An open Source Configuration for a Large - Scale Web Crawler with Clustering

Prof. R. Nedunchelian*

© Institute of Management Studies, Noida
Online access at www.publishingindia.com

Abstract

This paper aims at implementing a fully functional Free and Open Source Software (FOSS) based search engine. Implementation will involve the use of information retrieval procedures such as crawling, indexing and searching. In addition the results to the search queries are clustered so that similar results are grouped together into clusters, thereby easing the job of the end-user by not being forced to go through lists of results. We use an open source search engine called Nutch which performs the aforementioned crawling and indexing procedures and displays results to search queries. An open source results clustering engine called Carrot2 is used to cluster the results to the user queries and display them in groups of related results called as clusters. Such a search engine is a fully customizable and cost-effective solution for non-profit organizations and small businesses and is tailored to suit individual needs.

1. Introduction

1.1 Background

In recent years, the World Wide Web has become the primary medium for personal, social and commercial information dissemination. Finding useful information from the web, which is a large scale distributed structure, require efficient search strategies. In order to search information on the web we use what are referred to as search engines. *How does a search engine work?*

Most search engines maintain an index of web pages which is exploited to provide up-to-date information related to user queries. Highly distributed and dynamic nature of the source of web content is a major source of practical problems for search engines to maintain up-to-date index of the web content as they have to crawl the web periodically. Web search engines work by storing information about many web pages, which they retrieve from the WWW itself. These pages are retrieved by a Web crawler (sometimes also known as a spider) — an automated Web browser which follows every link it sees. The contents of each page are then analyzed to determine how it should be indexed (for example, words are extracted from the titles, headings, or special fields called meta tags). Data about web pages are stored in an index database for use in later queries. Some search engines, such as Google [16], store all or

**Professor and Head, Department of computer science and engineering, Sri Venkateshwara College of engineering, India*

part of the source page (referred to as a cache) as well as information about the web pages, whereas others, such as AltaVista[17], store every word of every page they find. This cached page always holds the actual search text since it is the one that was actually indexed, so it can be very useful when the content of the current page has been updated and the search terms are no longer in it. This problem might be considered to be a mild form of linkrot, and Google's handling of it increases usability by satisfying user expectations that the search terms will be on the returned web page. This satisfies the principle of least astonishment since the user normally expects the search terms to be on the returned pages. Increased search relevance makes these cached pages very useful, even beyond the fact that they may contain data that may no longer be available elsewhere. When a user enters a query into a search engine (typically by using key words), the engine examines its index and provides a listing of best-matching web pages according to its criteria, usually with a short summary containing the document's title and sometimes parts of the text. Most search engines support the use of the boolean operators AND, OR and NOT to further specify the search query. Some search engines provide an advanced feature called proximity search which allows users to define the distance between keywords.

1.2 Effectiveness of available search engines

The usefulness of a search engine depends on the relevance of the result set it gives back. While there may be millions of web pages that include a particular word or phrase, some pages may be more relevant, popular, or authoritative than others. Most search engines employ methods to rank the results to provide the "best" results first. How a search engine decides which pages are the best matches, and what order the results should be shown in, varies widely from one engine to another. The methods also change over time as Internet usage changes and new techniques evolve. From a quantitative viewpoint, the web is currently witnessing a veritable explosion. The Indexed Web contained at least 24.51 billion pages as on September, 2007[5]. In order to locate information on the Web, we use Web Search Engines. The usefulness of a search engine depends on the relevance of the result set it gives back. While there may be millions of web pages that include a particular word or phrase, some pages may be more relevant, popular, or authoritative than others. For several reasons such as the sheer size of the Web and ownership of the engine, even the generic search engines are known to have been ineffective in generating requisite search-results for domain specific queries. Crawling of the whole web takes order of days/weeks, large network bandwidth and storage capacity. The delay in updation of index causes display of irrelevant pages and/or missing highly relevant pages to the user queries. The web is growing at a faster rate compared to processor speed, network bandwidth and storage capacity which requires efficient scalable solutions.

