Generate a SQL query on WordPress

PHP, SQL, Wordpress

Web Skills : 5

Wordpress is not magic. It relies on a database to store post data. And a database gives the possibility to launch SQL queries to sort them according to certain criteria. And Wordpress has prepared everything, including actions that allow you to interfere in a query.

Published Sunday December 15th 2019, 13:42
Modified Monday August 26th 2024, 10:06

By Olivier Paudex

Introduction

An archive page without a search filter is simply not useful. The previous chapter**"Form and search filter"** gave the steps to build a form using Elementor on the archive page and to retrieve the data in the browser query bar. The following detailed explanation will allow this new URL to query the WordPress database using SQL.

Displaying the SQL query

The archive page of the **"travels"** section is starting to look like something. In addition to displaying posts of the same type, it displays a form made with Elementor. This form allows us to enter search criteria. The next part will allow us to create the SQL query.

First of all, and to separate the PHP code, we will create a new file named **"Travels_Filter.php"**.

- Add its name to the **"functions.php"** file, then create it in the **"php"** folder.

```
// Travelsinclude_once ($dir . '/php/Travels_Filter.php');
```

- In the **"Travels_Filter.php"** file, create the **"dump_request_travels"** function as below.

> This function is only useful during the creation of our archive page. It uses the **"posts_request"** hook, which displays the SQL query in plain text, in order to understand it and correct it if necessary. In production, you will have to delete the code or put it as a comment.

```
<?php
  /**
  * Plugin Name: Travels_Filter
  * Description: Travels archives filter
  * Author: Olivier Paudex
  * Author Web Site: https://www.fuyens.ch
  */
  function dump_request_travels ($input,$query) {
    // Run only on travels archive
    if (!is_admin() && $query->is_post_type_archive('travels')) {
      var_dump ($input);
    }
    return $input;
  }  add_filter ('posts_request', 'dump_request_travels',10,2);
```

Reload your archive page and the SQL query will appear at the top of it.

The query variables

In the SQL query, the parameters on which the action will take place are of the type **"?s=bern&travel_country=switzerland"**. If the **"s"** parameter is well known to WordPress as being the search parameter, all the others are unknown. WordPress does not know what to do with such information, so it thinks it is the name of a page and tries to display it. But since it doesn't exist, it will display **a 404 error page, page not found**

The code below will remedy this. The **"add_query_travels"** function will execute a **"query_vars"** filter so that WordPress can see these new query variables. These are called **"query vars"**, in WordPress

terminology.

All the variables are to be written in an array (here **$vars**). And to embellish the whole, it is possible to replace the famous variable **"s"** by another one much more meaningful like **"search"**.

> Be careful, if you follow me in the example, don't forget to replace the **"s"** parameter by **"search"**, in the **"travels_filter"** function, seen in the previous chapter.

