Active Record query. Queries will accept Query Builder restrictors such as `where()`, `or_where()`, `like()`, `or_like()`, etc. Example:

```
echo $this->db->count_all_results('my_table');  // Produces an integer, like 25
$this->db->like('title', 'match');
$this->db->from('my_table');
echo $this->db->count_all_results(); // Produces an integer, like 17
```

However, this method also resets any field values that you may have passed to `select()`. If you need to keep them, you can pass `FALSE` as the second parameter:

```
echo $this->db->count_all_results('my_table', FALSE);
```

**$this->db->count_all()**

Permits you to determine the number of rows in a particular table. Submit the table name in the first parameter. Example:

```
echo $this->db->count_all('my_table');  // Produces an integer, like 25
```

## Query grouping

Query grouping allows you to create groups of WHERE clauses by enclosing them in parentheses. This will allow you to create queries with complex WHERE clauses. Nested groups are supported. Example:

```php
$this->db->select('*')->from('my_table')
        ->group_start()
                ->where('a', 'a')
                ->or_group_start()
                        ->where('b', 'b')
                        ->where('c', 'c')
                ->group_end()
        ->group_end()
        ->where('d', 'd')
->get();

// Generates:
// SELECT * FROM (`my_table`) WHERE ( `a` = 'a' OR ( `b` = 'b' AND `c` = 'c' ) ) AND `d
```

**Note**

groups need to be balanced, make sure every group_start() is matched by a group_end().

**$this->db->group_start()**

Starts a new group by adding an opening parenthesis to the WHERE clause of the query.

**$this->db->or_group_start()**

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'OR'.

**$this->db->not_group_start()**

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'NOT'.

**$this->db->or_not_group_start()**

Starts a new group by adding an opening parenthesis to the WHERE clause of the query, prefixing it with 'OR NOT'.

**$this->db->group_end()**

Ends the current group by adding an closing parenthesis to the WHERE clause of the query.

## Inserting Data

**$this->db->insert()**

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
        'title' => 'My title',
        'name' => 'My Name',
        'date' => 'My date'
);

$this->db->insert('mytable', $data);
// Produces: INSERT INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My
```

The first parameter will contain the table name, the second is an associative array of values.

Here is an example using an object:

```
/*
class Myclass {
        public $title = 'My Title';
        public $content = 'My Content';
        public $date = 'My Date';
}
*/

$object = new Myclass;
$this->db->insert('mytable', $object);
// Produces: INSERT INTO mytable (title, content, date) VALUES ('My Title', 'My Content
```

The first parameter will contain the table name, the second is an object.

> **Note**
>
> All values are escaped automatically producing safer queries.

**$this->db->get_compiled_insert()**

Compiles the insertion query just like $this->db->insert() but does not *run* the query. This method simply returns the SQL query as a string.

Example:

```php
$data = array(
        'title' => 'My title',
        'name'  => 'My Name',
        'date'  => 'My date'
);

$sql = $this->db->set($data)->get_compiled_insert('mytable');
echo $sql;

// Produces string: INSERT INTO mytable (`title`, `name`, `date`) VALUES ('My title', '
```

The second parameter enables you to set whether or not the query builder query will be reset (by default it will be–just like $this->db->insert()):

```php
echo $this->db->set('title', 'My Title')->get_compiled_insert('mytable', FALSE);

// Produces string: INSERT INTO mytable (`title`) VALUES ('My Title')

echo $this->db->set('content', 'My Content')->get_compiled_insert();

// Produces string: INSERT INTO mytable (`title`, `content`) VALUES ('My Title', 'My Co
```

The key thing to notice in the above example is that the second query did not utilize *$this->db->from()* nor did it pass a table name into the first parameter. The reason this worked is because the query has not been executed using *$this->db->insert()* which resets values or reset directly using *$this->db->reset_query()*.

**Note**

This method doesn't work for batched inserts.

**$this->db->insert_batch()**

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
        array(
                'title' => 'My title',
                'name' => 'My Name',
                'date' => 'My date'
        ),
        array(
                'title' => 'Another title',
                'name' => 'Another Name',
                'date' => 'Another date'
        )
);

$this->db->insert_batch('mytable', $data);
// Produces: INSERT INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My
```

The first parameter will contain the table name, the second is an associative array of values.