2. Existing Approaches

2.1 Open source search engine

Most of the industry-standard search engine companies are not willing to divulge the internal workings of their search engines due to obvious competitive reasons. Although many search engines provide free local within-site search capabilities for non-profit and small businesses, the users are bound by the search provider's terms and can do little in terms of customizing the search engines to their requirements. In 2004, Nutch[1] was developed as a full-fledged open source search engine that runs on Lucene Java[12], an open source search engine library. One of the major goals of the Nutch search engine was to increase transparency in web search engines, which is practically impossible due to the secret nature of the algorithms used by most search engines. In addition, it was built to be fully scalable to the entire web without compromising on state-of-the-art search quality.

2.2 Open source clustering engine

Clustering of web search results involves accumulating results that share similar traits into groups, labeled as clusters, thereby making it easy for the user to identify related results. However, many search engines go not allow the user to view results to search queries in terms of clusters. The Lingo[3] algorithm used by Carrot2, an open source clustering engine[11] enables online clustering of search results similar to preprocessing the display of results.

This clustering technique can read search results from most of today's search engines and perform cluster analysis over those results to label them into different clusters. Lingo is a novel algorithm for clustering search results, which emphasizes cluster description quality by using a Description-Comes-First technique which selects possible cluster labels first, usually the reverse of most other clustering algorithms.

3. Web Search Engine Concepts

3.1 Definition

A Web search engine is a search engine designed to search for information on the World Wide Web. Information may consist of web pages, images and other types of files. Some search engines also mine data available in newsgroups, databases, or open directories. Unlike Web directories, which are maintained by human editors, search engines operate algorithmically or are a mixture of algorithmic and human input.

A web search engine operates in the following order:
(i) Web Crawling
(ii) Indexing
(iii) Searching

3.2 Web Crawling

A web crawler is a program or automated script which browses the World Wide Web in a methodical, automated manner. Other less frequently used names for web crawlers are ants, automatic indexers, bots, and worms. This process is called web crawling or spidering. Many sites, in particular search engines, use spidering as a means of providing up-to-date data. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine that will index the downloaded pages to provide fast searches. Crawlers can also be used for automating maintenance tasks on a website, such as checking links or validating HTML code. Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting e-mail addresses. A web crawler is one type of bot, or software agent. In general, it starts with a list of URLs to visit, called the seeds. As the crawler visits these URLs, it

identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the crawl frontier. URLs from the frontier are recursively visited according to a set of policies.

The behavior of a web crawler is the outcome of a combination of policies:

- A selection policy that states which pages to download.
- A re-visit policy that states when to check for changes to the pages.
- A politeness policy that states how to avoid overloading websites.
- A parallelization policy that states how to coordinate distributed web crawlers.

3.2.1 Architecture of a Web Crawler

The simple scheme for crawling demands several modules that fit together as shown in Figure 3.1. The architecture consists of the following modules :

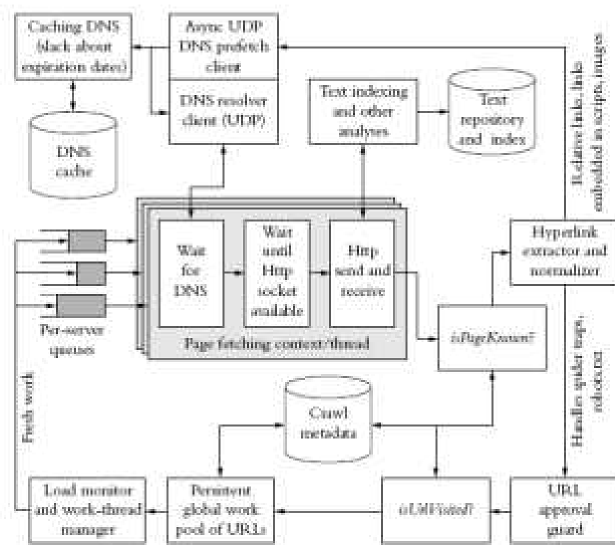

Figure 3.1 Architecture of a standard Web Crawler [7]