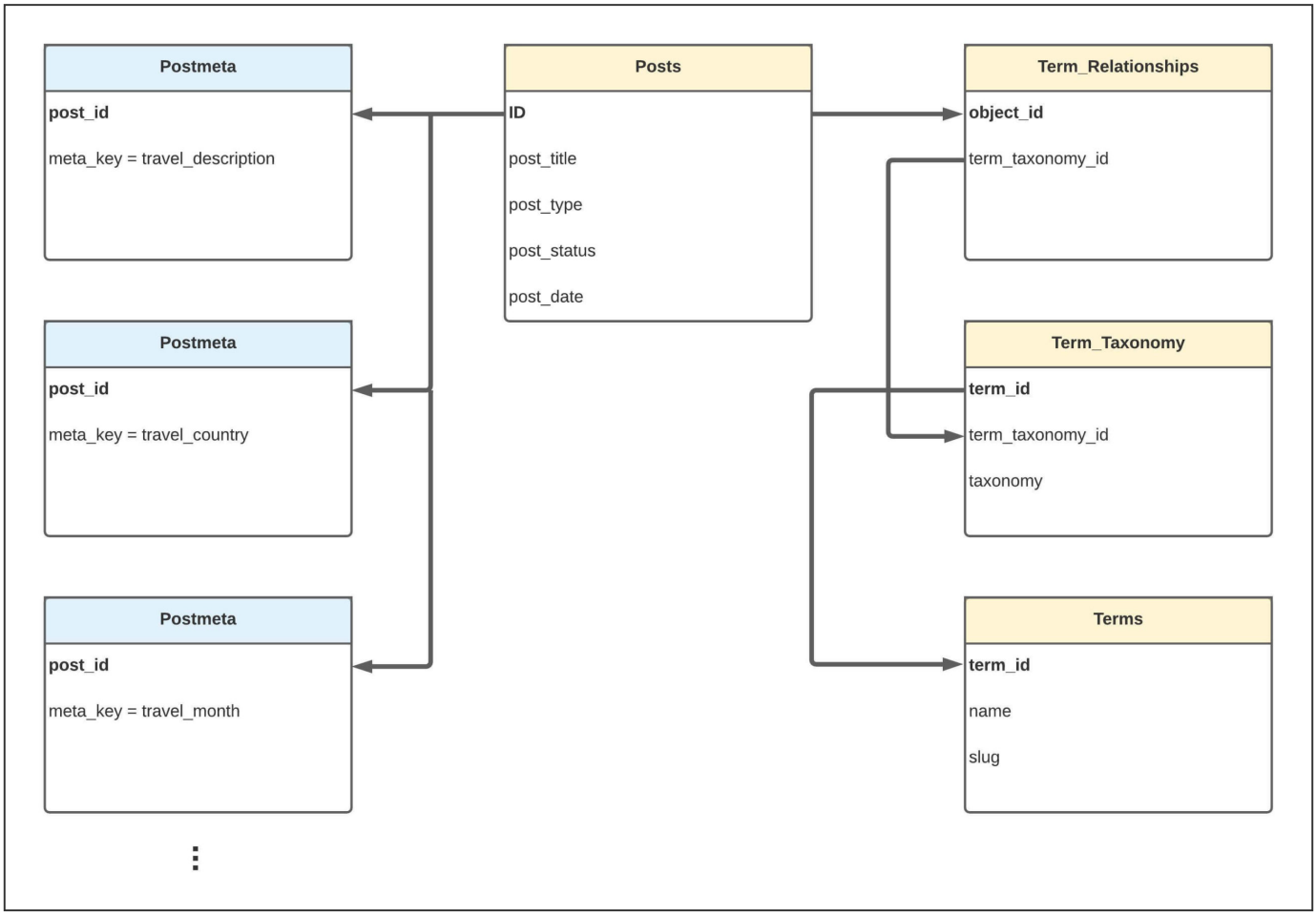
```
function add_query_travels_vars_filter ($vars) {

    // Global
    $vars[] .= 'search';
    // Type travels (ACF)
    $vars[] .= 'travel_description';
    $vars[] .= 'travel_country';
    $vars[] .= 'travel_month';
    $vars[] .= 'travel_year';
    $vars[] .= 'travel_taxonomy';
    $vars[] .= 'travel_order';

    return $vars;
  }  add_filter('query_vars', 'add_query_travels_vars_filter');
```

WordPress database diagram

However, creating the **"query vars"** will not solve everything. You still need to write the SQL query correctly. To do this, a quick look at the WordPress schema is necessary.

WordPress is composed of **twelve tables** in its basic version. Below, I show you the schema of **five tables** , without the cardinalities relations, that make up WordPress and on which this chapter will be based. The tables with yellow headers are tables that physically exist in the database. Tables with blue headers (**Postmeta**) are actually only one table. In the following example, they all represent **"query vars"**, except for **"travel_taxonomy"** which will look for data in the three yellow tables on the right. Finally, **"travel_order"** which is not a data but a sort query, does not use a table as such.

The rule is simple. All custom fields created with the **ACF** plugin add a pseudo table **"postmeta"** with a blue header.



An extract of the WordPress database diagram

The joints

The first step of our code will be to join the tables together. Basically, this represents the arrows. To create the joins in SQL, WordPress has provided a filter called **"posts_join"**.

