V2. Incomplete and Utter Guide for WP_Query

Rich and flexible database request handling in WordPress allows shortened development cycles, provides efficiency and stability, but it is up to developers to use it in an effective way to achieve optimal performance. Attention to the amount and complexity of database requests can lead to reasonable architectural decisions.

WordPress manages the main query — post, page, archive, feed, and so on — and sets all global variables for themes and plugins to use. In many cases, you might need to get additional data or modify the main query and for this we need to know how it works.

Functions and classes

Many built-in functions exist to get results in a robust way, providing a high level of abstraction, and usage of a function is preferable to object creation — functions already have some defaults that allow fewer arguments, checks, and validations to perform. And all unnecessary data is cleaned after the function returns the result, freeing memory that can be crucial on scale.

get_posts()

This function can satisfy most requests and can be called without any arguments to return 5 latest posts. get_posts() ignores sticky posts and is not asking the database to calculate the amount of rows that are matching the request, all other arguments are equal to the WordPress Query class (WP_Query).

get_posts() guards against incorrect usage of WP_Query class and is preferable to use.

WP_Query()

In cases where get_posts() cannot be used, if, for example, we need to know how many rows are matching the request to build pagination, WP_Query class can be used to create a new object.

<table>
<tr><td>Incorrect ❌
This doubles the request processing.</td><td>Correct ✅</td></tr>
<tr><td rowspan="2">$get_posts = new WP_Query($my_args);
$my_posts = $get_posts->get_posts();</td><td>$get_posts = new WP_Query();
$my_posts = $get_posts->query($my_args);</td></tr>
<tr><td>$get_posts = new WP_Query($my_args);</td></tr>
</table>

	$my_posts = $get_posts->posts;

Contrary to using it inside the get_posts() function, your $get_posts object will be kept in memory until the end of a scope. If, for example, we created this object in a plugin or theme outside any function, it will be accessible in the global scope and live until the end of the request processing as well as the cache of the data that was collected.

Main query()

Necessary making a request instead of the main query is a sign that the solution is far from optimal.
Example: Developers build a plugin with a feed for custom post type Note.
They registered a feed with add_feed() function and provided a callback function that makes the right markup, but they don't need posts and this is what they get. So, they make a new WP_Query() with correct arguments, but something is wrong with pagination. They want 50 notes on a page and they have 140 notes created, but only 28 posts exist and the second and further pages of the feed return code 404. They came up with a solution to force code 200.

The plugin is working and clients are grateful, even if they are noticing that the change in “Settings > Reading > Syndication feeds show the most recent X items” has no effect on the amount of the items in the notes feed.

In this example, the correct and easier way is to use the ‘pre_get_posts’ filter to set up the post type and leave the rest for WordPress.

Incorrect ❌ Main query made by WordPress is abandoned and resources are wasted	Correct ✅

add_feed('myfeed', 'myfeed_feed');
 	
function myfeed_feed() {
  // …
  if (isset($_GET['paged'])) {
  $paged = $_GET['paged'];
  } else {
  $paged = 1;
  }
  // …
  $args = [
  'post_type' => 'note',
  'posts_per_page' => 50,
  'paged' => $paged,
  ];
  }
	function myfeed_filter_pre_get_posts($query) {
	  if (! is_feed() || ! $query->is_main_query()) {
	  return;
	  }
	
	  if ('myfeed' !== ! $query->get('feed')) {
	  return;
	  }
	
	  $query->set('post_type', 'note');
	}

```
    $query = new WP_Query($args);
    // …
    while ($query->have_posts()) :
        $query->the_post();
        // …
    endwhile;
}


add_action('tempate_redirect',
'myfeed_header_fix');

function myfeed_header_fix() {
    if ( is_feed( 'myfeed' ) ) {
        header( 'HTTP/1.1 200 OK', true );
    }
}
```

```
add_action( 'pre_get_posts',
'myfeed_filter_pre_get_posts' );

add_feed( 'myfeed', 'myfeed_feed' );

function myfeed_feed() {
    // …
    while (have_posts()) :
        the_post();
        // …
    endwhile;
}
```

In most cases, if development with WordPress is getting too complicated, it is a sign to question the solution architecture. There is a gut feeling from times to times we need to trust.

get_post()

In case when we have IDs of posts to get, there are two possible options.