**Note**

All values are escaped automatically producing safer queries.

# Updating Data

**$this->db->replace()**

This method executes a REPLACE statement, which is basically the SQL standard for (optional) DELETE + INSERT, using *PRIMARY* and *UNIQUE* keys as the determining factor. In our case, it will save you from the need to implement complex logics with different combinations of `select()`, `update()`, `delete()` and `insert()` calls.

Example:

```
$data = array(
        'title' => 'My title',
        'name'  => 'My Name',
        'date'  => 'My date'
);

$this->db->replace('table', $data);

// Executes: REPLACE INTO mytable (title, name, date) VALUES ('My title', 'My name', 'M
```

In the above example, if we assume that the *title* field is our primary key, then if a row containing 'My title' as the *title* value, that row will be deleted with our new row data replacing it.

Usage of the `set()` method is also allowed and all fields are automatically escaped, just like with `insert()`.

### $this->db->set()

This function enables you to set values for inserts or updates.

**It can be used instead of passing a data array directly to the insert or update functions:**

```
$this->db->set('name', $name);
$this->db->insert('mytable');  // Produces: INSERT INTO mytable (`name`) VALUES ('{$nam
```

If you use multiple function called they will be assembled properly based on whether you are doing an insert or an update:

```
$this->db->set('name', $name);
$this->db->set('title', $title);
$this->db->set('status', $status);
$this->db->insert('mytable');
```

**set()** will also accept an optional third parameter ( `$escape` ), that will prevent data from being escaped if set to FALSE. To illustrate the difference, here is `set()` used both with and without the escape parameter.

```
$this->db->set('field', 'field+1', FALSE);
$this->db->where('id', 2);
$this->db->update('mytable'); // gives UPDATE mytable SET field = field+1 WHERE id = 2

$this->db->set('field', 'field+1');
$this->db->where('id', 2);
$this->db->update('mytable'); // gives UPDATE `mytable` SET `field` = 'field+1' WHERE `
```

You can also pass an associative array to this function:

```
$array = array(
        'name' => $name,
        'title' => $title,
        'status' => $status
);

$this->db->set($array);
$this->db->insert('mytable');
```

Or an object:

```
/*
class Myclass {
        public $title = 'My Title';
        public $content = 'My Content';
        public $date = 'My Date';
}
*/

$object = new Myclass;
$this->db->set($object);
$this->db->insert('mytable');
```

**$this->db->update()**

Generates an update string and runs the query based on the data you supply. You can pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
        'title' => $title,
        'name' => $name,
        'date' => $date
);

$this->db->where('id', $id);
$this->db->update('mytable', $data);
// Produces:
//
//      UPDATE mytable
//      SET title = '{$title}', name = '{$name}', date = '{$date}'
//      WHERE id = $id
```

Or you can supply an object:

```
/*
class Myclass {
        public $title = 'My Title';
        public $content = 'My Content';
        public $date = 'My Date';
}
*/

$object = new Myclass;
$this->db->where('id', $id);
$this->db->update('mytable', $object);
// Produces:
//
// UPDATE `mytable`
```

```
// SET `title` = '{$title}', `name` = '{$name}', `date` = '{$date}'
// WHERE id = `$id`
```

**Note**

All values are escaped automatically producing safer queries.

You'll notice the use of the $this->db->where() function, enabling you to set the WHERE clause. You can optionally pass this information directly into the update function as a string:

```
$this->db->update('mytable', $data, "id = 4");
```

Or as an array:

