Building pipeline for Wikipedia data:

The ETL pipeline has been designed in python. The script extracts the data from the website, transforms it and loads it in the database. The loaded data is then accessed to return results.

The choice of data system depends upon couple of factors:

1. Volume of data stored
2. Type of data stored
3. Users that are going to analyze it

Database systems can be broadly classified into 2 categories:

Relational and NoSQL database.

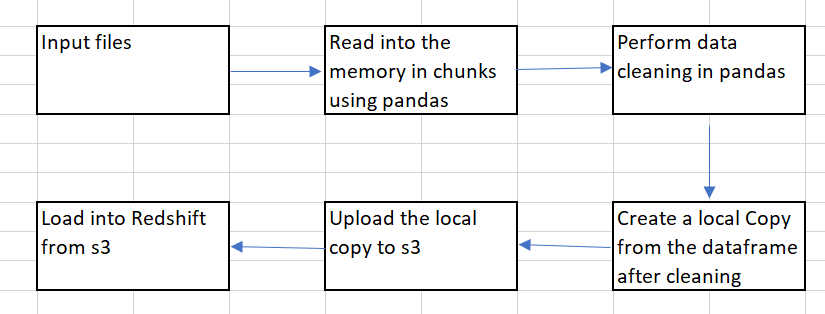
RDBMs are primarily designed for data integrity and not performance. It spends considerable resources to maintain data integrity and ensure that there are no update/delete anomalies. Since in our case, we are dealing with log data, the integrity mechanisms used by RDBMS are wasted. Log analysis is primarily I/O bound. Hence, we can safely eliminate RDBM as our choice of storage system.

Within NoSQL databases there are lot of options to choose from. In my opinion, the ideal database for the given problem would be a Cassandra database. Cassandra is a column-family\* store. It ensures data locality at the partition level, not the column level. This is a big advantage compared to other columnar databases. In our case, since there are very few columns and most of the queries would either be grouped by data or language, we can select these as the partition key and ensure that the data resides together leading to faster query performance. Cassandra databases are highly scalable and provides faster read and write performance. Cassandra scales linearly with the data and hence with increasing data we can simply add more nodes.

Due to my unfamiliarity with Cassandra, I decided to use Redshift. Redshift is a columnar database. Columnar databases reduce the I/O operations drastically. It is very fast when it comes to loading data and querying it. It uses massive parallel processing architecture which allows it to distribute and parallelize queries across multiple nodes. A caveat with the MPP architecture is that it is supported only for uploads from S3, DynamoDB or AmazonEMR. In my initial design, I was directly loading the dataframe to Redshift. However, this was very inefficient as it was doing single row inserts and not making use of the MPP. Hence to make full use of the capabilities of Redshift, I had to first upload the file to s3 and then load into the redshift table from s3. With more input files, we can scale the redshift cluster horizontally by adding more nodes. This takes care of the target database system.

Now coming to the source, one of the major challenge is to load a large dataset. I have used pandas.read\_csv command to read the file. But this becomes a problem for large dataset as it holds the dataframe in memory. There is a high possibility of running into Memory Error if the entire file is read at once. To overcome this, I have read the file in chunks using the chunksize operator. The chunksize can be varied depending on the system configuration. In my case, I have set the chunk size to load 2 million rows. Thus, irrespective of the size of the input file, there would not be a Memory error as the data is read, processed and loaded in chunks.

Hence, the entire operation can be summarized as follows:



This solution gives really good performance and the entire operation takes around 5 minutes to load 1 file. Also, the solution supports reading multiple files in one run.

I also played around with multithreading but it did not help much. Multiprocessing would further reduce the time taken by working on the dataframe chunks in parallel but it hasn’t been implemented.