Incorrect ❌

```
foreach ( $lessons_ids as $lesson_id ) {
    $lesson = get_post( $lesson_id );
    // …
}
```

Correct ✅

```
$args  = [
    'posts_per_page' => count( $lessons_ids
),
    'post__in'            => $lessons_ids,
    'orderby'             => 'post__in',
    'post_type'           => 'lesson',
 ];

$lessons = get_posts( $args );

foreach ( $lessons as $lesson ) {
    // …
}
```

Usage of get_post() looks simpler than get_posts() and it avoids complicated query build, but at the cost of a request to DB per each ID, so, the general rule is if you have a loop, collect the data with one request to DB beforehand.

Random posts

Request to get random posts instead of certain ones has more cost in processing by database and WordPress cannot cache it because the result is unpredictable.
We may want to have such a collection in the section: “You may also like” with products, news, articles etc.

Nonoptimal ❌	Optimal ✅
$args = ['posts_per_page' => 4, 'post__not_in' => $post_id, 'post_type' => 'product', 'orderby' => 'rand',]; $products = get_posts($args);	$args = ['posts_per_page' => 4, 'post__not_in' => $post_id, 'post_type' => 'product', 'orderby' => 'date', 'order' => 'DESC',]; $products = get_posts($args);

From a business point of view this section can be more useful presenting hand picked by editor, most popular items in the same category or items calculated based on statistics, but it requires more work and randomising looks like a chip alternative but can turn into a performance issue on a large scale.

Query caching

If the query is not using random order, it will be cached in the WP_Object_Cache() object until the end of processing the request or in persistent object cache such as Redis/Valkey.

Main query is cached by default and the need is obvious; the theme and plugins are requiring this data multiple times during the request process, but with request for additional data we can make a continuous decision to cache it or not to cache.

Caching strategy can differ in regards to persistent object cache (POC) usage. If data collected by the request is used once and there is no need to preserve further, optimal is not to cache it, but when persistent object cache is in place we still may want to cache it.

Optimal without POC ✅ Optimal with POC when result will be stored in cache by the handler itself ✅	Optimal with POC ✅
$args = ['posts_per_page' => 4, 'author' => $author_id,	$args = ['posts_per_page' => 4, 'author' => $author_id, 'post_type' => 'quote',

```
    'post_type'           => 'quote',
    'cache_results'       => 'false'
];

$quotes = get_posts( $args );
```

```
];

$quotes = get_posts( $args );
```

If ‘cache_results’ parameter will be true or absent, the request will be cached until the end of the process and WP_Query() will not get to the database second, third, forth and so on times, but still, it will calculate the whole request parameters, before searching in the cache storing object. By thinking ahead what data is needed and where, duplicate requests handling can be avoided. If different parts of the code need the same data, you can initiate an object that will store purely what you need, and feed this data from the object. After this `cache_results` can be set to false to avoid keeping data you don’t need and go through the process of building requests.

There are also options not to 'update_post_meta_cache' and 'update_post_term_cache', and the decision can differ from one case to another, but generally switching off caching is less preferable because memory usage can cost in performance less then trips to the database.

Note: The Query Monitor plugin can help to make a reasonable decision about caching.

Indexes

Getting a post (page etc) by ID is quick, getting it by title is a bit slower, search in the content is slower still. The difference cannot be significant for a small amount of rows, but this is all getting up to each other and if project growth can become significant.

The *_posts table has several indexes: ID, post_author, post_date, post_status, post_name (slug), post_parent and post_type. They can be used to limit the amount of rows a search request needs to look through. For example, on a big news portal we can limit search by dates, on an educational platform by author and post_type, etc.