(i) The URL frontier, containing URLs yet to be fetched in the current crawl (in the case of continuous crawling, a URL may have been fetched previously but is back in the frontier for re-fetching).
(ii) A DNS resolution module that determines the web server from which to fetch the page specified by a URL.
(iii) A fetch module that uses the http protocol to retrieve the web page at a URL.
(iv) A parsing module that extracts the text and set of links from a fetched web page.
(v) A duplicate elimination module that determines whether an extracted link is already in the URL frontier or has recently been fetched.

Crawling is performed by anywhere from one to potentially hundreds of threads, each of which loops through the logical cycle in Figure 3.1. These threads may be run in a single process, or be partitioned amongst multiple processes running at different nodes of a distributed system. We follow the progress of a single URL through the cycle of being fetched, passing through various checks and filters, then finally (for continuous crawling) being returned to the URL frontier. A crawler thread begins by taking a URL from the frontier and fetching the web page at that URL, generally using the http protocol. The fetched page is then written into a temporary store, where a number of operations are performed on it. Next, the page is parsed and the text as well as the links in it is extracted. The text (with any tag information - e.g., terms in boldface) is passed on to the indexer. Link information including anchor text is also passed on to the indexer for use in ranking. In addition, each extracted link goes through a series of tests to determine whether the link should be added to the URL frontier.

First, the thread tests whether a web page with the same content has already been seen at another URL. Next, a URL filter is used to determine whether the extracted URL should be excluded from the frontier based on one of several tests. For instance, the crawl may seek to exclude certain domains (say, all .com URLs) - in this case the test would simply filter out the URL if it were from the .com domain. A similar test could be inclusive rather than exclusive. Many hosts on the Web place certain portions of their websites off-limits to crawling, under a standard known as the Robots Exclusion Protocol.

The robots.txt file must be fetched from a website in order to test whether the URL under consideration passes the robot restrictions, and can therefore be added to the URL frontier. Rather than fetch it afresh for testing on each URL to be added to the frontier, a cache can be used to obtain a recently fetched copy of the file for the host. This is especially important since many of the links extracted from a page fall within the host from which the page was fetched and therefore can be tested against the host's robots.txt file. Thus, by performing the filtering during the link extraction process, we would have especially high locality in the stream of hosts that we need to test for robots.txt files, leading to high cache hit rates. If we were to perform the robots filtering before adding such a URL to the frontier, its robots.txt file could have changed by the time the URL is dequeued from the frontier and fetched. We must consequently perform robots-filtering immediately before attempting to fetch a web page. As it turns out, maintaining a cache of robots.txt files is still highly effective; there is sufficient locality even in the stream of URLs dequeued from the URL frontier.

Next, a URL should be normalized in the following sense: often the HTML encoding of a link from a web page indicates the target of that link relative to the page. Finally, the URL is checked for duplicate elimination: if the URL is already in the frontier or (in the case of a non-continuous crawl) already crawled, we do not add it to the frontier. When the URL is added to the frontier, it is assigned a priority based on which it is eventually removed from the frontier for fetching.

Certain housekeeping tasks are typically performed by a dedicated thread. This thread is generally quiescent except that it wakes up once every few seconds to log crawl progress statistics (URLs crawled, frontier size, etc.), decide whether to terminate the crawl, or (once every few hours of crawling) checkpoint the crawl. In checkpointing, a snapshot of the crawler's state (say, the URL frontier) is committed to disk. In the event of a catastrophic crawler failure, the crawl is restarted from the most recent checkpoint.

3.3 Indexing

The purpose of storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power. For example, while an index of 10,000 documents can be queried within milliseconds, a sequential scan of every word in 10,000 large documents could take hours. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval.