```
$this->db->update('mytable', $data, array('id' => $id));
```

You may also use the $this->db->set() function described above when performing updates.

**$this->db->update_batch()**

Generates an update string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
   array(
      'title' => 'My title' ,
```

```
        'name' => 'My Name 2' ,
        'date' => 'My date 2'
    ),
    array(
        'title' => 'Another title' ,
        'name' => 'Another Name 2' ,
        'date' => 'Another date 2'
    )
);

$this->db->update_batch('mytable', $data, 'title');

// Produces:
// UPDATE `mytable` SET `name` = CASE
// WHEN `title` = 'My title' THEN 'My Name 2'
// WHEN `title` = 'Another title' THEN 'Another Name 2'
// ELSE `name` END,
// `date` = CASE
// WHEN `title` = 'My title' THEN 'My date 2'
// WHEN `title` = 'Another title' THEN 'Another date 2'
// ELSE `date` END
// WHERE `title` IN ('My title','Another title')
```

The first parameter will contain the table name, the second is an associative array of values, the third parameter is the where key.

**Note**

All values are escaped automatically producing safer queries.

**Note**

`affected_rows()` won't give you proper results with this method, due to the very nature of how it works. Instead, `update_batch()` returns the number of rows affected.

**$this->db->get_compiled_update()**

This works exactly the same way as `$this->db->get_compiled_insert()` except that it produces an UPDATE SQL string instead of an INSERT SQL string.

For more information view documentation for *$this->db->get_compiled_insert()*.

**Note**

This method doesn't work for batched updates.

## Deleting Data

**$this->db->delete()**

Generates a delete SQL string and runs the query.

```
$this->db->delete('mytable', array('id' => $id));  // Produces: // DELETE FROM mytable
```

The first parameter is the table name, the second is the where clause. You can also use the where() or or_where() functions instead of passing the data to the second parameter of the function:

```
$this->db->where('id', $id);
$this->db->delete('mytable');

// Produces:
// DELETE FROM mytable
// WHERE id = $id
```

An array of table names can be passed into delete() if you would like to delete data from more than 1 table.

```
$tables = array('table1', 'table2', 'table3');
$this->db->where('id', '5');
$this->db->delete($tables);
```

If you want to delete all data from a table, you can use the truncate() function, or empty_table().

**$this->db->empty_table()**

Generates a delete SQL string and runs the query.:

```
$this->db->empty_table('mytable'); // Produces: DELETE FROM mytable
```

**$this->db->truncate()**

Generates a truncate SQL string and runs the query.

```
$this->db->from('mytable');
$this->db->truncate();

// or

$this->db->truncate('mytable');

// Produce:
// TRUNCATE mytable
```

**Note**

If the TRUNCATE command isn't available, truncate() will execute as "DELETE FROM table".

**$this->db->get_compiled_delete()**

This works exactly the same way as `$this->db->get_compiled_insert()` except that it produces a DELETE SQL string instead of an INSERT SQL string.

For more information view documentation for $this->db->get_compiled_insert().

## Method Chaining

Method chaining allows you to simplify your syntax by connecting multiple functions. Consider this example:

```
$query = $this->db->select('title')
                  ->where('id', $id)
                  ->limit(10, 20)
                  ->get('mytable');
```

## Query Builder Caching

While not “true” caching, Query Builder enables you to save (or “cache”) certain parts of your queries for reuse at a later point in your script’s execution. Normally, when an Query Builder call is completed, all stored information is reset for the next call. With caching, you can prevent this reset, and reuse information easily.

Cached calls are cumulative. If you make 2 cached select() calls, and then 2 uncached select() calls, this will result in 4 select() calls. There are three Caching functions available:

**$this->db->start_cache()**

This function must be called to begin caching. All Query Builder queries of the correct type (see below for supported queries) are stored for later use.

**$this->db->stop_cache()**

This function can be called to stop caching.

**$this->db->flush_cache()**

This function deletes all items from the Query Builder cache.