Horrible request ❌

```
$args  = [
    'posts_per_page'  => -1,
    'post_type'           => 'any',
];

$result = get_posts( $args );
```

Correct ✅

```
$args  = [
    'posts_per_page'  => 4,
    'post_author'        => $author_id,
    'post_type'           => 'article',
    'orderby'             => 'date',
    'order'               => 'DESC',
];

$articles = get_posts( $args );
```

Sorting by index fields is also more efficient than trying to sort fields without index. In case when we need to sort by title, post_name can be considered as a more effective substitute as this is an index field.

IDs

WP_Query has an option to return only IDs of matching rows, for example, this can be used to show the amount of author's articles outside of the author's archive where this information will be available as part of the main request, or when we need to know in general if any rows are matching the condition, for example to check if coupon code has already been used for a purchase by a user.

In cases when we need ids to loop through them and use get_post(), get_the_title(), get_permalink() on each of them, retrieval of the whole data is preferable.

Incorrect ❌	Correct ✅

$args = [
'posts_per_page' => 10,
'fields' => 'ids',
'post_author' => $author_id,
'post_type' => 'lesson',
'orderby' => 'date',
'order' => 'DESC',
];
 	
$get_posts = new WP_Query();
$lessons_ids = $get_posts->query($args);
 	
foreach ($lessons_ids as $lesson_id) {
$lesson = get_post($lesson_id);
// …
}
	$args = [
	'posts_per_page' => 1,
	'fields' => 'ids',
	'post_author' => $author_id,
	'post_type' => 'lesson',
	'orderby' => 'date',
	'order' => 'DESC',
	];
	
	$get_posts = new WP_Query($args);
	$lessons_amount =
	$get_posts->found_posts;
	
	// … Do something with $lessons_amount

Tax_ and meta_ query

Tax_ and meta_ queries can be slow, but in most cases there is no way around. We can narrow the amount of data to look through—the 'meta_key' column of *_postmeta table is an index, even if it can be quite long, so search by meta should at least have 'meta_key' and avoid partial matches or searches without 'meta_key'.

Sometimes it is prudent to save a preselected list of data we need into transient so as not to perform the query each time. What is better will depend on the circumstances.

For example, if we ponder creating a page “Small green handbags” and add on it matching products, we may consider adding all these products to a new category created especially for them and use this category instead of the page with complicated query or a transient.

$wpdb, why not?

$wpdb allows to perform any request to database and this is the only way to work with custom tables, for example, plugin that stores performance statistic in a custom table in the database has the reason to use $wpdb, but it should provide API to get or modify this data to user or any other consumers. This way the plugin can safely make migrations, change data structure and serve the same stable data through API.

The same logic applies to WordPress inbuilt tables — the whole interaction should go through system API and WP_Query class is such service for all request to *_posts table. And usage of existing system APIs makes your solution more robust and allows you to benefit from future performance improvements.

Incorrect ❌	Correct ✅
global $wpdb; $sql_query = "SELECT ID FROM {$wpdb->posts} WHERE ((post_date > 0 && post_date <= %s)) AND post_type = 'article' AND post_status = 'future'"; $tomorrow_time = current_time('timestamp') + 24 * 60 * 60; $sql = $wpdb->prepare($sql_query, date('Y-m-d H:i:s', $tomorrow_time)); $scheduled_post_ids = $wpdb->get_col($sql);	$args = ['posts_per_page' => 10, 'post_type' => 'article', 'post_status' => 'future', 'fields' => 'ids', 'date_query' => array(array('column' => 'post_date', 'before' => '+1 day',), array('column' => 'post_date', 'after' => 'now',), 'relation' => 'AND',),]; $scheduled_post_ids = get_posts($args);

V1. Optimising WP_Query queries for better performance

WordPress has several classes to work with different objects — posts, users, terms, sites, neworks, comments etc. Files with all these classes are named class-wp-*-query.php and can be found in /wp-includes/ folder. The WP_Query class is dedicated to work with posts, pages and all other post types that are living in the *_posts table in the database.

The main query — post, page, archive, feed and so on is handled by WordPress and it sets all global variables for theme and plugins to use, but in many cases we need to get additional data or modify the main query and for this we need to know how WP_Query works.