- Start by creating a **"travels_posts_join"** function based on this filter.

The function has two arguments **($join and $query)**. The first one is nothing else than the result of the join which is returned at the end of the function. The second is the query itself.

Finally, we initialize a global variable, **"$wpdb"**, which is an instance of the wpdb class. It allows to connect and to talk with the database.

```
function travels_posts_join ($join, $query) {

  global $wpdb;
  return $join;
}add_filter ('posts_join', 'travels_posts_join',10,2);
```

Limit the execution of the function

The function, as described above, will run on all the pages of the site, if we do not limit its execution. Here, we want it to run only on the **"travels"** archive page. This is possible thanks to the WordPress function **"is_post_type_archive()"**.

- Add this line to the above code.

```
// Searching and not in admin
if (!is_admin() && $query->is_post_type_archive('travels')) {
}
```

Reset the query

First of all, you have to reset the query. This is simply done by deleting the content of the output parameter **"$join"**.

- Add this line to the above code.

```
// Reset initial join (!! Very important !!)$join = '';
```

Prefix and collecting of table names

Each WordPress table has a prefix number and starts with the acronym **"wp"**. In the example, the **"Posts"** table is actually named **"wp_123456_posts"**. The 6-digit number is unique for a database. To make reading easier, it is possible to store these table (or field) names in variables. The prefix is retrieved with the syntax **"{$wpdb->prefix}"**.

- Add this line to the above code.

```
// Field names all$post_ID = "{$wpdb->prefix}posts.ID";
```

The above example stores the name of the **"ID"** field of the **"Posts"** table in the **"$post_ID"** variable.

The rest of the code is in the same style. For each custom field :

A variable is created to store the name of the **"postmeta"** table.
A variable is created to store the name of the **"postmeta.post_id"** field.
A variable is created to store the name of the **"postmeta.meta_key"** field.

Then, for taxonomy :

A variable is created to store the name of the **"term_relationships"** table.
A variable is created to store the name of the **"term_relationships.object_id"** field.
A variable is created to store the name of the **"term_relationships.term_taxonomy_id"** field.
A variable is created to store the name of the **"term_taxonomy"** table.
A variable is created to store the name of the **"term_taxonomy.term_taxonomy_id"** field.
A variable is created to store the name of the **"term_taxonomy.term_id"** field.
A variable is created to store the name of the **"terms"** table.
A variable is created to store the name of the **"terms.term_id"** field.

```
// Field names travels
$meta_travel_description = "{$wpdb->prefix}postmeta travel_description";
$meta_travel_description_ID = "travel_description.post_id";
$meta_travel_description_key = "travel_description.meta_key";
$meta_travel_country = "{$wpdb->prefix}postmeta travel_country";
$meta_travel_country_ID = "travel_country.post_id";
$meta_travel_country_key = "travel_country.meta_key";
$meta_travel_month = "{$wpdb->prefix}postmeta travel_month";
$meta_travel_month_ID = "travel_month.post_id";
$meta_travel_month_key = "travel_month.meta_key";
$meta_travel_year = "{$wpdb->prefix}postmeta travel_year";
$meta_travel_year_ID = "travel_year.post_id";
$meta_travel_year_key = "travel_year.meta_key";
// Field names taxonomy
$tr_travels_terms = "{$wpdb->prefix}term_relationships tr_travel_terms";
$tr_travels_terms_ID = "tr_travel_terms.object_id";
$tr_travels_tt_terms_ID = "tr_travel_terms.term_taxonomy_id";
$tt_travels_terms = "{$wpdb->prefix}term_taxonomy tt_travel_terms";
$tt_travels_terms_ID = "tt_travel_terms.term_taxonomy_id";
$tt_travels_t_terms_ID = "tt_travel_terms.term_id";
$t_travels_terms = "{$wpdb->prefix}terms t_travel_terms";
$t_travels_terms_ID = "t_travel_terms.term_id";
```