## An example of caching

Here's a usage example:

```
$this->db->start_cache();
$this->db->select('field1');
$this->db->stop_cache();
$this->db->get('tablename');
//Generates: SELECT `field1` FROM (`tablename`)

$this->db->select('field2');
$this->db->get('tablename');
//Generates:  SELECT `field1`, `field2` FROM (`tablename`)

$this->db->flush_cache();
$this->db->select('field2');
$this->db->get('tablename');
//Generates:  SELECT `field2` FROM (`tablename`)
```

**Note**

The following statements can be cached: select, from, join, where, like, group_by, having, order_by

# Resetting Query Builder

**$this->db->reset_query()**

Resetting Query Builder allows you to start fresh with your query without executing it first using a method like $this->db->get() or $this->db->insert(). Just like the methods that

execute a query, this will *not* reset items you've cached using Query Builder Caching.

This is useful in situations where you are using Query Builder to generate SQL (ex. `$this->db->get_compiled_select()`) but then choose to, for instance, run the query:

```
// Note that the second parameter of the get_compiled_select method is FALSE
$sql = $this->db->select(array('field1','field2'))
                                ->where('field3',5)
                                ->get_compiled_select('mytable', FALSE);

// ...
// Do something crazy with the SQL code... like add it to a cron script for
// later execution or something...
// ...

$data = $this->db->get()->result_array();

// Would execute and return an array of results of the following query:
// SELECT field1, field1 from mytable where field3 = 5;
```

**Note**

Double calls to `get_compiled_select()` while you're using the Query Builder Caching functionality and NOT resetting your queries will results in the cache being merged twice. That in turn will i.e. if you're caching a `select()` - select the same field twice.

## Class Reference

*class* `CI_DB_query_builder`

`reset_query` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Resets the current Query Builder state. Useful when you want to build a query that can be cancelled under certain conditions.

`start_cache` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Starts the Query Builder cache.

`stop_cache` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Stops the Query Builder cache.

`flush_cache` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Empties the Query Builder cache.

`set_dbprefix` ( [ *$prefix = ''* ] )

**Parameters:**
- **$prefix** (*string*) – The new prefix to use

**Returns:** The DB prefix in use

**Return type:** string

Sets the database prefix, without having to reconnect.

`dbprefix` ( [ *$table = ''* ] )

**Parameters:**
- **$table** (*string*) – The table name to prefix

**Returns:** The prefixed table name

**Return type:** string

Prepends a database prefix, if one exists in configuration.

`count_all_results` ( [ *$table = ''* [ , *$reset = TRUE* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$reset** (*bool*) – Whether to reset values for SELECTs

**Returns:** Number of rows in the query result

**Return type:** int

Generates a platform-specific query string that counts all records returned by an Query Builder query.

`get` ( [ *$table = ''* [ , *$limit = NULL* [ , *$offset = NULL* ] ] ] )

**Parameters:**
- **$table** (*string*) – The table to query
- **$limit** (*int*) – The LIMIT clause
- **$offset** (*int*) – The OFFSET clause

**Returns:** CI_DB_result instance (method chaining)

**Return type:** CI_DB_result

Compiles and runs SELECT statement based on the already called Query Builder methods.

`get_where` ( [ *$table = ''* [ , *$where = NULL* [ , *$limit = NULL* [ , *$offset = NULL* ] ] ] ] )

**Parameters:**
- **$table** (*mixed*) – The table(s) to fetch data from; string or array
- **$where** (*string*) – The WHERE clause
- **$limit** (*int*) – The LIMIT clause
- **$offset** (*int*) – The OFFSET clause

**Returns:** CI_DB_result instance (method chaining)

**Return type:** CI_DB_result

Same as `get()`, but also allows the WHERE to be added directly.

`select` ( [ *$select = '*'* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$select** (*string*) – The SELECT portion of a query
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a SELECT clause to a query.

`select_avg` ( [ *$select = ''* [ , *$alias = ''* ] ] )

**Parameters:**
- **$select** (*string*) – Field to compute the average of
- **$alias** (*string*) – Alias for the resulting value name

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a SELECT AVG(field) clause to a query.

`select_max` ( [ *$select = ''* [ , *$alias = ''* ] ] )

**Parameters:**
- **$select** (*string*) – Field to compute the maximum of
- **$alias** (*string*) – Alias for the resulting value name

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a SELECT MAX(field) clause to a query.