1. Use get_posts() is possible instead of WP_Query()

For different purposes we have different functions to simplify things and remove unnecessary complications, for example get_posts() has default values and can be called without any arguments to return 5 latest posts. This function ignores sticky posts and is not asking the database to calculate the amount of rows that are matching the request. All other arguments can be set and processed by WP_Query() inside. The object itself after returning the data will be destroyed by the PHP because it was initialised inside the function's scope that ends, that means freeing a bit of memory. This function is also garding against incorrect usage of WP_Query() having the right approach inside:

```
…
$get_posts = new WP_Query();
return $get_posts->query( $parsed_args );
```

There is also other possible way to use WP_Query():

```
$get_posts = new WP_Query( $parsed_args );
return $get_posts->posts; // $posts property is public
```

And incorrect way that doubles the request processing:

```
$get_posts = new WP_Query( $parsed_args );
return $get_posts->get_posts();
```

And it is widely used by developers because the get_posts() method looks like a getter and actually returns the result. So, to optimise our requests, we have to know the mechanic behind.

2. Do not make requests if you don't need it

Example: You are building a plugin that creates custom feed for third party applications. You registered a feed with add_feed() function and provided a callback function that is making the right markup but you don't need posts and want to add especially some custom post type instead — the main purpose of your new plugin. So, you are making new

WP_Query() with arguments you need and are having a lot of trouble ending with forcing any page returning code 200 without regards to what WordPress is considering as the correct status code. Your plugin is working and end users are satisfied for now. But the moment when you start to feel that the system is not behaving correctly, you should stop to think about what you are doing. You abandoned the main request that was already processed by the CMS and created your own that is not working properly because the main request has different pagination and different post_type. So, the right way here is to use the 'pre_get_posts' filter and set up whatever you want the main query to return.

```
function myfeed_filter_pre_get_posts( $query ) {
  if ( ! is_feed() || ! $query->is_main_query() ) {
    return;
  }

  if ( 'myfeed' !== ! $query->get( 'feed' ) ) {
    return;
  }

  $query->set( 'post_type', 'my_post_type' );
  $query->set( 'posts_per_rss', 50 );
}

add_action( 'pre_get_posts', 'myfeed_filter_pre_get_posts' );
```

3. Don't user get_post($my_post_id) in the cycle

If you have a list of posts that should appear on the page, for example you have a lesson plan with a certain order of lessons, it will be more effective to get them at once keeping the order you need.

```
$get_posts = new WP_Query();
$lessons    = $get_posts->query( [
          'posts_per_page' => -1,
          'post__in'            => $lessons_ids,
          'orderby'             => 'post__in'
          'post_type'           => 'any', // Whatever type you need here
  ] );
```

4. Do not make random requests if you can avoid them

Example: You want to add to a post related posts and decide to add them randomly.

```
$get_posts = new WP_Query();
$related_posts = $get_posts->query( [ 'orderby' => 'rand', 'posts_per_page' => 4  ] );
```

This request cannot be cached because of unpredictable results and it does not matter if you don't need to reuse this request further, but depending on a database size, it can consume more resources than necessary on the database side. If you don't want to add last posts, consider adding them with custom code or a plugin into the custom field (post_meta). It will

require a bit more work, but you will be able to attach posts that are related to the topic in post and can interest your audience more than a random set.

If manual selection is not a good fit for your purpose, consider selecting the last 10 posts and then pick 4 from them. Note that you cannot order by some index column and limit the initial selection and use rand at the same time.

