Implementation of MapReduce Algorithm and Nutch Distributed File System in Nutch

Kowsalya N
Lecturer in Computer Science

Dr. C. Chandrasekar
M.C.A., Ph.D.
Department of Computer Science, Associate Professor,Periyar University, Salem.

ABSTRACT
This paper provides an in-depth description of MapReduce algorithm and Nutch Distributed File System in Nutch web search engine. Nutch is an open-source Web search engine that can be used at global, local, and even personal scale. To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and web proliferation, creating a web search engine today is very different from ten years ago.

1. INTRODUCTION
The web creates new challenges for information retrieval. The amount of information on the web is growing rapidly, as well as the number of new users inexperienced in the art of web research. People are likely to surf the web using search engines. Automated search engines that rely on keyword matching usually return too many low quality matches. MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. It was developed at Google for indexing Web pages and replaced their original indexing algorithms and heuristics in 2004. The Distributed Nutch File System, a set of software for storing very large stream-oriented files over a set of commodity computers.

2. NEED FOR LARGE DATA PROCESSING
We live in the data age. It's not easy to measure the total volume of data stored electronically. The problem is that while the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives have not kept up. One typical drive from 1990 could store 1370 MB of data and had a transfer speed of 4.4 MB/s,§ so we could read all the data from a full drive in around five minutes. Almost 20 years later one terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk. This is a long time to read all data on a single drive—and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in less than two minutes. This shows the significance of distributed computing.

3. CHALLENGES IN DISTRIBUTED COMPUTING – MEETING NUTCH
Various challenges are faced while developing a distributed application. The first problem to solve is hardware failure: as soon as we start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available. This is how RAID works, for instance, although Nutchfilesystem, the Nutch Distributed Filesystem(NDFS), takes a slightly different approach.

The second problem is that most analysis tasks need to be able to combine the data in some way; data read from one disk may need to be combined with the data from any of the other 99 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes transforming it into a computation over sets of keys and values.

This, in a nutshell, is what Nutch provides: a reliable shared storage and analysis system. The storage is provided by NDFS, and analysis by MapReduce. There are other parts to Nutch, but these capabilities are its kernel.

Nutch is the popular open source implementation of MapReduce, a powerful tool designed for deep analysis and transformation of very large data sets. Nutch enables you to explore complex data, using custom analyses tailored to your information and questions. Nutch has its own filesystem which replicates data to multiple nodes to ensure if one node holding data goes down, there are at least 2 other nodes from which to retrieve that piece of information. This protects the data availability from node failure, something which is critical when there are many nodes in a cluster (aka RAID at a server level).

4. ORIGIN OF NUTCH
Nutch was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Nutch an open source web search engine, itself a part of the Lucene project. Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn't scale to the billions of pages on the Web. Help was at hand with the publication of a

paper in 2003 that described the architecture of Google's distributed filesystem, called GFS, which was being used in production at Google.# GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In particular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS). In 2004, Google published the paper that introduced MapReduce to the world. Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS. NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search.

5 . IMPLEMENTATION OF MAPREDUCE

The framework is divided into two parts:
Map, a function that parcels out work to different nodes in the distributed cluster.

Reduce, another function that collates the work and resolves the results into a single value.

MapReduce is a programming model and an associated implementation for processing and generating largedata sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

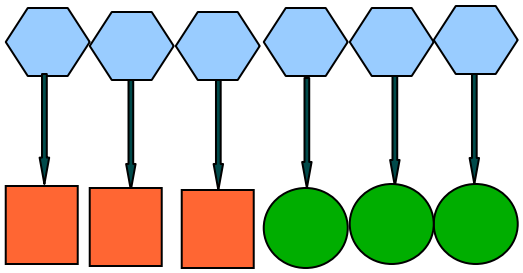
These abstractions are inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical .record. In our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user specialized map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

5.1 Programming Model

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

5.1.1 Map

map (in_key, in_value) -> (out_key, intermediate_value) list



Example: Upper-case Mapper

let map(k, v) = emit(k.toUpper(), v.toUpper())