`select_min` ( [ *$select* = '' [ , *$alias* = '' ] ] )

**Parameters:**
- **$select** (*string*) – Field to compute the minimum of
- **$alias** (*string*) – Alias for the resulting value name

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a SELECT MIN(field) clause to a query.

`select_sum` ( [ *$select* = '' [ , *$alias* = '' ] ] )

**Parameters:**
- **$select** (*string*) – Field to compute the sum of
- **$alias** (*string*) – Alias for the resulting value name

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a SELECT SUM(field) clause to a query.

`distinct` ( [ *$val* = *TRUE* ] )

**Parameters:**
- **$val** (*bool*) – Desired value of the "distinct" flag

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Sets a flag which tells the query builder to add a DISTINCT clause to the SELECT portion of the query.

`from` (*$from*)

**Parameters:**
- **$from** (*mixed*) – Table name(s); string or array

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Specifies the FROM clause of a query.

`join` (*$table, $cond* **[** *, $type = ''* **[** *, $escape = NULL* **]** **]** )

**Parameters:**
- **$table** (*string*) – Table name to join
- **$cond** (*string*) – The JOIN ON condition
- **$type** (*string*) – The JOIN type
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a JOIN clause to a query.

`where` (*$key* **[** , *$value = NULL* **[** , *$escape = NULL* **]** **]** )

**Parameters:**
- **$key** (*mixed*) – Name of field to compare, or associative array
- **$value** (*mixed*) – If a single key, compared to this value
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates the WHERE portion of the query. Separates multiple calls with 'AND'.

`or_where` (*$key* **[** , *$value = NULL* **[** , *$escape = NULL* **]** **]** )

**Parameters:**
- **$key** (*mixed*) – Name of field to compare, or associative array
- **$value** (*mixed*) – If a single key, compared to this value
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates the WHERE portion of the query. Separates multiple calls with 'OR'.

`or_where_in` ( **[** *$key = NULL* **[** , *$values = NULL* **[** , *$escape = NULL* **]** **]** **]** )

**Parameters:**
- **$key** (*string*) – The field to search

- **$values** (*array*) – The values searched on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates a WHERE field IN('item', 'item') SQL query, joined with 'OR' if appropriate.

`or_where_not_in` ( [ *$key = NULL* [ , *$values = NULL* [ , *$escape = NULL* ] ] ] )

**Parameters:**
- **$key** (*string*) – The field to search
- **$values** (*array*) – The values searched on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates a WHERE field NOT IN('item', 'item') SQL query, joined with 'OR' if appropriate.

`where_in` ( [ *$key = NULL* [ , *$values = NULL* [ , *$escape = NULL* ] ] ] )

**Parameters:**
- **$key** (*string*) – Name of field to examine
- **$values** (*array*) – Array of target values
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates a WHERE field IN('item', 'item') SQL query, joined with 'AND' if appropriate.

`where_not_in` ( [ *$key = NULL* [ , *$values = NULL* [ , *$escape = NULL* ] ] ] )

**Parameters:**
- **$key** (*string*) – Name of field to examine
- **$values** (*array*) – Array of target values
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** DB_query_builder instance

**Return type:** object

Generates a WHERE field NOT IN('item', 'item') SQL query, joined with 'AND' if appropriate.

`group_start` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Starts a group expression, using ANDs for the conditions inside it.

`or_group_start` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Starts a group expression, using ORs for the conditions inside it.

`not_group_start` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Starts a group expression, using AND NOTs for the conditions inside it.

`or_not_group_start` ()

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Starts a group expression, using OR NOTs for the conditions inside it.

`group_end` ()

**Returns:** DB_query_builder instance

**Return type:** object

Ends a group expression.

`like` (*$field* **[** , *$match = ''* **[** , *$side = 'both'* **[** , *$escape = NULL* **]** **]** **]** )

**Parameters:**

- **$field** (*string*) – Field name
- **$match** (*string*) – Text portion to match
- **$side** (*string*) – Which side of the expression to put the '%' wildcard on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a LIKE clause to a query, separating multiple calls with AND.

