

# Non-Relational, Heterogeneous Distributed Database Systems

GROUP No. - 7

GROUP MEMBERS

Jatin Arora (13CS10057)

Chinmaya Pancholi (13CS30010)

Ashish Sharma (13CS30043)

MENTOR: Prof. Pabitra Mitra



# INTRODUCTION

**Big Data** is an idea that the amount of data that we generate (and more importantly, collect) is increasing extremely quickly. More importantly, many organizations and companies are recognizing that this data can be used to make more accurate predictions, and therefore, can be leveraged to their advantage, like it can help them to make more money. For example, Facebook knows how often we visit many websites (due to the pervasive 'Like' on Facebook buttons) and wants to use that information to show us ads we are more likely to click on. Palantir wants to help governments fight crime and terrorism by using massive data to predict where issues may be.

In such situations, the number of records may blow-up to huge sizes like hundreds of millions or even billions, especially in cases like data storage for websites like Amazon which aims to perform tasks like personalizing the recommendations shown to its users.

## Hadoop

This huge volume of data now needs to be stored in a way so that it can be easily and efficiently worked upon. Also, the data can be of highly varied sources. But a lot of times, data of varied structures needs to be analysed together. A database which can store varied data in a single file system is called a **Heterogenous Database**.

This requires the use of heterogeneous distributed data management and storage system. One platform suited for this purpose is **Hadoop**.

It organizes the data fed to it in its **HDFS** (Hadoop Distributed File System). The data is split up into chunks and stores in different storage units (**datanodes**) and there is a master, the **namenode** that coordinates with the datanodes and stores an index as to where (in which datanode) a specific piece of information is stored. The communication channels between the namenode and the datanodes is very fast. Hadoop is also a very secure storage system. It replicates the same piece of data multiple times and stores it in separate datanodes to ensure that even if a particular datanode crashes, the data stored in it is not lost and can be recovered using the bits of it stored in the other datanodes. Not only this, Hadoop also maintains a secondary namenode which stores a backup of the namenode, so that even if the namenode crashes, everything can be recovered.

Now, with the data indexed inside Hadoop, when a query is fed, it is interpreted by the namenode, which then extracts the relevant pieces of information from the datanodes, compiles the results and presents it to the user as output.

## HBase

The problem with Hadoop is that it can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs. **HBase** is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable. It is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:

- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.

- Column is a collection of key value pairs.

## MODES OF OPERATION

Hadoop with HBase supports 3 modes of operation:

- **Single Node:** Here there is only a single node which stores the entire data. This is same as any normal relational database system, like SQL database.
- **Pseudo-Distributed System:** Here, various datanodes are made on the same machine. Hence it is distributed file system maintained within a single machine.
- **Fully-Distributed System:** Here, all the datanodes are distributed over a cluster of machines out of which one machine has the namenode and is the master. All other machines which have the datanodes are slaves.

## Google Protocol Buffers

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. It has the ability to store a large amount of structured data in a binary format which is easier to store and more structured than XML format. Several advantages of this format are,

- Simpler
- 3 to 10 times Smaller
- 20 to 100 times faster
- Less ambiguous
- Generates data access classes that are easier to use programmatically

Firstly, a structured format for the data is specified, similar to tags in XML. This format is stored in 'foo.proto' file. This file is then compiled with the ProtoBuf Compiler with the output language specified (c++, java, python etc). The output is a set of functions that can be directly used to read, modify and query components of the objects stored in the processed binary file of the data.

A sample '.proto' file:

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

## Explanation

The object/entity whose information is being stored is Person. To define a person, name, id and email are its attributes. The required attributes are ones which must be defined in each Person record and the optional attributes may or may not be present. Also with each attribute, we include a datatype which tells what kind of data this attribute stores (like integer (int32, int64), character sequence (string) etc).

## Advantage of using Protocol Buffers for BigData

Protocol Buffers reduce the size of the data to be stored by converting it to object form and storing it in the disk as binaries. This saves a huge amount of space when dealing with structured BigData.

Most modern day search engines have to re-index a large amount of data regularly to update their database as content on the internet gets updated. In such situations it is much easier to work with protocol buffers for data updation rather than parsing the XML documents word by word and making the changes.

### Tasks Completed

In this project we build the following 3 kind of Applications using Hadoop and HBase for BigData:

- Implementation of Aggregation Queries using Hadoop Map-Reduce Infrastructure in pseudo - distributed mode.
- Implementation of Aggregation Queries using the tabular structure of Hbase over Hadoop in fully distributed mode.
- Implementation of Top-K Queries using the tabular structure of Hbase over Hadoop in fully distributed mode.

# WORD COUNT USING HADOOP

This Map-Reduce job takes a **text document** as **input** and produces the **word counts** of all words occurring in the document as the **output**.

The test document is fed as input to the mapper.

**Map Phase:** Tokenizes the document and generates (key, value) pairs of the form, (token, 1) as output.

**Shuffle/Sort Phase:** The list of (token, 1) pairs from the map phase are fed as input to this phase, which reorganizes and sorts them on the basis of the key and sends it to the reduce phase. Example:

|                 |        |                 |
|-----------------|--------|-----------------|
| ('this', 1)     | -----> | ('is', 1)       |
| ('notebook', 1) |        | ('notebook', 1) |
| ('is', 1)       |        | ('thick', 1)    |
| ('thick', 1)    |        | ('this', 1)     |
| ('this', 1)     |        | ('this', 1)     |

**Reduce Phase:** This phase aggregates and merges the pairs having the same 'key' and adds their 'value' counts. This is the final output of the word count utility. Example:

|                 |        |                 |
|-----------------|--------|-----------------|
| ('is', 1)       | -----> | ('is', 1)       |
| ('notebook', 1) |        | ('notebook', 1) |
| ('thick', 1)    |        | ('thick', 1)    |
| ('this', 1)     |        | ('this', 2)     |
| ('this', 1)     |        |                 |

# RETRIEVING POPULAR WIKIPEDIA PAGES USING HBase

What do Wikipedia's readers care about? Is Britney Spears more popular than Taylor Swift? Is Virat Kohli more well-known than Sachin Tendulkar? Which was the most viewed page in the last week?

To answer such questions, the publicly available statistics of Wikipedia's page-visits were used and analyzed to retrieve suitable results as per the user's query.

An **Aggregation Query** is a query which operates on the data stored in multiple rows of a table. Thus, the values stored in multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning or measurement. Examples of aggregation queries include sum, average, maximum, minimum, median, mode, count etc.

## Objective

The objective of this system is to retrieve a list of Wikipedia pages by performing aggregation (sum) over their hourly page-counts within a particular timespan.

## Dataset

Each request of a page, whether for editing or reading, whether a "special page" such as a log of actions generated on the fly, or an article from Wikipedia or one of the other projects, reaches one of Wikipedia's squid caching hosts and the request is sent via udp to a filter which tosses requests from Wikipedia's internal hosts, as well as requests for wikis that aren't among Wikipedia's general projects. This filter writes out the project name, the size of the page requested, and the title of the page requested.