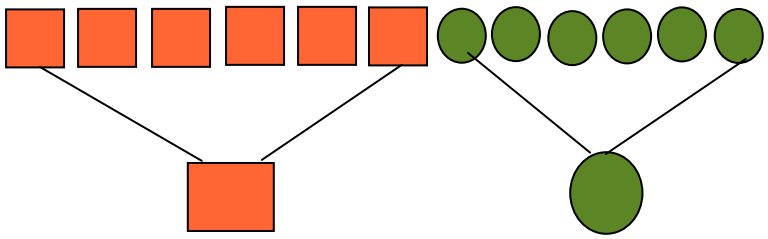
("foo", "bar") --> ("FOO", "BAR")

("Foo", "other") -->("FOO", "OTHER")

("key2", "data") --> ("KEY2", "DATA")

5.1.2 Reduce

reduce (out_key, intermediate_value list) ->out_value list



Example: Sum Reducer

let reduce(k, vals)

sum = 0

for each int v in vals:

sum += v

emit(k, sum)

("A", [42, 100, 312]) --> ("A", 454)

("B", [12, 6, -2]) --> ("B", 16)

Example2:-

Counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

map(String key, String value):

// key: document name

// value: document contents

for each word w in value:

EmitIntermediate(w, "1");

reduce(String key, Iterator values):

// key: a word

// values: a list of counts

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a mapreduce specification object with the names of the input and output files, and optional tuning parameters. The user then invokes the MapReduce function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++)

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures,

and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues. As a reaction to this complexity, Google designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

5.2 Types

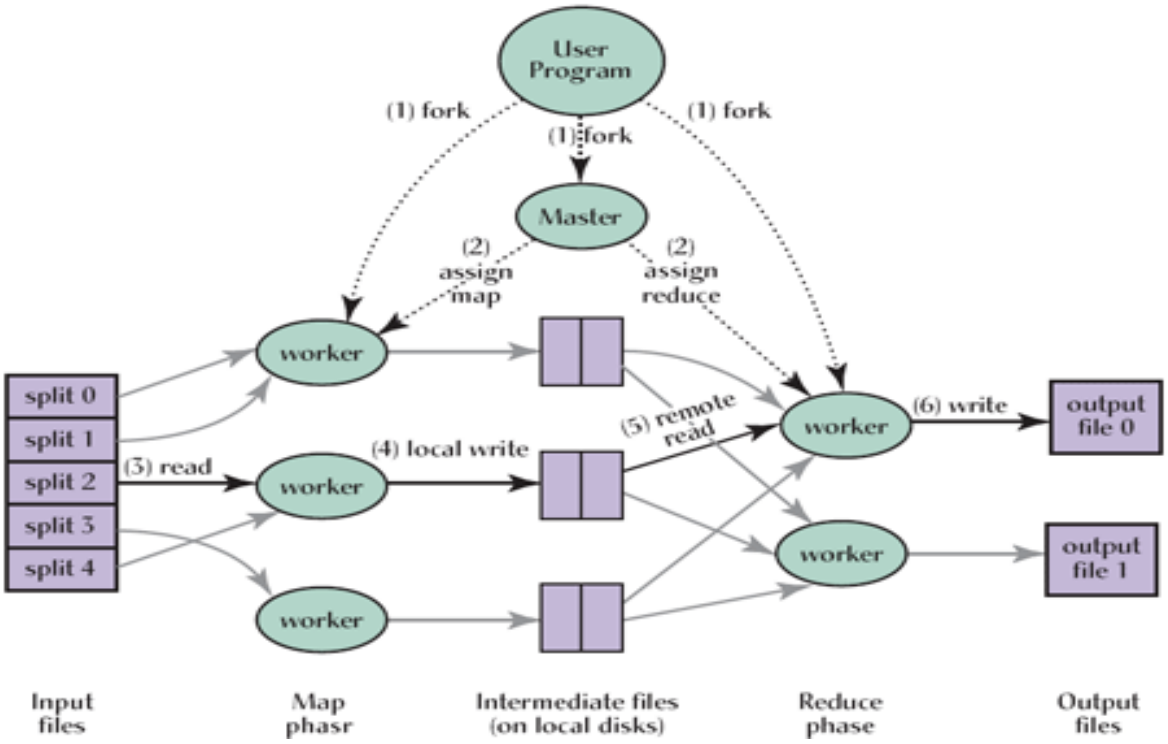
Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated

types:

map (k1,v1) ! list(k2,v2)

reduce (k2,list(v2)) ! list(v2)

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values. Our C++ implementation passes strings to and from the user-de_ned functions and leaves it to the user code to convert between strings and appropriate types.



Inverted Index: The map function parses each document, and emits a sequence of hword; document IDi pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a hword; list(document ID)i pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a hkey; recordi pair. The reduce function emits all pairs unchanged.

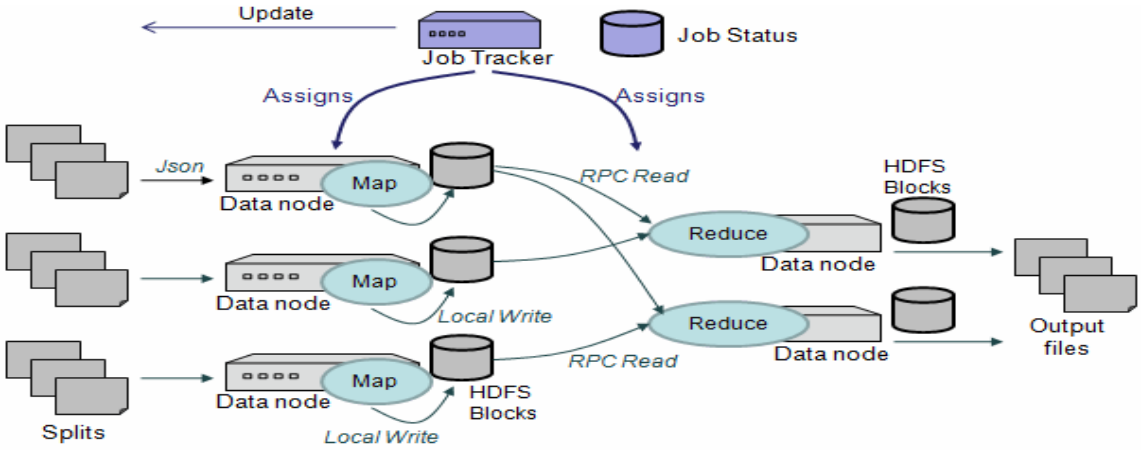
5.3 NutchMapReduce

Nutch Map-Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A Map-Reduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the Map-Reduce framework and the Distributed FileSystem are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Nutch runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a tasks fails, the jobtracker can reschedule it on a different tasktracker. Nutch divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Nutch creates one map task for each split, which runs the userdefined map function for each record in the split.



Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained. On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of a NDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created. Nutch does its best to run the map task on a node where the input data resides in NDFS. This is called the data locality optimization. It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any NDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data. Map tasks write their output to local disk, not to NDFS. Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in NDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Nutch will automatically rerun the map task on another node to recreate the map output. Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in NDFS for

reliability. For each NDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal NDFS write pipeline consume. The dotted boxes in the figure below indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between

nodes. The number of reduce tasks is not governed by the size of the input, but is specified independently.

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for every key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time. Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle since the processing can be carried out entirely in parallel.

5.4 Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Nutch allows the user to specify a combiner function to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Nutch does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

5.5 Nutch Streaming

Nutch provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. Nutch Streaming uses Unix standard streams as the interface between Nutch and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program. Streaming is naturally suited for text processing (although as of version 0.21.0 it can handle binary streams, too), and when used in text mode, it has a line-oriented view of data. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

6. NUTCH DISTRIBUTED FILE SYSTEM(NDFS)

Filesystems that manage the storage across a network of machines are called distributed filesystems. Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss. Nutch comes with a distributed filesystem called NDFS, which stands for Nutch Distributed Filesystem.