Document	Words
Document1	the,cow,says,moo
Document2	the,cat,and,the,hat
Document3	the,dish,ran,away,with,the,spo on

A major challenge in the design of search engines is the management of parallel computing processes. There are many opportunities for race conditions and coherent faults. For example, a new document is added to the corpus and the index must be updated, but the index simultaneously needs to continue responding to search queries. This is a collision between two competing tasks. Consider that authors are producers of information, and a web crawler is the consumer of this information, grabbing the text and storing it in a cache (or corpus). The forward index is the consumer of the information produced by the corpus, and the inverted index is the consumer of information produced by the forward index. This is commonly referred to as a producer-consumer model. The indexer is the producer of searchable information and users are the consumers that need to search. The challenge is magnified when working with distributed storage and distributed processing. In an effort to scale with larger amounts of indexed information, the search engine's architecture may involve distributed computing, where the search engine consists of several machines operating in unison. This increases the possibilities for incoherency and makes it more difficult to maintain a fully-synchronized, distributed, parallel architecture.

3.3.1 The Forward Index

The forward index stores a list of words for each document. The following is a Simplified form of the forward index :
The rationale behind developing a forward index is that as documents are parsing, it is better to immediately store the words per document. The delineation enables Asynchronous system processing, which partially circumvents the inverted index update bottleneck. The forward index is sorted to transform it to an inverted index. The forward index is essentially a list of pairs consisting of a document and a word, collated by the document. Converting the forward index to an inverted index is only a matter of sorting the pairs by the words. In this regard, the inverted index is a word-sorted forward index.

3.3.2 Inverted Indices

Many search engines incorporate an inverted index when evaluating a search query to quickly locate documents containing the words in a query and then rank these documents by relevance. Because the inverted index stores a list of the documents containing each word, the search engine can use direct access to find the documents associated with each word in the query in order to retrieve the matching documents quickly. The following is a simplified Illustration of an inverted index :

Word	Documents
the	Document1,Document3,Document 4, Document 5
cow	Document 2, Document 3, Document 4
says	Document 5
moo	Document 7

Table 3.1 A simple Forward Index

This index can only determine whether a Table 3.2 A simple Inverted Index word exists within a particular document, since it stores no information regarding the frequency and position of the word; it is therefore considered to be a boolean index. Such an index determines which documents match a query but does not rank matched documents. In some designs the index includes additional information such as the frequency of each word in each document or the positions of a word in each document. Position information enables the search algorithm to identify word proximity to support searching for phrases; frequency can be used to help in ranking the relevance of documents to the query. Such topics are the central research focus of information retrieval.

The inverted index is a sparse matrix, since not all words are present in each document. To reduce computer storage memory requirements, it is stored differently from a two dimensional array. The index is similar to the term document matrices employed by latent semantic analysis. The inverted index can be considered a form of a hash table. In some cases the index is a form of a binary tree, which requires additional storage but may reduce the lookup time. In larger indices the architecture is typically a distributed hash table. Inverted indices can be programmed in several computer programming languages.

3.3.3 Index Merging

The inverted index is filled via a merge or rebuild. A rebuild is similar to a merge but first deletes the contents of the inverted index. The architecture may be designed to support incremental indexing, where a merge identifies the document or documents to be added or updated and then parses each document into words. For technical accuracy, a merge conflates newly indexed documents, typically residing in virtual memory, with the index cache residing on one or more computer hard drives.

After parsing, the indexer adds the referenced document to the document list for the appropriate words. In a larger search engine, the process of finding each word in the inverted index (in order to report that it occurred within a document) may be too time consuming, and so this process is commonly split up into two parts, the development of a forward index and a process which sorts the contents of the forward index into the

inverted index. The inverted index is so named because it is an inversion of the forward index.

