Understanding the Offset and Cursor Pagination

A quick look at the pagination algorithms

Jason Ngan

Photo by Monstera on Pexels

As the volume of data increases, pagination becomes an integral part of software development.

Instead of returning a huge chunk of data in a request, pagination divides and returns data to clients in smaller batches.

The cursor and offset pagination are the two most commonly used algorithms and they carry their respective tradeoffs.

Let's dive deep into how they work today.

Offset Pagination

The offset pagination leverages the `OFFSET` and `LIMIT` commands in SQL to paginate data.

Implementation

Paginate using the OFFSET & LIMIT command in SQL

Said we are implementing an API to get a list of user information.

```
API: GET /api/user/list

request: {
 page_size: 10,
 page_number: 3
}
```

In each request, clients pass a `page_size` (offset) and a `page_number` (limit).

- Page size indicates the number of data to be returned.
- Page number indicates the current requesting page.

```
SELECT COUNT(*) AS total FROM user_tab;
```

The server first queries the total number of records from the user table.

- This allows the clients to grasp the number of total pages.

```
SELECT * FROM user_tab
ORDER BY id DESC
LIMIT 10 OFFSET 20;
```

The server utilises the offset and limit commands to retrieve ten records from the table.

- Since the given `page_number` is 3, the offset = 10 * 2 = 20.

```
response: {
    "users": [...],
    "paging": {
        "total_record": 295,
        "page": 3,
        "total_pages": 30
    }
}
```

The server returns the paging information to the clients allowing them to keep track of the current and available pages.

Pros

- It allows the clients to view the total number of pages.
- It allows clients to jump to a specific page by passing the page number.

Cons

Result inconsistency

- If an item in a previous page is deleted, data will shift forward, causing some results to be skipped.
- If an item in a previous page is added, data will shift backwards, causing some results to be duplicated.

Offset inefficiency — Doesn't scale well with large dataset

- The database looks up for (offset + limit) number of records before discarding the unwanted ones and returning the remaining.
- Hence, the query time increases drastically as the offset increases.

Cursor Pagination

The cursor pagination utilizes a **pointer** that refers to a specific database record.

Implementation

Clients provide a cursor that points to a unique database record

```
API: GET /api/user/list

request: {
 cursor: 12345,
 page_size: 10
}
```

In each request, clients pass a `cursor` and a `page_size`

- The cursor refers to a specific unique value in the database.
- If the cursor is not given, the server fetches from the first record.

```
SELECT * FROM users
WHERE id <= %cursor
ORDER BY id DESC
LIMIT %<limit + 1>
```

The server fetches (`limit + 1`) records whose ID is smaller than the cursor value.

Note that the limit is equal to the given page size plus one.

- If the number of records returned is less than the `LIMIT`, it implies that we are on the last page.

- The extra record is not returned to the client. The `ID` of the extra record is passed back to the client as the `next_cursor`.

```
response: {
    "users": [...],
    "next_cursor": "12335",  # the user id of the extra result
}
```

Pros

Stable pagination window

- Since we are fetching from a stable reference point, the addition or deletion of record will not affect the pagination window.

Scale well with large datasets

- The cursor is unique and indexed.
- The database jumps directly to the record without iterating through the unwanted data. Hence, making it more efficient.

Cons

- The cursor pagination doesn't allow clients to jump to a specific page.
- The cursor must come from a unique and sequential column (E.g. timestamp). Otherwise, some data will be skipped.
- Limited sort features. If the requirement is to sort based on a non-unique column (E.g. first name), it will be challenging to implement using cursor pagination. Concatenating multiple columns to get a unique key leads to slower time complexity.

Encoded Cursor

The encoded cursor suggests returning an **encoded base64 string** regardless of the underlying pagination solution.

When using offset pagination, we encode the `page_number` and `total_page`

into a base64 string and return it as a cursor to the clients.

```
response: {
    // "page=3|offset=20|total_pages=30"
    next_cursor: "dcjadfaXMDdQTQ"
}
```

Similarly, we can encode the cursor in the cursor pagination into a base64 string before returning it to the clients.

```
response: {
    // "next_cursor:1234"
    next_cursor: "dcjadfaXMDdQTQ"
}
```

The client can always pass a `cursor` and a `page_size` without knowing the underlying implementation.

```
request: {
    cursor: "dcjadfaXMDdQTQ",
    page_size: 10
}
```

This allows the server to implement different underlying pagination solutions while providing a consistent interface to the API consumers.

Summary

Pagination is an inevitable yet tricky topic in software development.

While there isn't a one-size-fits-all solution, knowing the pros and cons allows us to make better tradeoffs when designing our next API.

I hope you will find this helpful, and I will see you at the next one!

If you are interested in articles like this, join me and sign up for Medium today!

Join Medium with my referral link — Jason Ngan

Read every story from Jason Ngan (and thousands of other writers on Medium). Your membership fee directly supports...