Here are a few **sample lines** from one file:

```
fr.b Special:Recherche/Achille_Baraguey_d%5C%27Hilliers 1 624
```

```
fr.b Special:Recherche/Acteurs_et_actrices_N 1 739
```

```
fr.b Special:Recherche/Agrippa_d/%27Aubign%C3%A9 1 743
```

```
fr.b Special:Recherche/All_Mixed_Up 1 730
```

```
fr.b Special:Recherche/Andr%C3%A9_Gazut.html 1 737
```

In the above, the first column "fr.b" is the project name. The following abbreviations are used:

wikibooks: ".b"

wiktionary: ".d"

wikimedia: ".m"

wikipedia mobile: ".mw"

wikinews: ".n"

wikiquote: ".q"

wikisource: ".s"

wikiversity: ".v"

mediawiki: ".w"

Projects without a period and a following character are wikipedia projects.

The second column is the title of the page retrieved, the third column is the number of requests, and the fourth column is the size of the content returned.

These are hourly statistics, so in the line :

```
en Main_Page 242332 4737756101
```

one can see that the main page of the English language Wikipedia was requested over 240 thousand times during the specific hour. In some directories one would see files which have names starting with "projectcount". These are total views per hour per project, generated by summing up the entries in the pagecount files. The first entry in a line is the project name, the second is the number of non-unique views, and the third is the total number of bytes transferred. In short, each file is generated hourly and contains a list of the pages visited in the last hour along with the number of page-visits made to each listed page.

This entire data is stored at : ["https://dumps.wikimedia.org/other/pagecounts-raw/"](https://dumps.wikimedia.org/other/pagecounts-raw/) and has been made publicly available by Wikipedia.

## Methodology

The data (as discussed in 'Dataset' section) was first processed and then inserted suitably into Hbase tables. Then, the stored data was processed and output was generated as per the input given to the system.

- **DATA PARSING AND STORING**

Each file containing the data was parsed to extract the Page Title and the corresponding Page Visit-count and this data was stored along with the exact date and time for which these statistics were collected. The parsed data was stored in Google Protocol Buffers format for efficient and quick handling and reduction in size.

The input XML data is first converted into Protocol Buffer format so that it can be easily stored (as binary) and efficiently fed to Hbase. The format of the data is specified in 'wikicount.proto' file which is then fed to the ProtoBuf Compiler to generate JAVA Functions for accessing and modifying the objects.

### [wikicount.proto](#)

```
package wikicount;
```

```
option java_package = "com.dbms.wikicount";
```

```
option java_outer_classname = "WikiCountProtos";
```

```

message DocInfo {
    optional string doctype = 1;
    optional string name = 2;
    optional int64 visits = 3;
    optional int64 size = 4;
    optional string filename = 5;
}

```

```

message RetrivedDocs {
    repeated DocInfo docinfo = 1;
}

```

### Explanation

The .proto file starts with a package declaration, which helps to prevent naming conflicts between different projects. In Java, the package name is used as the Java package unless the `java_package` has been explicitly specified.

`java_outer_classname` specifies the name of the generated .java file, here, `WordCountProto.java`.

`DocInfo` is the structure which stores the each document record and `doctype`, `name`, `visits`, `size`, `filename` (implicitly storing the date and time information) are its attributes.

`RetrivedDocs` is the structure which stores the collection of documents (as `DocInfo` objects). 'repeated' attribute captures the fact that there can be multiple copied of this attribute withing the `RetrivedDocs` object.

- **INSERTING DATA INTO HBase**

After parsing the files and storing the data in Google Protocol Buffer format, this data was stored into Hbase tables for later retrieval and processing.

The Hbase table used for storing the data has the following format :

| KEY          | FILE DETAILS | COUNT DETAILS |
|--------------|--------------|---------------|
| PageTitle+ID | DateTime     | Visit Count   |

**Fig2.1 Page Count Table Structure**



Here, the key of each row is the page title of the Wikipedia page for which the row contains the details. Also, the table has two column families : File Details and Count Details, each containing one column. Each row, thus, contains two columns : DateTime (for storing the time-period that the statistics belong to) and the VisitCount (for storing the number of times the page was visited in the above mentioned time-span).

The HBase table used for retrieving and storing the final results has the following format :

| KEY       | COUNT DETAILS |
|-----------|---------------|
| PageTitle | Visit Count   |
|           |               |

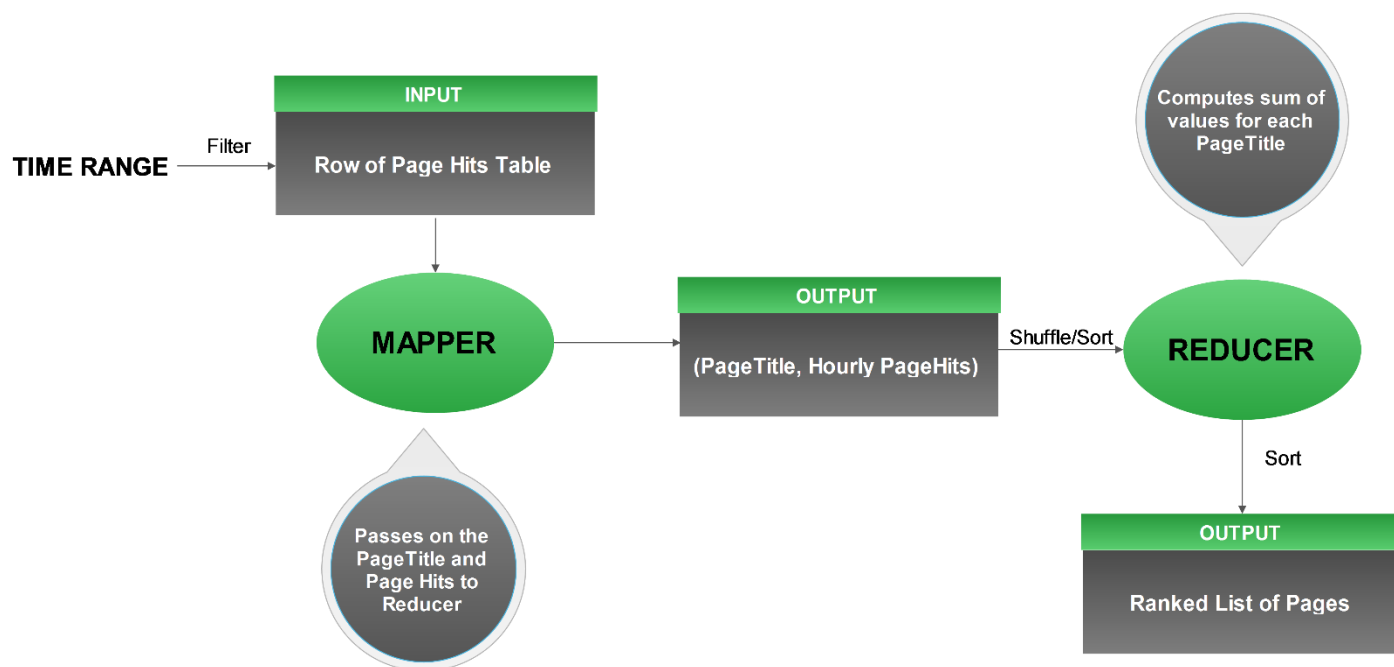
**Fig2.2 Aggregated Results Table Structure**

Here also, the key of each row is the page title of the Wikipedia page for which the row contains the details. This table has only one column family : Count Details, containing one column. Each row, thus, contains only one column : VisitCount (which stores the total number of times the page was visited in the time-span given as input to the system).