3.3.4 Meta Tag Indexing

Specific documents often contain embedded meta information such as author, keywords, description, and language. For HTML pages, the meta tag contains keywords which are also included in the index. Earlier Internet search engine technology would only index the keywords in the meta tags for the forward index; the full document would not be parsed. At that time full-text indexing was not as well established, nor was the hardware able to support such technology. The design of the HTML markup language initially included support for meta tags for the very purpose of being properly and easily indexed, without requiring tokenization[9].

The keywords used to describe webpages (many of which were corporate-oriented webpages similar to product brochures) changed from descriptive to marketing-oriented keywords designed to drive sales by placing the webpage high in the search results for specific search queries. Search engine designers and companies could only place so many 'marketing keywords' into the content of a webpage before draining it of all interesting and useful information. Given that conflict of interest with the business goal of designing user-oriented websites which were 'sticky', the customer lifetime value equation was changed to incorporate more useful content into the website in hopes of retaining the visitor. In this sense, full-text indexing was more objective and increased the quality of search engine results, as it was one more step away from subjective control of search engine result placement, which in turn furthered research of full-text indexing technologies.

3.4 Web Search Query

A web search query is a query that a user enters into web search engine to satisfy his or her information needs. Web search queries are distinctive in that they are unstructured and often ambiguous; they vary greatly from standard query languages which are governed by strict syntax rules. There are three broad categories that cover most web search queries[8]:*Informational queries* – Queries that cover a broad topic (e.g., colorado or trucks) for which there may be thousands of relevant results. *Navigational queries* – Queries that seek a single website or web page of a single entity (e.g., youtube or delta airlines). *Transactional queries* – Queries that reflect the intent of the user to perform a particular action, like purchasing a car or downloading a screen saver.

Search engines often support a fourth type of query that is used far less frequently :

Connectivity queries – Queries that report on the connectivity of the indexed web graph (e.g., which links point to this URL? and How many pages are indexed from this domain name?).

3.4.1 Characteristics

Most commercial web search engines do not disclose their search logs, so information about what users are searching for on the Web is difficult to come by. Some exciting characteristics of a study[6] are :

(i) The average length of a search query was 2.4 terms.
(ii) About half of the users entered a single query while a little less than a third of users entered three or more unique queries.
(iii) Close to half of the users examined only the first one or two pages of results (10 results per page).
(iv) Less than 5% of users used advanced search features (e.g., Boolean operators like AND, OR, and NOT).
(v) The top three most frequently used terms were and, of, and sex.

In the study, Yahoo[18]'s query logs revealed 33% of the queries from the same user were repeat queries and that 87% of the time the user would click on the same result. This suggests that many users use repeat queries to revisit or re-find information.

In addition, much research has shown that query term frequency distributions conform to the power law, or long tail distribution curves. That is, a small portion of the terms observed in a large query log (e.g. > 100 million queries) are used most often, while the remaining terms are used less often individually. This example of the Pareto principle (or 80-20 rule) allows search engines to employ optimization techniques such as index or database partitioning, caching and pre-fetching.

4. Clustering Concepts

4.1 Definition

Clustering is the classification of objects into different groups, or more precisely, the partitioning of a data set into subsets (clusters), so that the data in each subset (ideally) share some common trait - often proximity according to some defined distance measure. Data clustering is a common technique for statistical data analysis, which is used in many fields, including machine learning, data mining, pattern recognition, image analysis and bioinformatics. The computational task of classifying the data set into k clusters is often referred to as k-clustering.

Besides the term data clustering (or just clustering), there are a number of terms with similar meanings, including cluster analysis, automatic classification, numerical taxonomy, botryology and typological analysis.