NDFS, the Nutch Distributed File System, is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information. Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications.

6.1 NDFS Filesystem Semantics

Files can only be written once. After the first write, they become read-only. (Although they can be deleted.)

Files are stream-oriented; you can only append bytes, and you can only read/seek forward.

There are no user permissions or quotas, although these could be added fairly easily.

So, all access to the NDFS system is through approved client code. There is no plan to create an OS-level library to allow any program to access the system. Nutch is the model user, although there are probably many applications that could take advantage of NDFS. (Pretty much anything that has very large stream-oriented files would find it useful, such as data mining, text mining, or media-oriented applications.)

6.2 System Design

There are two types of machines in the NDFS system:

1) Namenodes, which manage the file namespace

2) Datanodes, which actually store blocks of data

The NDFS namespace has a single Namenode which defines it. There can be an arbitrary number of Datanodes, all of which are configured to communicate with the single Namenode. Namenodes are responsible for storing the entire namespace and filesystem layout. This basically consists of table of the following tuples:

filename_0 -->BlockID_A, BlockID_B, ...BlockID_X, etc.
filename_1 -->BlockID_AA, BlockID_BB, ... BlockID_XX, etc. etc.

A filename is a string, and the BlockIDs are just unique identifiers. Each filename can have an arbitrary number of blocks associated with it, growing with the file length. This is the only Namenode data structure that needs to be written to disk. All others are reconstructed at runtime. The Namenode is a critical failure point, but it shouldn't be an issue for load-management. It needs to do very little actual work, mainly serving to guide the large team of Datanodes. Datanodes are responsible for actually storing data. A datastore consists of a table of the following tuples:

BlockID_X --> [array of bytes, no longer than BLOCK_SIZE] BlockID_Y --> [array of bytes, no longer than BLOCK_SIZE] etc.

This is the only structure that the Datanode needs to keep on disk. It can reconstruct everything else at runtime. A given block can, and should, have copies stored on multiple Datanodes. A given Datanode has at most one copy of a given Block, and will often have no copies. It should be clear how a single Namenode table and a set of partly- overlapping Datanode tables are enough to reconstruct an entire filesystem. Upon startup, all Datanodes contact the central Namenode. They upload to the Namenode the blocks they have on the local disk. The Namenode thus builds a picture of where to find each copy of every block in the system. This picture will always be a little bit out of date, as Datanodes might become unavailable at any time.

Datanodes also send periodic heartbeat messages to the Namenode. If these messages disappear, the Namenode knows the Datanode has become unavailable. The system can now field client requests. Imagine that a client wants to read file "foo.txt". It first contacts the Namenode over the network; the Namenode responds with two arrays:

The list of Blocks that make up the file "foo.txt"

The set of Datanodes where each Block can be found

The client examines the first Block in the list, and sees that it is available on a single Datanode. Fine. The client contacts that Datanode, and provides the BlockID. The datanode transmits the entire block. The client has now successfully read the first BLOCK_SIZE bytes of the file "foo.txt". (We imagine BLOCK_SIZE will be around 32MB.) So it is now ready to read the second Block. It finds that the Namenode claims two Datanodes hold this Block. The client picks one at random and contacts it.

Imagine that right before the client contacts that Datanode, the Datanode's network card dies. The client can't get through, so it contacts the second Datanode provided by the Namenode. This time, the network connection works just fine. The client provides the ID for the second Block, and receives up to BLOCK_SIZE bytes in response. This process can be repeated until the client reads all blocks in the file.