#### • RETRIEVING RESULTS AS PER INPUT QUERY

The data stored in the HBase tables is first filtered as per the input query i.e. only the records falling into the input time range are dealt with further.

These filtered records were passed to the Map-Reduce phase for retrieving the output by aggregating over the relevant records.



**Fig2.3 Map-Reduce Operation**

**Mapper:** The input to the Mapper is the filtered table generated above by the system and the output generated by the mapper is a list of tuples of the form "(PageTitle, Hourly Page-

visits)” where PageTitle corresponds to the 'PageTitle+ID' column of the initial table and Hourly Page-visits corresponds to the 'VisitCount' column of the initial table.

**Shuffle/Sort:** In this phase, the input is the output of the Mapper i.e. the list of tuples generated above. This list of tuples is shuffled and sorted and the final output of this phase is a list of tuples where the first element is the PageTitle and the second element is a list of visit-counts corresponding to that page i.e. of the form “(PageTitle, Count[1] -> Count[2] -> Count[3] -> ....)”.

**Reducer :** The input to the reducer is the output of the Shuffle/Sort phase. The reducer sums over all the count-values stored as the second element of each tuple given to it as input. Thus, the output of the reducer is “(PageTitle,  $\sum \text{Count}[i]$ )”.

Finally, the output generated by the reducer is sorted in descending order of the second value of each tuple i.e. the sum of page-visits for each page.

## Input and Output Format

The input taken by the system is a time-range i.e. a starting time (Starting Date and Time) and ending time (Ending Date and Time). Based on the input to the system, a list of pages is retrieved in descending order of the number of visits made to the page in time-period given as input to the system.

# TOP-K MATCHES USING HBase

## OBJECTIVE

In this part we tried to simulate a Search Engine for Wikipedia Articles using Modern-Day Technologies. TF.IDF

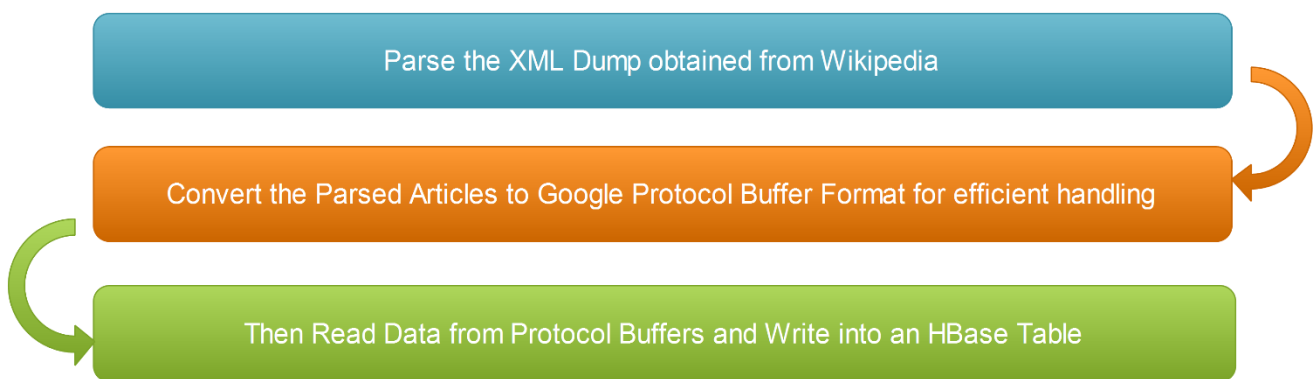
## DATASET

Wikipedia Page Articles Dump available on <https://dumps.wikimedia.org/enwiki/> is used. Each of the Article is a separate Document for this system and contains on an average around 15000 words. The number of articles are varied for analysing purposes.

## METHODOLOGY

There are 3 major steps involved:

- **PARSE THE DATA AND LOAD INTO HBase**



**Fig3.1 Parsing the Data and Loading Into HBase**

The Process is explained in **Fig3.1**. The Wikipedia Page Articles Dump extracted from Wikipedia as XML are first cleaned using WikiExtractor (<https://github.com/attardi/wikiextractor>). The cleaned XML is then parsed using Java SAX(the Simple API for XML) Parser to get the required Articles. The parsed Articles are converted to Google Protocol Buffer format using the same method mentioned in previous section. This ensures simpler and faster access to the data. Data is then read from these Protocol Buffers and inserted into the HBase Table **TFIDFDocs** with the **Title** of the Article as key and a Column Family **FileDetails** which can contains columns **URL** and **Content**. The Structure of the table is shown in **Fig3.2**.

| KEY        | FILE DETAILS |         |
|------------|--------------|---------|
| Page Title | URL          | Content |
|            |              |         |

**Fig3.2 TFIDFDocs**

### [wikidata.proto](#)

```
package wikidata;
```

```
option java_package = "com.dbms.wikidata";  
option java_outer_classname = "WikiDataProtos";
```

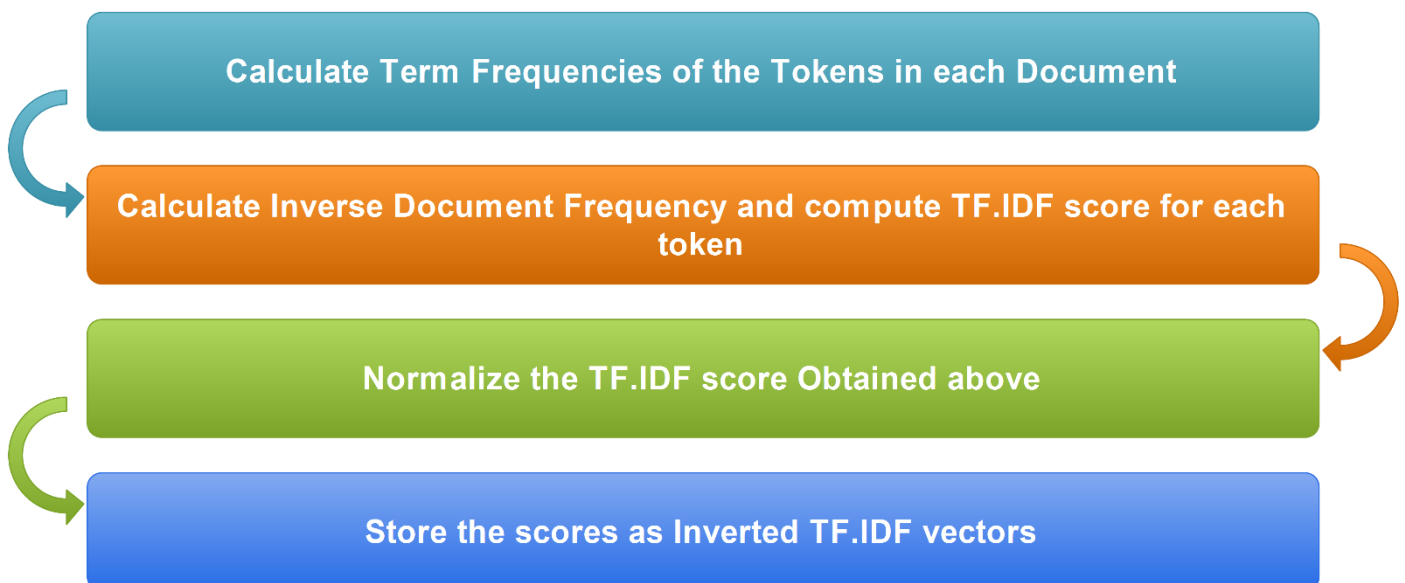
```
message DocInfo {  
    optional string title = 1;  
    optional string content = 2;  
    optional string url = 3;  
}
```

```
message RetrivedDocs {  
    repeated DocInfo docinfo = 1;  
}
```

- **COMPUTE INVERTED TF.IDF VECTORS**

After getting all the data into the table, the next step is to compute the TF.IDF

Scores for each Term-Document pair as well compute the final TF.IDF Vectors. These are compute them in an inverted fashion. This requires 4 Map-Reduce Phases as explained below.