`or_like` (*$field* **[** , *$match = ''* **[** , *$side = 'both'* **[** , *$escape = NULL* **]** **]** **]** )

**Parameters:**
- **$field** (*string*) – Field name
- **$match** (*string*) – Text portion to match
- **$side** (*string*) – Which side of the expression to put the '%' wildcard on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a LIKE clause to a query, separating multiple class with OR.

`not_like` (*$field* **[** , *$match = ''* **[** , *$side = 'both'* **[** , *$escape = NULL* **]** **]** **]** )

**Parameters:**
- **$field** (*string*) – Field name
- **$match** (*string*) – Text portion to match
- **$side** (*string*) – Which side of the expression to put the '%' wildcard on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

CI_DB_query_builder

**Return type:**

Adds a NOT LIKE clause to a query, separating multiple calls with AND.

`or_not_like` (*$field* [ , *$match = ''* [ , *$side = 'both'* [ , *$escape = NULL* ] ] ] )

**Parameters:**
- **$field** (*string*) – Field name
- **$match** (*string*) – Text portion to match
- **$side** (*string*) – Which side of the expression to put the '%' wildcard on
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a NOT LIKE clause to a query, separating multiple calls with OR.

`having` (*$key* [ , *$value = NULL* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$key** (*mixed*) – Identifier (string) or associative array of field/value pairs
- **$value** (*string*) – Value sought if $key is an identifier
- **$escape** (*string*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a HAVING clause to a query, separating multiple calls with AND.

`or_having` (*$key* [ , *$value = NULL* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$key** (*mixed*) – Identifier (string) or associative array of field/value pairs
- **$value** (*string*) – Value sought if $key is an identifier
- **$escape** (*string*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a HAVING clause to a query, separating multiple calls with OR.

`group_by` (*$by* [ , *$escape = NULL* ] )

**Parameters:**
- **$by** (*mixed*) – Field(s) to group by; string or array

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds a GROUP BY clause to a query.

`order_by` (*$orderby* [ , *$direction = ''* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$orderby** (*string*) – Field to order by
- **$direction** (*string*) – The order requested - ASC, DESC or random
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds an ORDER BY clause to a query.

`limit` (*$value* [ , *$offset = 0* ] )

**Parameters:**
- **$value** (*int*) – Number of rows to limit the results to
- **$offset** (*int*) – Number of rows to skip

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds LIMIT and OFFSET clauses to a query.

`offset` (*$offset*)

**Parameters:**
- **$offset** (*int*) – Number of rows to skip

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds an OFFSET clause to a query.

`set` (*$key* [ , *$value = ''* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$key** (*mixed*) – Field name, or an array of field/value pairs
- **$value** (*string*) – Field value, if $key is a single field

- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds field/value pairs to be passed later to `insert()`, `update()` or `replace()`.

`insert` ( [ *$table = ''* [ , *$set = NULL* [ , *$escape = NULL* ] ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$set** (*array*) – An associative array of field/value pairs
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** TRUE on success, FALSE on failure

**Return type:** bool

Compiles and executes an INSERT statement.

`insert_batch` (*$table* [ , *$set = NULL* [ , *$escape = NULL* [ , *$batch_size = 100* ] ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$set** (*array*) – Data to insert
- **$escape** (*bool*) – Whether to escape values and identifiers
- **$batch_size** (*int*) – Count of rows to insert at once

**Returns:** Number of rows inserted or FALSE on failure

**Return type:** mixed

Compiles and executes batch `INSERT` statements.

> **Note**
>
> When more than `$batch_size` rows are provided, multiple `INSERT` queries will be executed, each trying to insert up to `$batch_size` rows.

`set_insert_batch` (*$key* [ , *$value* = '' [ , *$escape* = *NULL* ] ] )

**Parameters:**
- **$key** (*mixed*) – Field name or an array of field/value pairs
- **$value** (*string*) – Field value, if $key is a single field
- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds field/value pairs to be inserted in a table later via `insert_batch()`.