Finally, the joints are created using the code below. The code is much more readable than if, for each table and each field, the prefix would have been added.

```
// Join clauses travels
$join .= " LEFT JOIN $meta_travel_description ON ($post_ID = $meta_travel_description_ID)";
$join .= " AND $meta_travel_description_key = 'travel_description'";
$join .= " LEFT JOIN $meta_travel_country ON ($post_ID = $meta_travel_country_ID)";
$join .= " AND $meta_travel_country_key = 'travel_country'";
$join .= " LEFT JOIN $meta_travel_month ON ($post_ID = $meta_travel_month_ID)";
$join .= " AND $meta_travel_month_key = 'travel_month'";
$join .= " LEFT JOIN $meta_travel_year ON ($post_ID = $meta_travel_year_ID)";
$join .= " AND $meta_travel_year_key = 'travel_year'";
$join .= " LEFT JOIN $tr_travels_terms ON ($post_ID = $tr_travels_terms_ID)";
$join .= " LEFT JOIN $tt_travels_terms ON ($tr_travels_tt_terms_ID = $tt_travels_terms_ID)";
$join .= " LEFT JOIN $t_travels_terms ON ($tt_travels_t_terms_ID = $t_travels_terms_ID)";
```

Orderly classification

The rest of the code will allow you to manage the order of the posts, either from the oldest to the most recent (ascending order) or the opposite (descending order).

I skip the first lines of code which only repeat the principle set up for the joints.

The line that starts with **"$query->query_vars"**, retrieves the data from the **"travel_order"** field, if it exists, and stores it in a variable **"$travel_order"**. Otherwise, the variable takes as default value the word **"desc"**, to mean that the ranking will be done in descending order.

- Add these lines in a new function **"travels_posts_orderby"**, which will use the **"posts_orderby"**

filter.

```
function travels_posts_orderby ($orderby, $query) {
  global $wpdb;
  // Searching and not in admin
  if (!is_admin() && $query->is_post_type_archive('travels')) {
    // Reset initial orderby (!! Very important !!)
    $orderby = '';
    // Tables names
    $post_date = "{$wpdb->prefix}posts.post_date";
    // Get the GET parameters
    $query->query_vars['travel_order'] ? $travel_order = trim(rawurldecode($query->query_vars[
    // Order by clause
    $orderby .= " $post_date " . $travel_order;
  }
  return $orderby;
}add_filter ('posts_orderby', 'travels_posts_orderby',10,2);
```

Criteria of the the query

To finish the code, we will have to create a last function named **"travels_posts_where"** which uses the **"posts_where"** filter. The first lines are identical to the join and sort functions.

Then comes the writing of the **"where"** clause. These always start by calling the method **"$wpdb->prepare()"**. It allows to kill two birds with one stone. The first one allows to concatenate the **"where"** clauses. The second one allows to manage the security of what is called in the jargon, the SQL injection. It would allow malicious visitors to inject SQL code from a browser, for example, to launch deletion orders.

> Managing URL security in a website is certainly not an easy task, but preparing SQL queries in advance, allowing only a certain syntax, prevents SQL injection automatically.