**Fig3.3 Computing Inverted TF.IDF Vectors**

### 1. Map-Reduce 1 - Calculate Term Frequencies

First of all, the Term Frequencies, i.e. Number of times a term occurs in each document is computed for each Term-Document. The Map-Reduce operations involved are as following:

**Mapper:** The Mapper phase takes each row of the **TFIDFDocs** table which contains content of the articles as input. The Content of the Article is tokenized. Some Pre-processing like Removal of Punctuations and Removal of Stop Words is done on these tokens. To count these Word-Document pairs, each of them is given a count 1 and is passed to Shuffle/Sort with Word@Document as key and 1 as the value.

**Shuffle/Sort:** The Shuffle/Sort Phase gets all the (Word@Document,1) pairs from the Mapper. It shuffles and sorts the pairs and lists all the values corresponding to a Word@Document together. These listed together values along with the key are then passed to the Reducer.

**Reducer:** The Reducer Phase gets (Word@Document, list of 1's) as an input from Shuffle/Sort. It sums all the 1's corresponding to each Word@Document pair and thus calculates its Term Frequency(TF). The **Term Frequency Weight** is then calculated as  $1+\log(\text{TF})$  and the result is stored in the **Term-Frequency** Table. The Process is depicted in Fig3.4.

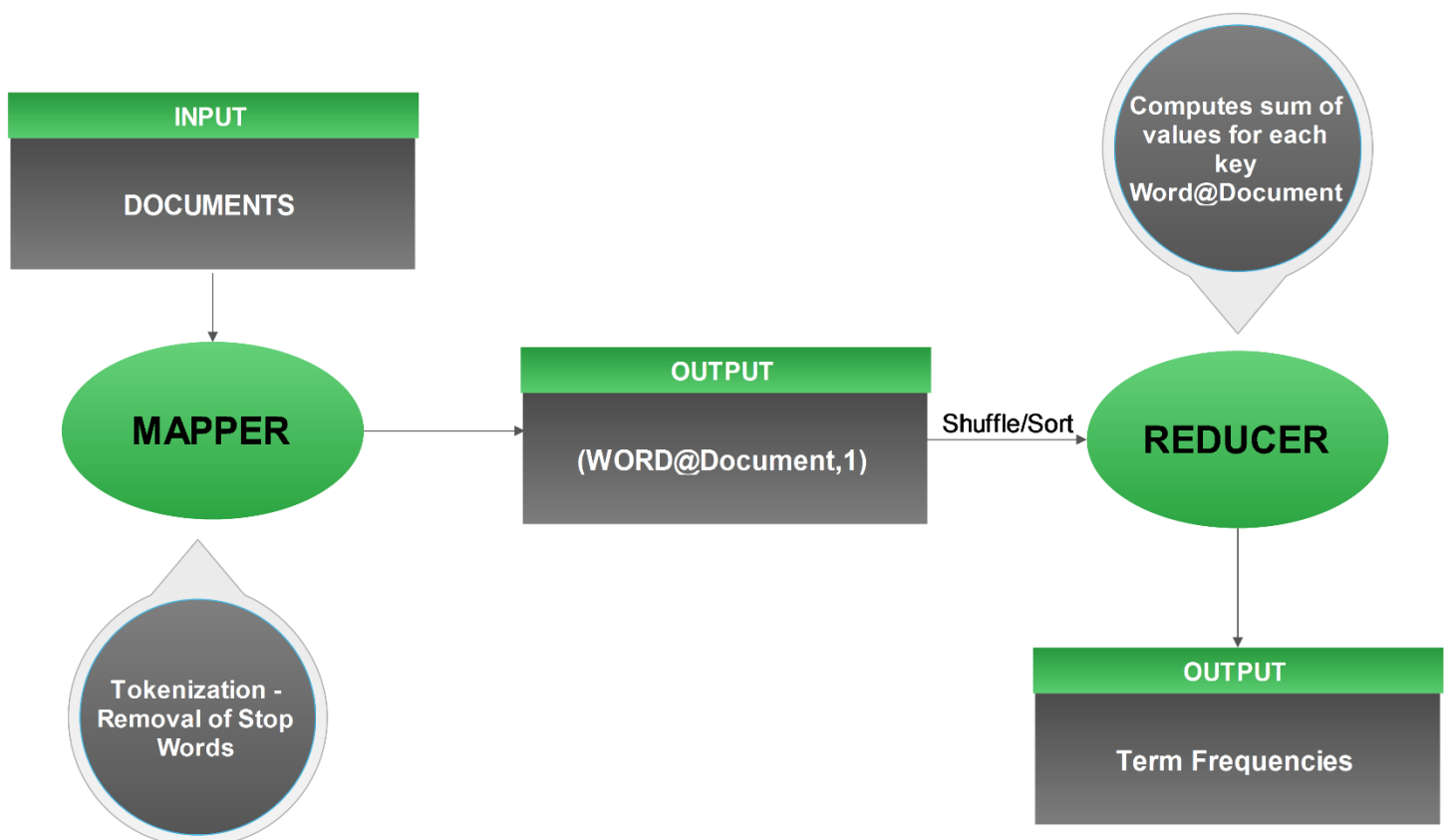


Fig3.4 MR1 - Calculate Term Frequencies

## 2. Map-Reduce 2 - Compute IDF and TF.IDF

After getting the Term Frequencies, the next task is to calculate the Inverse Document Frequency, **IDF**, i.e. the number of documents in which a term occurs. This IDF weight is

then multiplied with TF weight to get the final TF.IDF score. The Map-Reduce operations involved are as following:

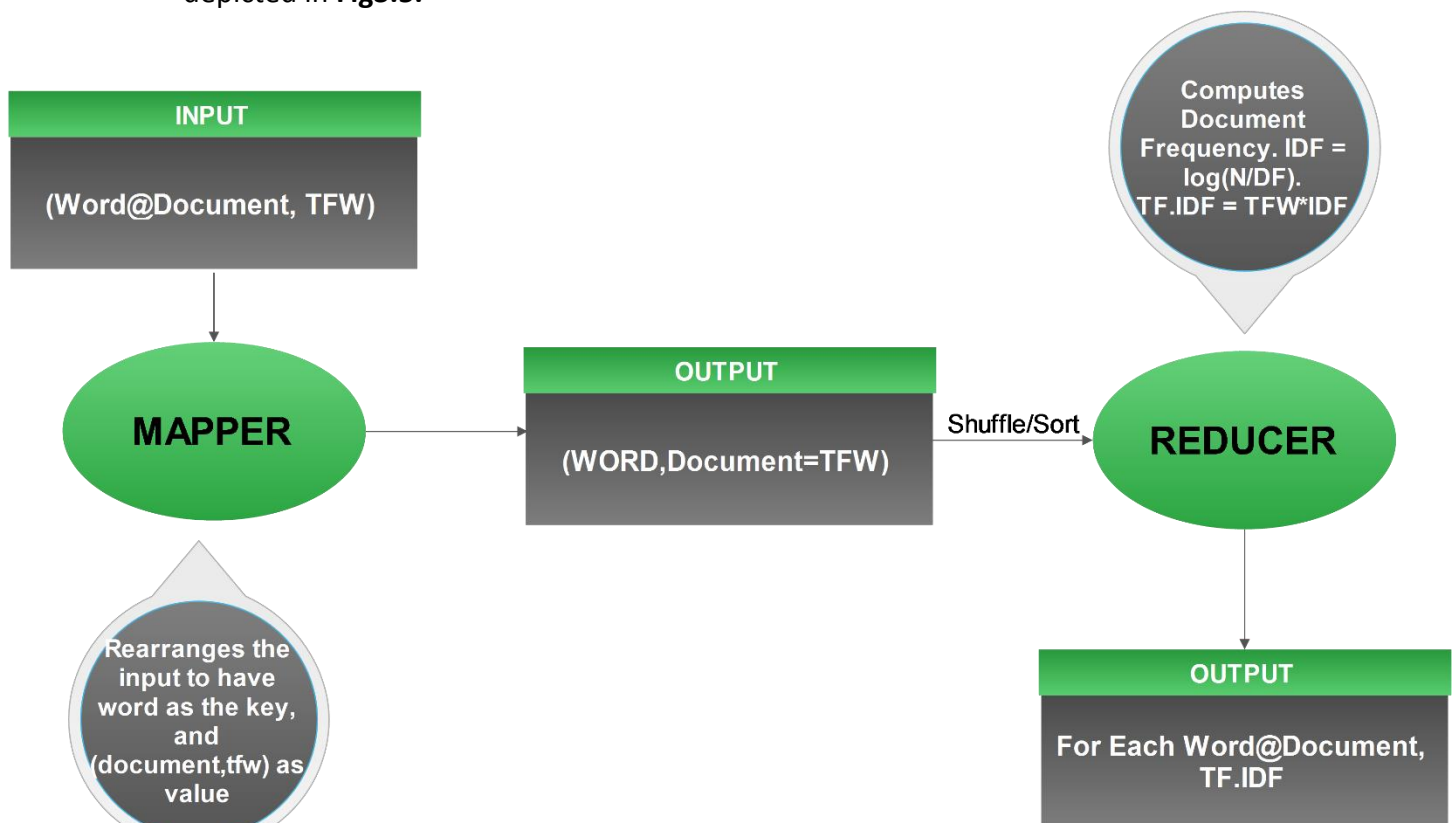
**Mapper:** The Mapper phase takes each row of the **Term-Frequency** table generated in MR1 as input which contains the Term-Frequency weight of each of the Word@Document pair. It rearranges the input to have Word as the key and Document along with Term-Frequency Weight as the value so as to count the occurrence of the word in this document. These (Word, Document=TFW) pairs are passed to Shuffle/Sort phase.

**Shuffle/Sort:** The Shuffle/Sort Phase gets all the (Word, Document=TFW) from the Mapper as input. It shuffles and sorts the pairs and lists all the Documents corresponding to a Word together. These listed together Documents and TF Weight along with the Word are then passed to the Reducer.

**Reducer:** The Reducer Phase gets (Word, list of (Document, TFW) pairs) as an input from Shuffle/Sort Phase. It Count number of Documents in the list corresponding to a word to get the Document Frequency(DF) of the Word. The **Inverse Document Frequency Weight** is  $\log(N/DF)$  where N is the Number of Documents(Articles) in the corpus. As it also has the Term-Frequency weights for the Word-Document pairs it multiplies TFW and IDFW to get the final TF.IDF score for each Word-Document pair which is:

$$w_{t,d} = (1 + \log(tf_{t,d})) \times \log(N / df_t)$$

The TF.IDF scores for each Word-Document pair is stored in the **TFIDF** table. The Process is depicted in **Fig3.5**.



**Fig3.5 MR2-Calculating IDF and TF.IDF**

### 3. Map-Reduce 3 - NORMALIZING TF.IDF SCORE

The third Map-Reduce is done in order to normalize the TF.IDF score obtained from MR2. This ensures Documents with larger lengths do not get higher TF.IDF score for their words. The Map-Reduce operations involved are as following:

**Mapper:** The Mapper phase here takes each row of the **TFIDF** table generated in MR2 as input which contains the TF.IDF score of each of the Word-Document pair. It rearranges the input to have Document as the key and Word along with TF.IDF score as the value. These (Document, Word=TF.IDF) pairs are passed to Shuffle/Sort phase.

**Shuffle/Sort:** The Shuffle/Sort Phase gets all the (Document, Word=TF.IDF) from the Mapper as input. It shuffles and sorts the pairs and lists all the Words along with TF.IDF score corresponding to a Document together. These listed together TF.IDF scores for words along with the Document are then passed to the Reducer.

**Reducer:** The Reducer Phase gets (Document, list of (Word, TF.IDF) pairs) as an input from Shuffle/Sort Phase. It computes the sum of squares of the TF.IDF scores of the words in the list corresponding to a document. Let this be  $X$ . The **Normalized TF.IDF Score** for the Word-Document Pairs is then  $\text{TF.IDF}/\sqrt{X}$ . These Normalized TF.IDF scores are stored in **TFIDFNormalized** Table. The Process is depicted in **Fig3.6**.