7. SCALABILITY

Nutch hasn't scaled beyond 100 million pages so far, for both economic and technical reasons. Maintaining an up-to-date copy of the entire Web is inherently an expensive proposition, costing substantial amounts of bandwidth. (Perhaps 10 terabytes per month at minimum, which is about 30 megabits per second, which costs thousands of dollars per month)

Answering queries from the general public is also inherently an expensive proposition, requiring large amounts of computer equipment to house terabytes of RAM and large amounts of electricity to power it, as well as one to three orders of magnitude more bandwidth.

Nutch's link analysis stage requires the entire link database to be available on each machine participating in the calculation and includes a significant non-parallel final collation step.

As Nutch partitions its posting lists, across cluster nodes by document, each query must be propagated to all of the hundreds or thousands of machines serving a whole-web index. This means that hardware failures will happen every few hours, the likelihood of a single slow response causing a query to wait several seconds for a retransmission is high, and the CPU resources required to process any individual query become significant.

As Nutch crawls retrieve unpredictable amounts of data, load-balancing crawls with limited disk resources are difficult.

Much of Nutch's distribution across clusters must be done manually or by home-grown cluster management machinery; in particular, the distribution of data files for crawling and link analysis, and the maintenance of search-servers.txt files all must be done by hand. Large deployments will require fault-tolerant automation of these functions.

Further work is being done in this area to enhance Nutch's scalability. The Nutch Distributed File System (NDFS) is in current development versions of Nutch, to enhance performance along the lines proposed by the Google File System. NDFS has been used recently to run a link analysis stage over 35 million pages on twelve machines.

8. PLUGIN BASED ARCHITECTURE:

The open source nature of Nutch makes it ideal for modification. This potential for customization is further aided by the modular nature of Nutch. The Nutch search engine is built upon a basic code backbone which is augmented heavily through the use of plugins. Plugins are program extensions which are be added to a host application. The release version of Nutch contains dozens of plugins which may be added or removed as desired by changing the Nutch configuration. These plugins are responsible for the parsing of different file types during the crawl, indexing of crawl results, protocols through which the crawl can operate, and querying of indexed crawl results, among other tasks. Essentially, the majority of the primary search engine functions are performed by plugins. Therefore, modifying the search engine may be accomplished by changing the configuration of the plugins, which may include adding new plugins. In order to add WordNet-related functionality to Nutch, a plugin should be created.

9. CONCLUSION:

The MapReduce programming model has been successfully used at Google for many different purposes. First, MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. MapReduce provides a programming model that abstracts the problem from disk reads and writes transforming it into a computation over sets of keys and values. Nutch is the popular open source implementation of MapReduce, a powerful tool designed for deep analysis and transformation of very large data sets.

Second, Filesystems that manage the storage across a network of machines are called distributed filesystems. NDFS, the Nutch Distributed File System, is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to these information.

Finally, Nutch provides an API to MapReduce that allows to map and reduce functions in languages other than Java. The Nutch open source search engine is built upon a basic code backbone which is augmented heavily through the use of plugins. Plugins are program extensions which are be added to a host application. These plugins are responsible for the parsing of different file types during the crawl, indexing of crawl results, protocols through which the crawl can operate, and querying of indexed crawl results, among other tasks.

10. REFERENCES:

[1] White, Tom. 2006. “Introduction to Nutch, Part 1: Crawling”. Retrieved from http://today.java.net/pub/a/today/2006/01/10/introduction-to-nutch-1.html.

[2] Smart, John Ferguson. 2006. “Integrate Advanced Search Functionalities Into Your Apps”. Retrieved from http://www.javaworld.com/javaworld/jw-09-2006/jw-0925-lucene.html .

[3] Open source at: http://en.wikipedia.org/wiki/Open_Source

[4] Apache nutch at: http://lucene.apache.org/nutch/

[5] Nutch at: http://www.nutch.org/

[6] M. Cafarella and D. Cutting. Building Nutch: open source search. 2004.

[7] MapReduce and Simplified Data Processing on large Clusters, journal of Jeffrey Dean and Sanjay Ghemawat, Google, Inc.