Extracted from:

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

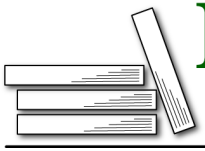
Peter Ullrich
Edited by Michael Swaine

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

Peter Ullrich

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-973-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2023

Paginating in the Database

As usual, we start with adding the new functionality to our Meerkats context. Open up lib/meow/meerkats.ex and add the following code:

```
def list_meerkats_with_total_count(opts) do
  query = from(m in Meerkat) |> filter(opts)

  total_count = Repo.aggregate(query, :count)

  result =
    query
    |> sort(opts)
    |> paginate(opts)
    |> Repo.all()

  %{meerkats: result, total_count: total_count}
end

defp paginate(query, %{page: page, page_size: page_size})
     when is_integer(page) and is_integer(page_size) do
  offset = max(page - 1, 0) * page_size

  query
  |> limit(^page_size)
  |> offset(^offset)
end

defp paginate(query, _opts), do: query
```

As you can see, we create a new list_meerkats_with_total_count/1 function instead of extending the existing list_meerkats/1 function. We could also rewrite the existing list_meerkats/1 function; however, since we change the return type of the function, we'll create a new function instead. We return the total count of all entries affected by the query since we need this information for our pagination to work. Let's take a step back and understand why this is necessary.

In the UI, we want to show how many pages our data has so that the user can click through them. To do so, we need to calculate the total number of pages so that we can show one button per page. We can calculate the number of pages if we know the total count of the entries and the current page size. The total number of pages is the total count divided by the page size, rounded up. So, when we have 90 entries and a page size of 20, it's 4.5 rounded up, so 5 pages in total. Later on, we'll see how we can adjust the page size in the UI, but we need to calculate the total count in our Meerkats context whenever we fetch the meerkat data.

That's what our new list_meerkats_with_total_count/1 function is for. It sorts, filters, and fetches our meerkat data just as before, but it also returns the total count of all entries that match our query. Additionally, it only returns the number

of entries that we specify with the page size. So, instead of returning *all* meerkat data, it only returns, for example, 20 rows, which is exactly what we want to achieve with our pagination.

A Small Deep Dive into How Pagination Works

We implement the pagination by adding the `limit/2` and `offset/2` statements to our query. The `limit/2` call instructs the database to return only the number of entries we specify. The `offset/2` statement instructs the database from where it should count the number of entries to return. As an example, if we set the offset to 10 and the limit to 20, the database will return 20 entries counting from the eleventh row, effectively ignoring the first 10 entries.

This is how we achieve the pagination. We calculate the `offset` by multiplying the current page number by the page size. In the UI, we present the page numbers as starting from 1, but in our database, the first page is page 0. That's why we subtract 1 from the page number before calculating the offset. Let's look at an example:

Let's say we want to fetch the data for the first page with a page size of 10. We first subtract 1 from the page number, receiving a new page number of 0. Then we multiply the 0 with our page size of 10, and we get an offset of 0. This means that the database will return 10 entries beginning with the very first row, which is exactly what we want.

Now, let's say we want the data for the second page. We subtract 1 from the page number of 2, resulting in a new page number of 1. We multiply it with our page size of 10 and get an offset of 10. This means that the database will skip the first 10 entries and return entries from the eleventh row onward. So, for our second page, we return the second chunk of data, which is exactly what we want as well.

As a safety precaution, we use the `max/2` function to ensure that our page number is always equal to or greater than 0. In case we make a mistake somewhere and set the page number to 0 or even -1, this will fix the mistake by replacing it with 0.

A Short Comparison of Offset vs. Cursor Pagination

The pagination approach just described is called *offset pagination*. It's a popular approach to pagination because it's relatively simple and easy to understand. It lets the user jump to any page in the data. However, for very large datasets (we're talking multiple millions of rows here) and for real-time data presentation, offset pagination is unsuitable. The reason is that the

database needs to count many rows to fetch, for example, the ten-millionth row.

The time complexity of offset pagination is *O(offset + limit)*, which means that it has *linear complexity*. So, if you have a large dataset and you need to fetch the last page of the data frequently, offset pagination might not be efficient enough for your use-case since it will take potentially very long until your query returns the data.

Given this drawback, offset pagination is unsuitable for real-time data presentation where new data is continuously appended to the table and the most recent data is fetched often. Your database needs to count and ignore many rows every time you fetch the latest data and the number of counted rows increases as new data is added.

Another problem with offset pagination is that you might return repeated rows to the user if you add data to a previous page. As an example, imagine that we sort the meerkat data by the meerkat's name and we chunk the data into pages of ten rows each. Furthermore, imagine that the first page has ten rows of names beginning with *A* and the second page has ten rows with names beginning with *B*. So, if the user request the first page, they will see ten names beginning with *A*.