5. Choose what you want to cache

If the query is not using random order, it will be cached in the WP_Object_Cache() object until the end of processing the request or in persistent object cache if you are using one. Working with persistent object cache is out of scope of this article, but be careful with any request that returns a huge amount of data and if you have just default object caching, making each request, decide if you need this data further during the process or not. Main query will be cached and it is what we need because the theme and plugins will be requiring this data multiple times. But if you have a custom post type for motivational quotes and chousen quote should appear on a page once, there is no point in caching the request that retrieves it.

```
$get_posts = new WP_Query();
$quotes = $get_posts->query( [
        ‘post_type => ‘quote’,
        'posts_per_page' => 1,
        'cache_results' => 'false'
] );
```

If ‘cache_results’ parameter will be true or absent, the request will be cached until the end of the process and WP_Query() will not get to the database second, third, forth and so on times, but still, it will calculate the whole request parameters, before searching in the cache storing object. By thinking ahead in the beginning of the page build about what data is needed further, duplicate requests can be avoided. If different parts of the code need the same data, you can initiate an object that will store purely what you need, and feed this data from the object. After this `cache_results` can be set to false to avoid keeping data you don’t need and go through the process of building requests.

There is also an option not to 'update_post_meta_cache' and 'update_post_term_cache', make a conscious choice about what you need to reuse.

6. Use indexes if you can

Getting a post (page etc) by ID is quick, getting it by title is a bit slower, search in the content is slower still. The difference cannot be significant if you have a small amount of rows, but this is all getting up to each other and it is better to think about performance without regard to the project size. So, the *_posts table has several indexes: ID, post_author, post_date, post_status, post_name (slug), post_parent and post_type. Try to build your requests with them. If you have search on a site with many custom post types, consider to offer by default search in most popular to limit the amount of rows to look through.

And it is better to add post_type and posts_per_page (posts_per_rss for rss) to all your WP_Query requests to avoid confusion. If you have a wrong result it is obvious what to add, but if you have no result at all it can take several attempts to remember that there is a default

post_type equal to ‘post’ as well as default posts_per_page/posts_per_rss values that can be changed in the Admin (Settings > Reading).

7. Do not sort data if you don’t need or try to sort by index column if you can

Most likely you need to sort by ID or post_date and these are indexes. And if you want to sort by post_title, consider post_name instead—in some cases it will not be a good idea, but in others it will work just fine.

8. Try to avoid retrieving data you don’t need

If you only need posts IDs and you don’t need these post further, it is better just to ask for this ids like this:

```
$get_posts = new WP_Query();
return $get_posts->query( [ 'fields' => 'ids'  ] );
```

And this is another option 'id=>parent' you can use in certain circumstances.

9. Do not count all matching rows in the database if you don’t need this amount.

```
$get_posts = new WP_Query();
return $get_posts->query( [ 'no_found_rows' => 'true'  ] );
```

10. Avoid setting up global variables if you need the result here and now.

```
$get_posts = new WP_Query( [ 'post_type' => 'page'  ] );
while ( $get_posts->have_posts() ) {
  $get_posts->the_post(); // Setting up global variables
  the_title();
  // Doing something
}
wp_reset_data(); // Restoring global variables to original stage (main query)
```

It is convenient to use functions such as get_the_ID() or the_title() but it is just quite similar to get the same information from the WP_Post object that already contains the information you need.

```
$get_posts = new WP_Query();
$my_pages =  $get_posts->query( [ 'post_type' => 'page' ] );
foreach ( $my_pages as $my_page ) {
        $my_title = $my_page->post_title;
}
```

Because $my_page is an instance of WP_Post it will have all the properties that you expect and you can rely on such properties as $post_title, $post_status etc to exist without checking them first.

11. Supply template functions with instance of WP_Post object

Note that a call of a template function is getting the full post from database and store it in the cache until the end of the process (if you don't have persistent cache) and by calling just get_the_title($my_post_id) and get_permalink($my_post_id) we are not optimising anything. What is more, if we are providing just $my_post_id instead of an instance of WP_Post object, the object will be recreated and then disposed of each time we are calling the template function. To remove a bit of a hustle, the post object should be provided.

```
function get_the_title( $post = 0 ) {
	$post = get_post( $post );
…
}

function get_post( $post = null, $output = OBJECT, $filter = 'raw' ) {
…
	if ( $post instanceof WP_Post ) {
		$_post = $post;
	} elseif ( is_object( $post ) ) {
		if ( empty( $post->filter ) ) {
			$_post = sanitize_post( $post, 'raw' );
			$_post = new WP_Post( $_post );
		} elseif ( 'raw' === $post->filter ) {
			$_post = new WP_Post( $post );
		} else {
			$_post = WP_Post::get_instance( $post->ID );
		}
	} else {
		$_post = WP_Post::get_instance( $post );
	}

…

	return $_post;
}
```