4.2 Benefits of Clustered Data

The idea of clustering originates from statistics where it was applied to numerical data. However, computer science and data mining in particular, have extended this notion to other types of data such as text or multimedia. Search results clustering is an attempt to apply the idea of clustering to document references (snippets) returned by a search engine in response to a query. Thus, it can be perceived as a way of organizing the snippets into a set of meaningful thematic groups. There are several ways in which end users can benefit from such a clustered view :

(i) **Fast access to relevant documents** — Having documents divided into clearly described categories, users who look for documents on a particular subject can navigate directly to the groups whose labels indicate relevant content. Similarly, most irrelevant documents can be skipped easily, again relying only on cluster description.
(ii) **Broader view of the search results** — Some users issue general queries to learn about the whole spectrum of available sub-topics. These users will no longer be made to

manually scan hundreds of references, and instead, they will be presented with a concise summary of all subjects dealt with in the results.

(iii) **Relevance feedback functionality** — With the ability to divide the results into sub-categories, search results clustering can also provide the functionality of relevance feedback, enabling the users to refine the initial query based on the labels of clusters generated for that query.

It is important to emphasize here that search results' clustering is performed after the documents matching the query have been identified by the search engine. Thus, this type of clustering can be regarded as a form of post-processing in presentation of search results.

4.3 Distance Measure

An important step in any clustering is to select a distance measure, which will determine how the similarity of two elements is calculated. This will influence the shape of the clusters, as some elements may be close to one another according to one distance and further away according to another.

Common distance functions are :

(i) The Euclidean distance (also called distance as the crow flies or 2-norm distance). A review of cluster analysis in health psychology research found that the most common distance measure in published studies in that research area is the Euclidean distance or the squared Euclidean distance.

(ii) The Manhattan distance (also called taxicab norm or 1-norm)

(iii) The maximum norm

(iv) The Mahalanobis distance corrects data for different scales and correlations in the variables

(v) The angle between two vectors can be used as a distance measure when clustering high dimensional data. See Inner product space.

(vi) The Hamming distance (sometimes edit distance) measures the minimum number of substitutions required to change one member into another.

4.4 Types of Clustering [4]

Data clustering algorithms can be hierarchical or partitional. Hierarchical algorithms find successive clusters using previously established clusters, whereas partitional algorithms determine all clusters at once. Hierarchical algorithms can be agglomerative ("bottom-up") or divisive ("top-down"). Agglomerative algorithms begin with each element as a separate cluster and merge them into successively larger clusters. Divisive algorithms begin with the whole set and proceed to divide it into successively smaller clusters.

Two-way clustering, co-clustering or biclustering are clustering methods where not only the objects are clustered but also the features of the objects, i.e., if the data is represented in a data matrix, the rows and columns are clustered simultaneously.

Another important distinction is whether the clustering uses symmetric or asymmetric distances. A property of Euclidean space is that distances are symmetric (the distance from object A to B is the same as the distance from B to A). In other applications (e.g., sequence-alignment methods), this is not the case.

4.4.1 Hierarchical Clustering

Hierarchical clustering builds (agglomerative), or breaks up (divisive), a hierarchy of clusters. The traditional representation of this hierarchy is a tree (called a dendrogram), with individual elements at one end and a single cluster containing every element at the other. Agglomerative algorithms begin at the top of the tree, whereas divisive algorithms begin at the root.

This method builds the hierarchy from the individual elements by progressively merging clusters. For example, we have six elements {a} {b} {c} {d} {e} and {f}. The first step is to determine which elements to merge in a cluster. Usually, we want to take the two closest elements, according to the chosen distance. Optionally, one can also construct a distance matrix at this stage, where the number in the i-th row j-th column is the distance between the i-th and j-th elements. Then, as clustering progresses, rows and columns are merged as the clusters are merged and the distances updated. This is a common way to implement this type of clustering, and has the benefit of caching distances between clusters. A simple agglomerative clustering algorithm is described in the single linkage clustering page; it can easily be adapted to different types of linkage.