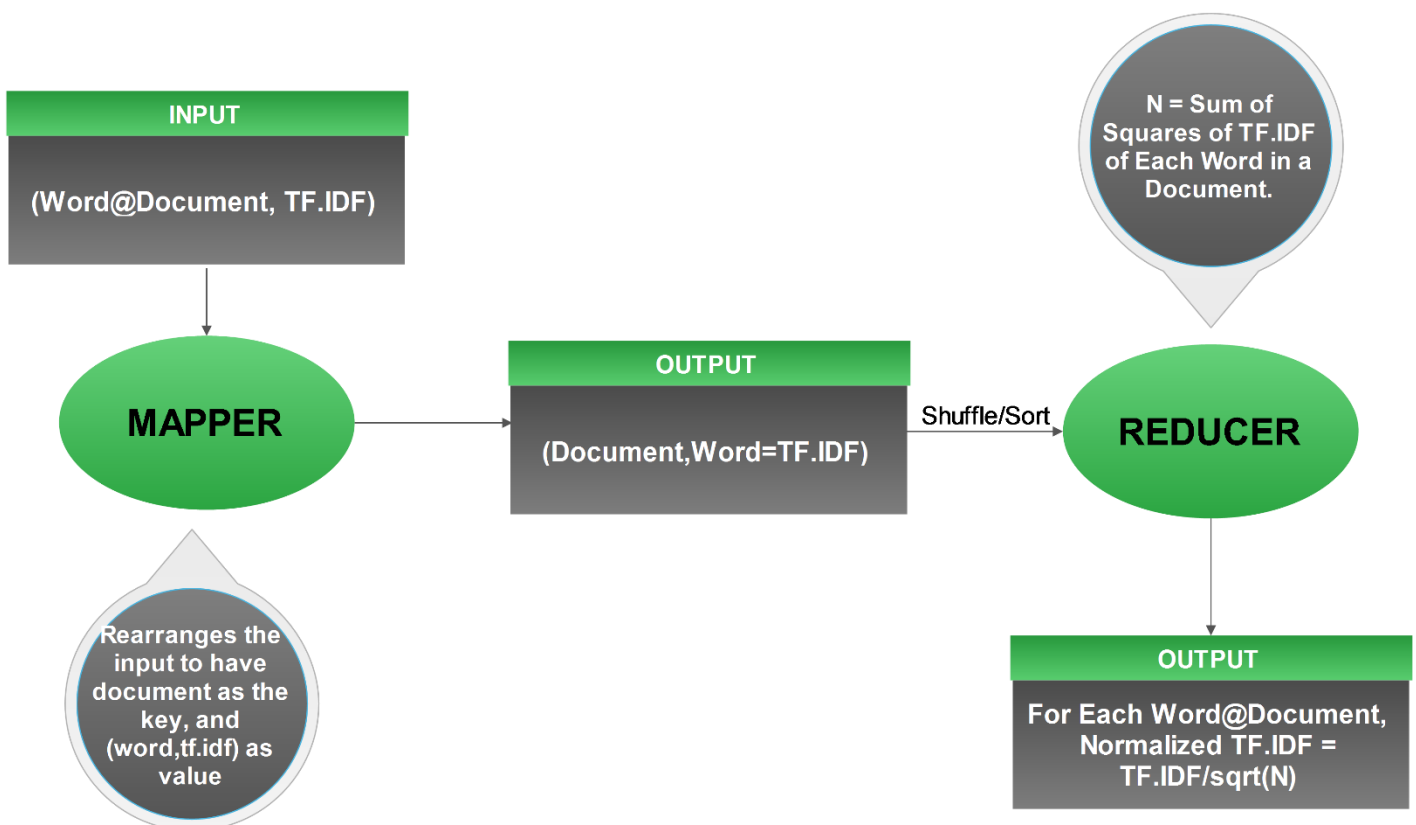


Fig3.6 MR3 - Normalizing TF.IDF

#### 4. Map-Reduce 4 - STORING THE SCORES AS INVERTED TF.IDF VECTORS

Finally, the Normalized TF.IDF scores obtained are stored as inverted vectors with a vector for each of the word in the vocabulary and the vector space being the documents in the corpus. TF.IDF score for a Word-Document pair is the weight of the weight of the Word in the direction of Document. Words are inserted as key in the final HBase Table and the vectors being the big column. The vectors are stored in an inverted fashion in order to ensure quicker retrieval of results. As the length of a search query is typically of the order of 4-10 words, keeping the words as the key is an efficient way as access by key in HBase is very fast. Thus, one need not search for the words in the vocabulary and directly get the documents in which the words of a query are present. The Map-Reduce operations involved are as following:

**Mapper:** The Mapper phase here takes each row of the **TFIDFNormalized** table generated in MR3 as input which contains the Normalized TF.IDF score of each of the Word-Document pair. It rearranges the input to have Word as the key and Document along with Normalized TF.IDF score as the value. These (Word, Document=TF.IDF) pairs are passed to Shuffle/Sort phase.

**Shuffle/Sort:** The Shuffle/Sort Phase gets all the (Word, Document=TF.IDF) from the Mapper as input. It shuffles and sorts the pairs and lists all the Documents along with normalized TF.IDF score corresponding to a Word together. These listed together Documents and TF.IDF scores along with the Word are then passed to the Reducer.

**Reducer:** The Reducer Phase gets (Word, list of (Document, TF.IDF) pairs) as an input from Shuffle/Sort Phase. It creates a row with Word as the key in **InvertedTFIDFVectors** table and inserts Documents as columns and the normalized TF.IDF scores as value in the Big Column family **DocList**. The structure of this table is shown in **Fig3.7**. The Process is depicted in **Fig3.8**.

| KEY  |        | DOC LIST |         |
|------|--------|----------|---------|
| Word | Doc1   | Doc2     | Doc i   |
|      | Score1 | Score2   | Score i |

**Fig3.7 InvertedTFIDFVectors Table Structure**



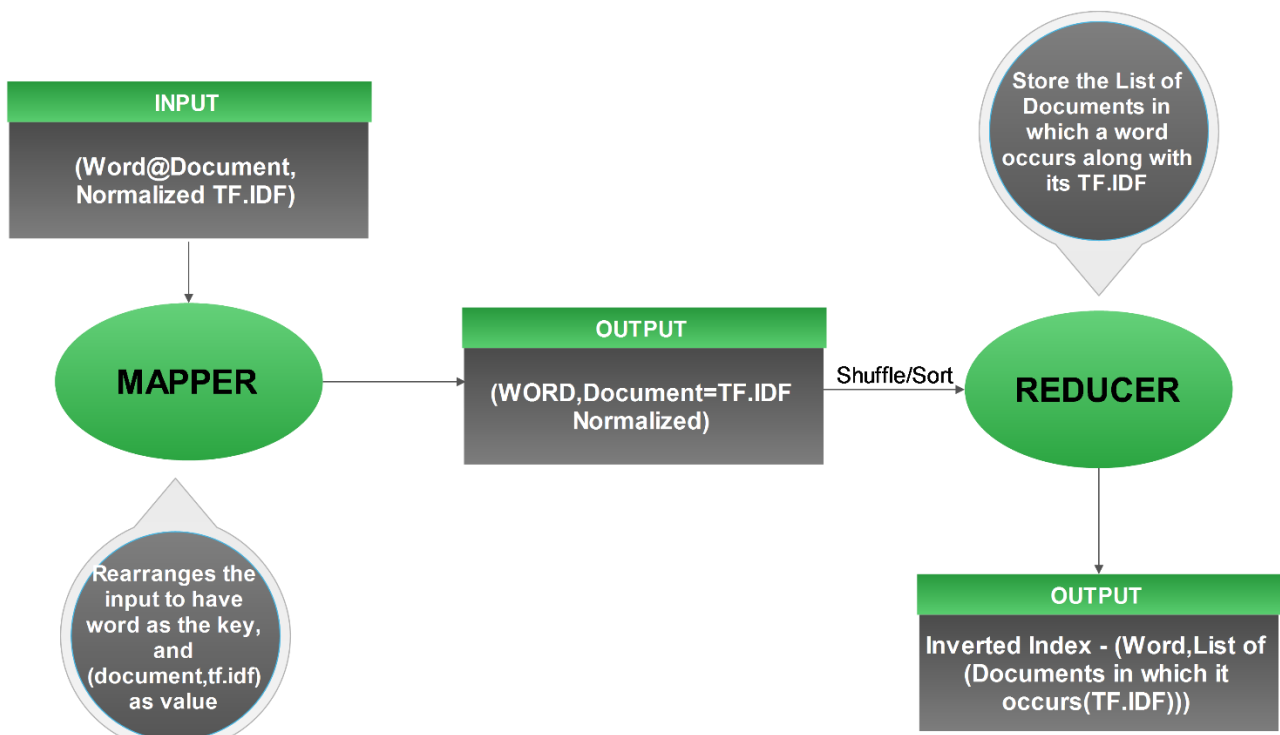


Fig3.8 MR4 - Storing Inverted TF.IDF Vectors

## • SEARCH QUERY

The goal here is to find the **cosine similarity** between a query and a document retrieve Top-k matching Documents.