12. Avoid using $wpdb if you can get what you need from WP_Query().

$wpdb allows to perform any request to database you want and this is the only way to work with your own custom tables, but it you need data from custom table that belong to plugin, this plugins API is the correct source to get it, and if you need data from *_posts table, WP_Query() is the provider you need. Usage of existing system APIs makes your solution more robust and allows you to benefit from future performance improvements.

13. Meta and tax queries

Your IDE will highlight that you have meta or tax query and they are possibly slow once but most likely you cannot be avoided but if you have other data you can narrow the amount of data to look through, use it. The 'meta_key' column is also an index, even if it can be quite

long, narrow the search with ‘meta_key’ and don’t go on a wild goose chase without it. Try to specify requests as much as possible avoiding partial matches etc.

If you have a special selection for some purpose, consider keeping the list of posts you need in option to avoid querying with meta. What is better will depend on the circumstances.

Before using the same taxonomies for different post types, consider use cases and what will be better from an architectural point of view. Having use cases and thoughtful architecture can save you a lot of trouble later, but maintaining a live project architectural change is most likely not possible or very difficult.

[EXAMPLES]

14. Filters suppression

Do not add ‘suppress_filters’ if you are not sure. If you don’t have filters to suppress, this will not bring any visible benefit but if you have some filters you are not aware about, the outcome can be unexpected. Look at filters that can be suppressed if you decide if there is something you need to switch off for certain cases. The better approach will be to manage the application of the filter itself if possible.

15. Sticky posts

If you have no sticky posts, you don’t need to specify ‘ignore_sticky_posts’ to avoid additional requests for them, because their IDs will be stored in an auto loaded option and retrieved all at once if this array is not empty. This is a good example of how to behave when you need a special set of posts - to avoid meta query that will look for certain values but keeping them all in one option instead.

16. Find the right moment to stop optimization

Desire to switch off everything that you don’t need makes sense, for example suppress all filters, but these actions can have side effects and even if something is not in use right now, it doesn’t mean that it will not be needed shortly. Switched off functionality can cause confusion and should be documented in the project README and in the code when this is taking place explaining the decision.

17. Pay attention to your logs

Having detailed information about what is going on on your site is very convenient, and sending data into Sentry is great for debugging, but if you have many issues, even notices, it can result in 5xx errors due to additional load in sending this information into the Sentry or writing a log and you will not have clear issue to fix. So, be careful with logs and don’t leave issues unattended. In the end to users it doesn't matter on what code line your site or application is falling or slowing down. So, do not stop on queries and look further what you can do about your site, plugin or theme performance.

18. If your queries are too complex and expensive to build, consider denormalization and creating a separate pivot table for your purpose with index columns that will fit your requests. WordPress is providing a convenient API for managing custom tables. The main problem will be to update it when needed. Many popular plugins now store data in such tables but from time to time users are experiencing incorrect data due to

these tables going async for some reason and there should be a way to rebuild this table from scratch from the original data, if it went out of sync.

19. Use Query Monitor plugin during development and optimization to find if you have duplicate queries, expensive and some unnecessary queries that can be avoided.

Note that due to different types of caching, main operational system load and scheduled actions, the outcome will be different on each page reload.

20. Use caching

When you will see the amount of requests that are performed on the page even apart from all logic that is building the page - filtering the content, building head, enqueuing ng styles and scripts, it will be obvious that cache has to be used for better performance even if you optimised the whole flow and there is nothing useless. Caching is out of scope of this article, but apart from plugins that cache full pages, there is browser cache, object cache, OPcahce, mySQL cache, CDNs for static files to work with.

21. If you are just starting to build an application, you have a great opportunity to fix many performance issues before they accrue.