Suppose we have merged the two closest elements b and c, we now have the following clusters {a}, {b, c}, {d}, {e} and {f}, and want to merge them further. To do that, we need to take the distance between {a} and {b c}, and therefore define the distance between two clusters. Usually the distance between two clusters and is one of the following :

- The maximum distance between elements of each
- The minimum distance between elements of each cluster
- The mean distance between elements of each cluster
- The sum of all intra-cluster variance
- The increase in variance for the cluster being merged
- The probability that candidate clusters spawn from the same distribution function

Each agglomeration occurs at a greater distance between clusters than the previous agglomeration, and one can decide to stop clustering either when the clusters are too far apart to be merged (distance criterion) or when there is a sufficiently small number of clusters (number criterion).

4.4.2 Partitional Clustering

Partitional clustering, on the other hand, attempts to directly decompose the data set into a set of disjoint clusters. The criterion function that the clustering algorithm tries to minimize may emphasize the local structure of the data, as by assigning clusters to peaks in the probability density function, or the global structure. Typically the global criteria involve minimizing some measure of dissimilarity in the samples within each cluster, while maximizing the dissimilarity of different clusters.

4.4.2.1 K-Means Clustering

The K-means algorithm assigns each point to the cluster whose

center (also called centroid) is nearest. The center is the average of all the points in the cluster — that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster. The algorithm steps are :

(i) Choose the number of clusters, k.
(ii) Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
(iii) Assign each point to the nearest cluster center.
(iv) Recompute the new cluster centers.
(v) Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed).

The main advantages of this algorithm are its simplicity and speed which allows it to run on large datasets. Its disadvantage is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. It minimizes intra-cluster variance, but does not ensure that the result has a global minimum of variance.

4.4.2.2 Fuzzy C-Means Clustering

In fuzzy clustering, each point has a degree of belonging to clusters, as in fuzzy logic, rather than belonging completely to just one cluster. Thus, points on the edge of a cluster may be in the cluster to a lesser degree than points in the center of cluster. The fuzzy c-means algorithm is very similar to the k-means algorithm :

(i) Choose a number of clusters.
(ii) Assign randomly to each point coefficients for being in the clusters.
(iii) Repeat until the algorithm has converged (that is, the coefficients' change between two iterations is no more than ε, the given sensitivity threshold)
(iv) Compute the centroid for each cluster, using the formula above.
(v) For each point, compute its coefficients of being in the clusters, using the formula above.

The algorithm minimizes intra-cluster variance as well, but has the same problems as k-means, the minimum is a local minimum, and the results depend on the initial choice of weights.

4.4.2.3 QT Clustering Algorithm

QT (quality threshold) clustering is an alternative method of partitioning data, invented for gene clustering. It requires more computing power than k-means, but does not require specifying the number of clusters a priori, and always returns the same result when run several times. The algorithm is :

(i) The user chooses a maximum diameter for clusters.
(ii) Build a candidate cluster for each point by including the closest point, the next closest, and so on, until the diameter of the cluster surpasses the threshold.
(iii) Save the candidate cluster with the most points as the first true cluster, and remove all points in the cluster from further consideration.
(iv) Recurse with the reduced set of points.

The distance between a point and a group of points is computed using complete linkage, i.e. as the maximum distance from the point to any member of the group.

4.5 Elbow Criterion

The elbow criterion is a common rule of thumb to determine what number of clusters should be chosen, for example for k-means and agglomerative hierarchical clustering. It should also be noted that the initial assignment of cluster seeds has bearing on the final model performance. Thus, it is appropriate to re-run the cluster analysis multiple times.

The elbow criterion says that you should choose a number of clusters so that adding another cluster doesn't add sufficient information. More precisely, if you graph the percentage of variance explained by the clusters against the number of clusters, the first clusters will add much information (explain a lot of variance), but at some point the marginal gain will drop, giving an angle in the graph (the elbow). On the following graph, the elbow is indicated by the circle. The number of clusters chosen should therefore be 4.