- Add this new function after the others.

```
function travels_posts_where ($where, $query) {

  global $wpdb;
  // Searching and not in admin
  if (!is_admin() && $query->is_post_type_archive('travels')) {

    // Reset initial where ((!! Very important !!))
    $where = '';
    // Field names all
    $post_ID = "{$wpdb->prefix}posts.ID";
    $post_title = "{$wpdb->prefix}posts.post_title";
    $post_content = "{$wpdb->prefix}posts.post_content";
    $post_type = "{$wpdb->prefix}posts.post_type";
```

```
    $post_status = "{$wpdb->prefix}posts.post_status";
    $post_author = "{$wpdb->prefix}posts.post_author";
    // Field names travels
    $meta_travel_description = "{$wpdb->prefix}postmeta travel_description";
    $meta_travel_description_key = "travel_description.meta_key";
    $meta_travel_description_value = "travel_description.meta_value";
    $meta_travel_country = "{$wpdb->prefix}postmeta travel_country";
    $meta_travel_country_key = "travel_country.meta_key";
    $meta_travel_country_value = "travel_country.meta_value";
    $meta_travel_month = "{$wpdb->prefix}postmeta travel_month";
    $meta_travel_month_key = "travel_month.meta_key";
    $meta_travel_month_value = "travel_month.meta_value";
    $meta_travel_year = "{$wpdb->prefix}postmeta travel_year";
    $meta_travel_year_key = "travel_year.meta_key";
    $meta_travel_year_value = "travel_year.meta_value";
    $t_travels_terms = "{$wpdb->prefix}terms t_travel_terms";
    $t_travels_terms_slug = "t_travel_terms.slug";
    $tt_travels_terms = "{$wpdb->prefix}term_taxonomy tt_travel_terms";
    $tt_travels_terms_taxonomy = "tt_travel_terms.taxonomy";
    // Prepare the placeholder for the post_type
    $custom_post_type_placeholder = '%s';
    $custom_post_type = "travels";
    // Get the GET parameters
    $query->query_vars['search'] ? $search_text = trim(rawurldecode($query->query_vars['search
    $query->query_vars['travel_country'] ? $travel_country = trim(rawurldecode($query->query_va
    $query->query_vars['travel_month'] ? $travel_month = trim(rawurldecode($query->query_vars[
    $query->query_vars['travel_year'] ? $travel_year = trim(rawurldecode($query->query_vars['t
    $query->query_vars['travel_taxonomy'] ? $travel_taxonomy = trim(rawurldecode($query->query_
    // Write the where clause
    if (!empty($search_text)) {
      $where .= $wpdb->prepare(" AND (($post_title LIKE '%%%s%%')", $search_text);
      $where .= $wpdb->prepare(" OR ($post_content LIKE '%%%s%%')", $search_text);
      $where .= $wpdb->prepare(" OR ($meta_travel_description_key = 'travel_description' AND $m
    }
    // Where clause travels
    if (!empty($travel_country)) {
      $where .= $wpdb->prepare(" AND ($meta_travel_country_key = 'travel_country' AND $meta_tra
    }
    if (!empty($travel_month) && $travel_month != pll__('all')) {
      $where .= $wpdb->prepare(" AND ($meta_travel_month_key = 'travel_month' AND $meta_travel_
    }
    if ($travel_year > 0) {
      $where .= $wpdb->prepare(" AND ($meta_travel_year_key = 'travel_year' AND $meta_travel_ye
    }
    if (!empty($travel_taxonomy) && $travel_taxonomy != 'tous')) {
      $where .= $wpdb->prepare(" AND $t_travels_terms_slug = %s", $travel_taxonomy);
      $where .= $wpdb->prepare(" AND $tt_travels_terms_taxonomy = %s", 'travel_type');
    }
    // Where clause all
    $where .= $wpdb->prepare(" AND $post_type IN ($custom_post_type_placeholder)",$custom_post_
    $where .= " AND ($post_status = 'publish'";
    $where .= " OR $post_author = 1";
    $where .= " AND $post_status = 'private')";
    // Group by
    $where .= " GROUP BY $post_ID";
  }
  return $where;
}add_filter('posts_where', 'travels_posts_where',10,2);
```

And WP_Query, then...

Some WordPress experts will tell me that there are **"WP_Query"** functions, embedded in the core of WordPress. The answer to this last remark is that **"WP_Query"** does not allow to be as flexible, especially in algebraic **AND and OR** relations.

I also use **"WP_Query"**, for example to display the posts on the homepage or the mini posts. They are the subject of a chapter **WP Query with Elementor**.

The final word

This ends this chapter, certainly the one that took me the most time to code to get to my goal. It is fast, very handy and adapts to any situation. If you want to create other custom types, no problem. Just copy and paste the code into a new file and adapt the variable names.

The next chapter will deal with URL rewriting. Indeed, writing a URL in the way described in this post is not very **SEO** friendly. **SEO** is a term that literally means **"search engine optimization"**. Many search engines, including Google, rely on URLs to index the pages of a website. If these contain parameters such as **"?s=bern&travel_country=switzerland"**, you are not likely to receive a good rating and will be positioned at the bottom of the rankings.

That's it, follow me on this topic in the next post **"SEO and URL rewriting"**.