# Pagination and Streaming Data within Distributed Database

In this article I would like to share my experience dealing with large dataset hosted on Data Lake using Impala, I will cover three different steps I used to achieve my target, which is perform better and scale horizontally when the client asks large dataset.

## Quick overview about Distributed Database

A **distributed database** is a [database](https://en.wikipedia.org/wiki/Database) in which [storage devices](https://en.wikipedia.org/wiki/Computer_data_storage) are not all attached to a common [processor](https://en.wikipedia.org/wiki/Processor_(computing)).[[1]](https://en.wikipedia.org/wiki/Distributed_database#cite_note-1) It may be stored in multiple [computers](https://en.wikipedia.org/wiki/Computers), located in the same physical location; or may be dispersed over a [network](https://en.wikipedia.org/wiki/Computer_network) of interconnected computers. Unlike [parallel systems](https://en.wikipedia.org/wiki/Parallel_computing), in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components.

For whom does not know about Impala/Cloudera and Data Lake checkout the following links

Pagination Keyset

Streaming data

Pagination OFFSET/LIMIT

Most of developers we use to apply Pagination using OFFSET and LIMIT ordered by some field, without caring much about the side effect of this solution, especially in distributed system.

OFFSET has been designed:

*the rows are first sorted according to the <order by clause> and then limited by dropping the number of rows specified in the <result offset clause> from the beginning*

*Big OFFSETS impose a lot of work on the database no matter whether SQL or NoSQL*

So let’s break it down why Pagination using OFFSET is not suitable in distributed system.

* The data are distributed in multiple computers / locations
* the more deep you paginate more the performance goes down
* Uses huge amount of memory
* Overhead elaboration Database side

What is really happening when we use OFFSET [see the following figure]

Request

1st Page 100 Rows 100 Rows

Coordinating

Return TOP 100 of 500 rows

100 100 100 100 100

In this scenario the client want to get the first 100 rows having 5 distributed nodes/servers/partitions, the query will be executed to all available nodes and each of them will get back with its top 100 records then the coordinating will in care about all dataset by merging them and returning the Top 100 (sorted) through 500 rows to the client.

Now imagine that the client ask for page 1,000—results 10,001 to 10,100. Everything works in the same way except that each node has to produce its top 10,100 results. The coordinating node then sorts through all 50,500 results and discards 50,400 of them!

<https://www.quora.com/Which-algorithms-can-sort-data-that-is-split-across-multiple-machines>

Based on the above analysis, from multiple vendors, for a distributed query system such as Impala, HBase, Elasticsearch,MongoDB, or Spark, the **O(cost)** of a single pagination request expensive, equivalent to requesting the entire dataset.

This is in line with Cloudera’s recommendation to avoid using paging when running impala queries:

In Impala 1.2.1 and higher, you can combine a LIMIT clause with an OFFSET clause to produce a small result set that is different from a top-N query, for example, to return items 11 through 20. This technique can be used to simulate "paged" results. Because Impala queries typically involve substantial amounts of I/O, use this technique only for compatibility in cases where you cannot rewrite the application logic. For best performance and scalability, wherever practical, query as many items as you expect to need, cache them on the application side, and display small groups of results to users using application logic.

<https://www.cloudera.com/documentation/enterprise/5-9-x/topics/impala_offset.html>

*This issue applies not only for Impala, but for all distributed systems, the cost of sorting results grows exponentially the deeper we page.*

*I hope now you have clear Idea why Pagination using OFFSET is pain in the ass for distributed system;*

What are the other solutions then?

## Keyset/Seek Pagination

It is quite simple and straightforward solution, do not use LIMIT and use WHERE clause ☺

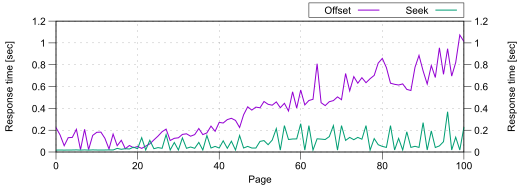
Its is quite interesting that you can achieve the same result with huge performance improvement, but this required additional work in database side which means in order to perform this technique, the Table or Collection in case of Mongo or Document in case of ElasticSearch should have unique and sequential ID. Sequential number Generation in Impala. <http://community.cloudera.com/t5/Interactive-Short-cycle-SQL/Sequence-number-generation-in-impala/td-p/38126>

Example of Keyset:

*Select X,Z,Y From [Table]* **where** *sequential ID >= 0 Fetch 100*

*Select X,Z,Y From [Table]* **where** *sequential ID >= 101 Fetch 100*

Using this technique it drastically improve the performance, the coordinator node doesn’t need to do huge work before to get back to the client with the results.



For more details check out the following links which will give you further information about Keyset

Markus Winands Post

<http://use-the-index-luke.com/no-offset>

https://blog.jooq.org/tag/keyset-pagination/

Conclusion

This technique improves a lot in terms of performance, but what about large dataset:

* When time becomes a major factor in the delivery of a complete data
* When a single large entity is being prepared for http wrangling
* When Memory usage becomes not sustainable during concurrent requests

Then It is not enough using only Keyset pagination.

Here comes useful have Streaming API to stream entire dataset straight to the client without over engineering and adding additional hardware costs.

## Streaming Data using .NET

Streamed data is sent in small chunks whose size is controlled by the Server, in our case by the API Server.

### Streaming Data from Impala using Sequential Access Behaviour

Using JDBC connector it’s possible to stream the data from Impala to the client with additional process such as scrolling all columns.

The Sequential Access Behaviour allows the network streaming from Database, this means the client won’t deal with entire data putting into memory, this behaviour it is really useful especially when you are dealing with Blob data.

await command.ExecuteReaderAsync(**CommandBehavior.SequentialAccess**);

Once the connection has been established and the query executed, the data can be processed sequentially without putting entire dataset into memory instead processed row by row and send it back to client using **PushStreamContent** asynchronously.

### Pros

* Less memory usage
* The API Server can decide how many records to stream
  + The data will be transmitted chuck by chunk
* Faster during Concurrent queries
* The Client can start to process the data as soon as the data becomes available, doesn’t need to wait the completion of the request

### Cons

* Long-running Connection
  + The Connection of Impala remain OPENED for entire streaming completion
* If something fails, the client should Retry and Process everything again, as long as we don’t persist when the streaming has been interrupted this will allow as executing again the query by adding additional filters.

# Final Conclusion

Using Pagination with OFFSET it is good when you really really need to jump from one page to another without ordinary pagination, but this solution costs you a lot in terms of performance.

Using Keyset pagination fit very well when the data to be sent back to the client was not large data set the performance is incredibly better than the OFFSET.

Using Streaming you can push the data as soon as the data becomes available, and the client can start to process the data. Remember this option doesn’t need to paginate the data instead stream entire dataset chunk by chunk. This solution reduces memory usage cross all levels especially for the concurrent request it is possible to scale vertically rather than horizontally.