Now, imagine we add another meerkat whose name begins with an *A* while the user views the first page in their browser. Now, if the user moves to the second page, suddenly they see a page containing one name beginning with *A* and nine names beginning with *B*! Worse even, they might have seen the name beginning with *A* on the previous page already! This happens because the second page returns the tenth to the twentieth entries, regardless of which data you presented previously. Since we now have eleven names with *A* and ten with *B*, it returns the eleventh name with *A* and nine names with *B*. Hence, the user see a repeated row with *A*, which might cause an inferior user experience.

So, if you have a very large dataset, need to fetch rows at the end of the table often, and add data anywhere in the dataset frequently, offset pagination might not be for you. In such cases, you might want to look into *cursor pagination* instead.

In short, cursor pagination uses *previous* and *next* pointers to indicate the upper and lower bounds of the current page. As an example, imagine you are on the third page of the meerkat data and use the `id` field to paginate the data. In this case, your previous, or lower bound, pointer would be 20 and the next,

or upper bound, pointer would be 29 because we are 0-based here. So, how can you use these pointers to request the previous or the next page?

Fetching the next page is relatively simple. The query would look something like this:

```
from(m in Meerkat,
  where: m.id > ^29,
  order_by: [asc: :id],
  limit: 10
)
```

So, using the next pointer, we can simply fetch the next page by filtering out any meerkat data with an id lower or equal to the next pointer, which is 29 in our case. This way, we fetch the thirtieth entry up to and including the thirty-ninth entry, which is equal to the fourth page of our data. So far, so good. However, fetching the previous page is slightly more complicated. Let's have a look at how a query could look:

```
from(m in Meerkat,
  where: m.id < ^20,
  order_by: [desc: :id],
  limit: 10
)
|> Repo.all()
|> Enum.sort_by(& &1.id, :asc)
```

First, we filter out any data with an id equal to or higher than our previous pointer, which is 20 in our case. We receive a dataset of twenty entries with IDs from 0 to 19. So far so good. However, we want to fetch the tenth until the nineteenth entry, which is why we sort the entries descending by their ID. This results in a dataset where the entry with the ID 19 comes first, followed by 18, 17, and so on. From that dataset, we then take ten entries as indicated by the limit: 10 instruction. This returns the entries with an ID from 19 down to 10, which is the data we expect in the second page of our dataset. However, the data is sorted descending by ID, which is why we inverse the order using Enum.sort_by/3 before we return the data from the function.

Phew! As you can see, implementing a cursor-based pagination is definitely more complex than the offset-based alternative. But the big advantage of cursor-based pagination is that it uses the where instruction instead of the offset instruction. The advantage of where over offset is that Postgres efficiently ignores any rows that don't match the where statement, whereas it counts and skips the rows before the given offset. If the offset is high, Postgres counts every single row until the offset, which can result in a full-table scan in the worst

case. With `where`, it first filters out any unsuitable rows before starting to count. This results in a much more efficient query.

However, the efficiency gains of cursor pagination largely depend on properly configured indices for the variables that you want to use for pagination. Our primary key `id` already has an index, but if you wanted to paginate using the `name` variable, you would have to create the proper index yourself. Otherwise, Postgres still needs to check every row whether it matches the `where` statement or not, and your cursor pagination won't be better than a simple offset pagination.

Cursor pagination solves the problem with showing repeated entries, as well, since it doesn't rely on offsets but uses pointers to its upper and lower bound entries instead. In our previous example, the upper bound of the first page would be the last name with *A*. So, when the user requests the second page, our query would only show names beginning with *B* since $B > A$. Even if we added another meerkat with a name beginning with *A* in the meantime, the second page would only show entries *after* the last name with *A*, so all names with *B*.

The big downside of cursor pagination, aside from its implementation complexity, is that users can't jump to specific pages in the dataset. Instead, they have to browse through every single page before the page that they want to access. This is because we don't have an *offset* anymore but only upper and lower bounds pointers. If this is a common use-case for your users, consider using offset pagination instead.

In summary, for small datasets whose last page is rarely accessed, the offset pagination is a simple and suitable solution. If your dataset is very large and you need to access its latest data frequently, a cursor-based pagination might be more suitable. In this book, we'll implement an offset-based pagination since it's easier to understand and implement. Also, once you have the offset pagination in place, it's a relatively small step to switch to curser-based pagination.

Alright! Now that we discussed how the pagination works on the database side, let's see how we can expose the pagination functionality to the user. As usual, we'll create a LiveComponent to encapsulate the UI logic. Let's see what that looks like.