`update` ( [ *$table* = '' [ , *$set* = *NULL* [ , *$where* = *NULL* [ , *$limit* = *NULL* ] ] ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$set** (*array*) – An associative array of field/value pairs
- **$where** (*string*) – The WHERE clause
- **$limit** (*int*) – The LIMIT clause

**Returns:** TRUE on success, FALSE on failure

**Return type:** bool

Compiles and executes an UPDATE statement.

`update_batch` (*$table* [ , *$set = NULL* [ , *$value = NULL* [ , *$batch_size = 100* ] ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$set** (*array*) – Field name, or an associative array of field/value pairs
- **$value** (*string*) – Field value, if $set is a single field
- **$batch_size** (*int*) – Count of conditions to group in a single query

**Returns:** Number of rows updated or FALSE on failure

**Return type:** mixed

Compiles and executes batch `UPDATE` statements.

**Note**

When more than `$batch_size` field/value pairs are provided, multiple queries will be executed, each handling up to `$batch_size` field/value pairs.

`set_update_batch` (*$key* [ , *$value = ''* [ , *$escape = NULL* ] ] )

**Parameters:**
- **$key** (*mixed*) – Field name or an array of field/value pairs
- **$value** (*string*) – Field value, if $key is a single field

- **$escape** (*bool*) – Whether to escape values and identifiers

**Returns:** CI_DB_query_builder instance (method chaining)

**Return type:** CI_DB_query_builder

Adds field/value pairs to be updated in a table later via `update_batch()`.

`replace` ( [ *$table = ''* [ , *$set = NULL* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$set** (*array*) – An associative array of field/value pairs

**Returns:** TRUE on success, FALSE on failure

**Return type:** bool

Compiles and executes a REPLACE statement.

`delete` ( [ *$table = ''* [ , *$where = ''* [ , *$limit = NULL* [ , *$reset_data = TRUE* ] ] ] ] )

**Parameters:**
- **$table** (*mixed*) – The table(s) to delete from; string or array
- **$where** (*string*) – The WHERE clause
- **$limit** (*int*) – The LIMIT clause
- **$reset_data** (*bool*) – TRUE to reset the query "write" clause

**Returns:** CI_DB_query_builder instance (method chaining) or FALSE on failure

**Return type:** mixed

Compiles and executes a DELETE query.

`truncate` ( [ *$table = ''* ] )

**Parameters:**
- **$table** (*string*) – Table name

**Returns:** TRUE on success, FALSE on failure

**Return type:** bool

Executes a TRUNCATE statement on a table.

> **Note**
>
> If the database platform in use doesn't support TRUNCATE, a DELETE statement will be used instead.

`empty_table` ( [ *$table = ''* ] )

**Parameters:**
- **$table** (*string*) – Table name

**Returns:** TRUE on success, FALSE on failure

**Return type:** bool

Deletes all records from a table via a DELETE statement.

`get_compiled_select` ( [ *$table = ''* [ , *$reset = TRUE* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$reset** (*bool*) – Whether to reset the current QB values or not

**Returns:** The compiled SQL statement as a string

**Return type:** string

Compiles a SELECT statement and returns it as a string.

`get_compiled_insert` ( [ *$table = ''* [ , *$reset = TRUE* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$reset** (*bool*) – Whether to reset the current QB values or not

**Returns:** The compiled SQL statement as a string

**Return type:** string

Compiles an INSERT statement and returns it as a string.

`get_compiled_update` ( [ *$table = ''* [ , *$reset = TRUE* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$reset** (*bool*) – Whether to reset the current QB values or not

**Returns:** The compiled SQL statement as a string

**Return type:** string

Compiles an UPDATE statement and returns it as a string.

`get_compiled_delete` ( [ *$table = ''* [ , *$reset = TRUE* ] ] )

**Parameters:**
- **$table** (*string*) – Table name
- **$reset** (*bool*) – Whether to reset the current QB values or not

**Returns:** The compiled SQL statement as a string

**Return type:** string

Compiles a DELETE statement and returns it as a string.

Previous Next