# Pagination and Streaming Data within Distributed Database

## 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.

In this article I would like to share my experience dealing huge amount of dataset hosted on Hadoop using Impala, I will cover three different steps I used to achieve my target.

The objective is how we can perform better when the client asks huge amount of data filtered by timestamp.

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.

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

Bit of theory about OFFSET

*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*

Let’s break it down why Pagination using OFFSET and LIMIT 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

What is really happening [see the following figure]

Request

1st Page 100 Rows 100 Rows

Coordinating node

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, the query will be executed to all available nodes and each of them will get back with its top 100 records then the coordinating node 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!

What the Cloudera team is suggesting for Impala

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>

In my opinion this wouldn’t for long term solution as long as we are moving the problem from one to another, especially adding high complexity into the client, more client you have more your they have to deal with this issue.

*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 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.

Example of Keyset:

*Select X,Z,Y From [Table]* **where** *sequential ID >= 0 AND sequential ID <= 100 AND [Additional Cond]*

*Select X,Z,Y From [Table]* **where** *sequential ID >= 101 AND sequential ID <= 200 AND [Additional Cond]*

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.

Request

Coordinating node

100 Records

All nodes returns the records satisfies the where clause

Checkout the following links how you can create sequential number generation within Impala

<http://community.cloudera.com/t5/Interactive-Short-cycle-SQL/Sequence-number-generation-in-impala/td-p/38126>

You can find more details about Keyset pagination in Markus Winands Post <http://use-the-index-luke.com/no-offset> or 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 record/s has been transmitted, doesn’t need to wait the completion of the request
* Can be used different Media Formatter

### 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 or provide different filters