When a user fires a query, the query is tokenized and the TF.IDF score for the tokens in the query are computed. The rows corresponding to the tokens of the query as key are filtered out. These rows and TF.IDF of query tokens are passed to the Mapper which operates on it as following:

**Mapper:** The Mapper phase for search takes rows of the **InvertedTFIDFVectors** table with key as query tokens along TF.IDF score of the token in the query (qt) as the input. It multiplies qt with each of the TF.IDF scores of the Docs (dt) present in the DocList of the token. The pairs (Document, dt\*qt) are passed to Shuffle/Sort phase.

**Shuffle/Sort:** The Shuffle/Sort Phase gets all the (Document, dt\*qt) from the Mapper as input. It shuffles and sorts the pairs and lists all the dt\*qt values corresponding to a Document together. These listed together dt\*qt values scores along with the Document are then passed to the Reducer.

**Reducer:** The Reducer Phase gets (Document, list of dt\*qt values) as an input from Shuffle/Sort Phase. It sums up all the values corresponding to a document which gives the **Cosine Similarity** of the query with that particular Document.

The **Cosine Similarities** are then sorted and the Top-K matching Documents are displayed to the user. The Query Process is depicted in **Fig3.9**.

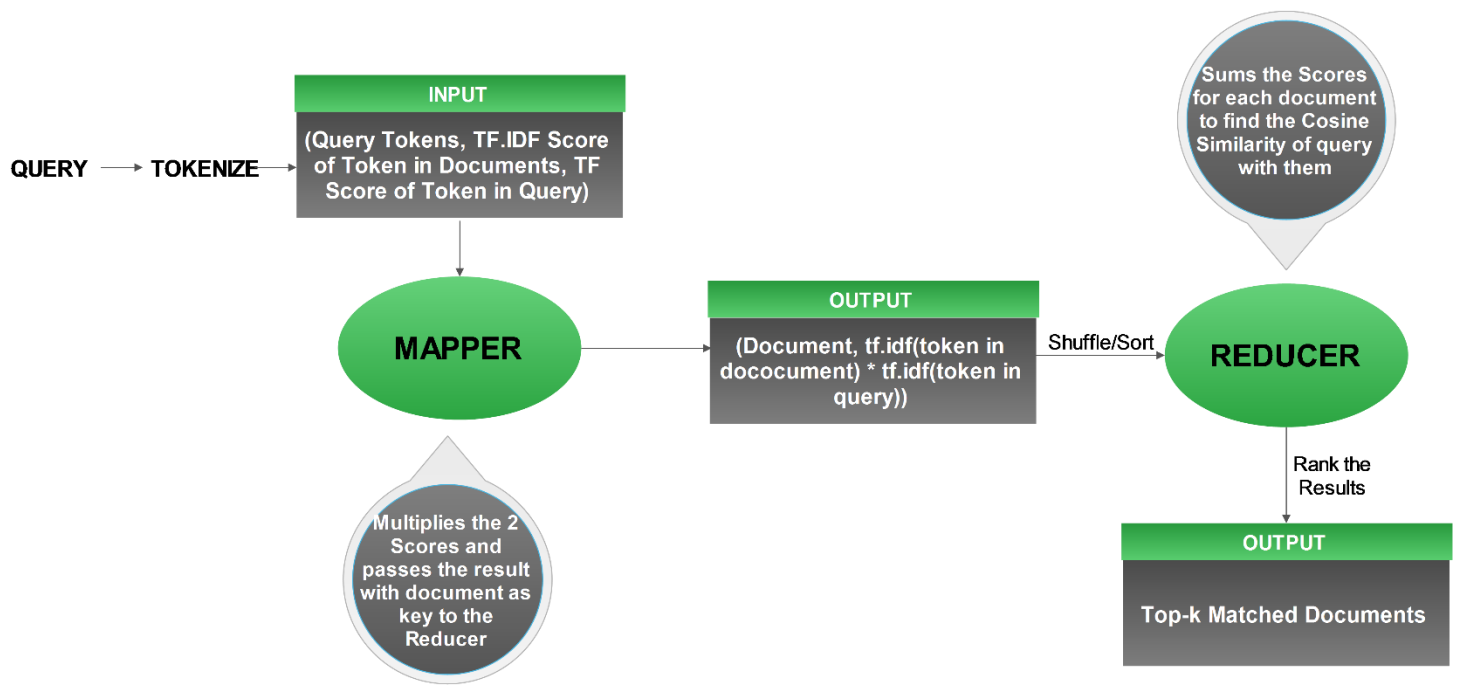


Fig3.9 Searching Query

## Analysis

For evaluation of the performance of the system, a constant set of versatile queries was fed into the system and the average of all the times taken for the computation of each query was taken as the metric for analysis. For evaluation, two parameters were varied :

- The size of data which was indexed initially by the system and
- The number of data-nodes in the Hadoop cluster.

The graph was obtained as follows:

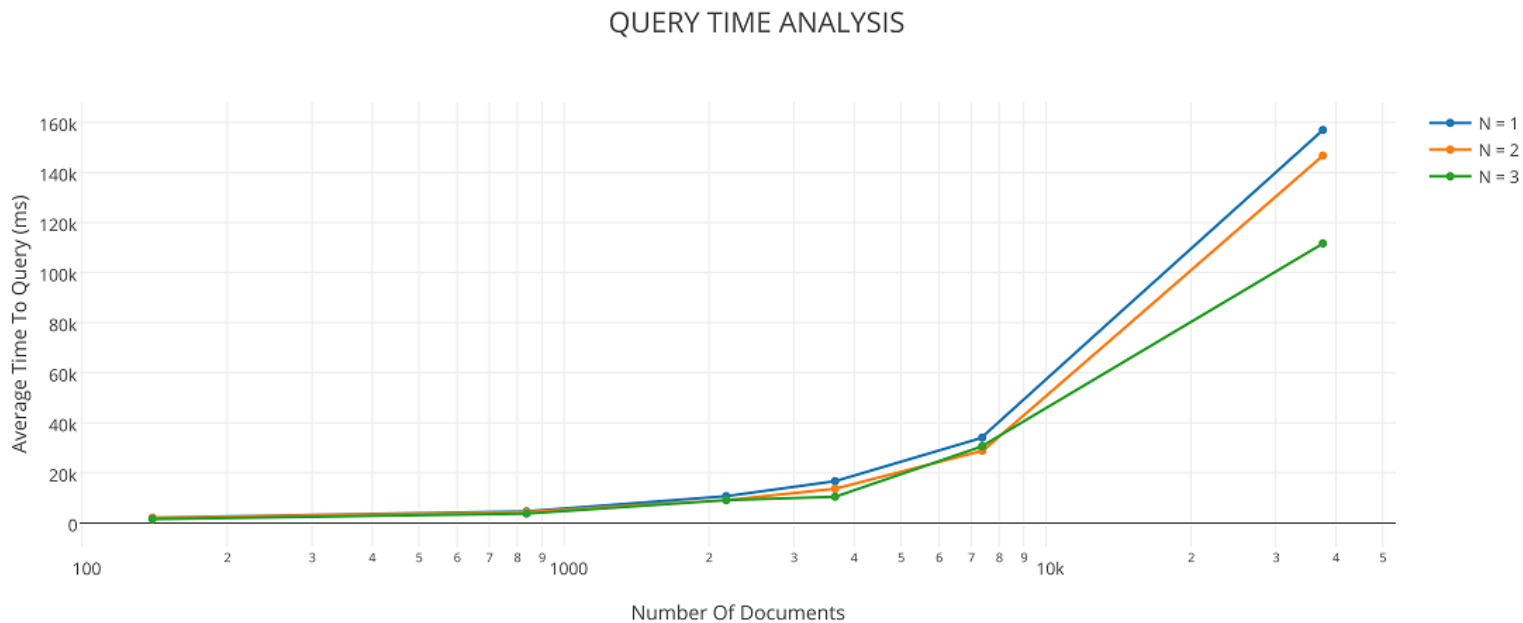


Fig3.10 Query Time Analysis

### Varying the data-size indexed by the system

In this part of the analysis, the data which was initially indexed by the system was varied by changing the number of Wikipedia pages being fed into the system. Results were obtained for the following values of the number of Wikipedia documents being indexed : 140, 2171, 3653, 7375 and 37558.

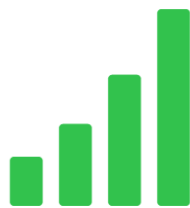
As expected, the average time taken to query increased with an increase in the size of the data being indexed. This behavior was verified individually for Hadoop clusters of size 1,2 and 3. Also, it was observed that the rate of increase of the average time was more for larger values of a data-size than the smaller data-size values. This can be attributed to the fact that initially, when the data-size is small, the time taken for querying increases marginally for an increase in the data-size (e.g. it hardly matters whether the system indexes 5 documents or 10 documents). However, when the data-size becomes adequately large, the variation in the time taken to query increases manifold for the same change in the data-size.

## Varying the size of Hadoop cluster :

In this part of the analysis, the number of data-nodes in the Hadoop cluster were varied. Results were obtained for 1, 2 and 3 node clusters. It was observed that for a given size of data being indexed, the time taken to query is almost same in casee then the data-size is small. But, when the size becomes adequately large ( $\geq 7000$  Wikipedia pages in our case), the time taken by a larger cluster is lesser than that taken by smaller clusters. Also, the gap in the average time taken to query by clusters of different sizes keeps on increasing with an increase in the data-size.

# WIKIMANIA - THE FINAL APPLICATION

This is the utility we developed as part of this project. Here are some screen shots of the same:



WIKI POPULAR PAGES



WIKI SEARCH

## POPULAR PAGES

FROM: 7/4/16 1:16:56 AM

TO: 18/4/16 1:17:13 AM



| Page Title   | Page Hits |
|--|-----------|
| Wikisource:%E9%A6%96%E9%A1%B5                                    | 140       |
| Ikhasi_Elikhulu  | 120       |
| Special:MyTalk   | 41        |
| %E9%A6%96%E9%A1%B5   | 35        |
| User:46.151.55.4   | 18        |
| User:46.151.55.21  | 15        |
| User:Gregorydavid  | 15        |
| IsiNgisi   | 14        |
| User:46.151.55.2   | 12        |
| Wikisource:%E5%8F%B2%E6%9B%B8                                    | 11        |
| User:46.151.55.15  | 9         |
| User:46.151.55.5   | 9         |
| Florida  | 8         |
| IKrowati   | 8         |
| Ikhasi_elikhulu  | 8         |
| Talk:Iwolintshi  | 8         |
| Wikipedia  | 8         |
| Wikisource:%E5%85%B8%E7%B1%8D                                    | 8         |
| I-Eshiya   | 7         |
| IGibhithe  | 7         |
| Iqhwa  | 7         |
| Talk:Bongo_Maffin  | 7         |
| Wikivoyage:%E4%BA%92%E5%8A%A9%E5%AE%A2%E6%A0%88                  | 7         |
| Special:%E6%9C%80%E8%BF%91%E6%9B%B4%E6%94%B9                     | 6         |
| User:46.151.55.13  | 6         |
| User:46.151.55.14  | 6         |
| User:46.151.55.19  | 6         |
| User:46.151.55.3   | 6         |
| User:46.151.55.31  | 6         |
| User:46.151.55.7   | 6         |
| Wikivoyage:%E5%B8%AE%E5%8A%A9%E7%BB%B4%E5%9F%BA%E5%AF%BC%E6%B... | 6         |
| 2012   | 5         |



# WIKIMANIA SEARCH



## ANSI C

**Content:** ansi c ansi c also known as c89 and c90 depending on the year of ratification refers to the family of successive standards published by the american national standards institute ansi for the c programming language software developers writing in c are encouraged to conform to the standards as doing so aids portability between compilers the first standard for c was published by ansi although this document was subsequently adopted by international organization for standardization iso an.....

URL: <https://en.wikipedia.org/wiki?curid=3172>

Score: 0.0696841756637332

## APL (programming language)

**Content:** apl programming language apl named after the book a programming language is a programming language developed in the 1960s by kenneth e iverson its central datatype is the multidimensional array it uses a large range of special graphic symbols to represent most operators giving very concise code it has been an important influence on the development of concept modeling spreadsheets functional programming and computer math packages it has also inspired several other programming lan.....

URL: <https://en.wikipedia.org/wiki?curid=1451>

Score: 0.04768598014125666

## AWK

**Content:** awk awk is an interpreted programming language designed for text processing and typically used as a data extraction and reporting tool it is a standard feature of most unix like operating systems awk was created at bell labs in the 1970s and its name is derived from the family names of its authors alfred aho peter weinberger and brian kernighan the acronym is pronounced the same as the name of the bird auk which acts as an emblem of the language such as on the awk programming language.....

URL: <https://en.wikipedia.org/wiki?curid=1456>

Score: 0.045602738131081066

## AMOS (programming language)

**Content:** amos programming language amos basic is a dialect of the basic programming language implemented on the amiga computer amos basic was published by euromicro software and originally written by fran ois lionet with constantin sotiropoulos amos is a descendant of stos basic for the atari st amos basic was first produced in 1990 amos competed on the amiga platform with acid software s blitz basic both basics differed from other dialects on different platforms in that they allowed the easy c.....

URL: <https://en.wikipedia.org/wiki?curid=2957>

Score: 0.0425456637332