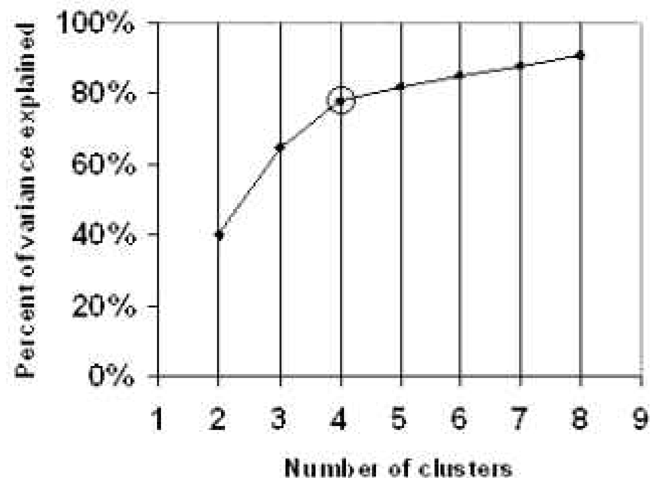

Figure 4.1 Elbow Criterion

5. The Nutch Component

5.1 Advent of Nutch

Search engines are as critical to Internet use as any other part of the network infrastructure, but they differ from other components in two important ways. First, their internal workings are secret, unlike, say, the workings of the DNS (domain name system). Second, they hold political and cultural power, as users increasingly rely on them to navigate online content. When so many rely on services whose internals are closely guarded, the possibilities for honest mistakes, let alone abuse, are worrisome. Further, keeping search-engine algorithms secret means that further advances in the area will become less likely. Much relevant research is kept behind corporate walls, and useful methods remain largely unknown.

To address these problems, the Nutch software project, an open

source search engine free for anyone to download, modify, and run, either as an internal intranet search engine or as a public Web search service was started. Much of the challenge in designing a search engine is making it scale. Writing a Web crawler that can download a handful of pages is straightforward, but writing one that can regularly download the Web's nearly 5 billion pages is much harder.

Further, a search engine must be able to process queries efficiently. Requirements vary widely with site popularity: a search engine may receive anywhere from less than one to hundreds of searches per second.

Finally, unlike many software projects, search engines can have high ongoing costs. They may require lots of hardware that consumes lots of Internet bandwidth and electricity. It's helpful to keep in mind a few ideas :

(i) The cost of one part of the search engine scales with the size of the document collection. The collection might be very small when Nutch is searching a single intranet, but could be as large as the Web itself.
(ii) Another part of the search engine scales with the size of the query load. Each query takes a certain amount of time to process and consumes some bandwidth.

With these two factors in mind, a system that can easily distribute the work of both fetching and query processing over a set of standard machines was designed. Nutch is an effort to build an open source search engine based on Lucene Java[15] for the search and index component. The web crawler has been written from scratch solely for this project. Nutch has a highly modular architecture allowing developers to create plugins for the following activities :

- media-type parsing
- data retrieval
- querying, and clustering.

5.2 Design of Nutch [10]

The Nutch search engine consists of three key components:

(i) The *fetcher*, a crawler which discovers and retrieves web pages
(ii) The *WebDB*, a custom database that stores known URLs and fetched page contents
(iii) The *indexer*, which dissects pages and builds keyword-based indexes from them.

Apart from the above three components, it has a Search Web Application. This application is a JSP application that can be configured and deployed in a servlet container.

The WebDB is a persistent custom database that tracks every known page and relevant link. It maintains a small set of facts about each, such as the last-crawled date. WebDB is meant to exist for a long time, across many months of operation.

Since WebDB knows when each link was last fetched, it can easily generate a set of fetchlists. These lists contain every URL we're interested in downloading. WebDB splits the overall workload into several lists, one for each fetcher process. URLs are distributed almost randomly; all the links for a single domain are fetched by the same process, so it can obey politeness constraints.

The fetchers consume the fetchlists and start downloading from the Internet. The fetchers are "polite," meaning they don't overload a single site with requests, and they observe the Robots Exclusion Protocol. (This allows Web-site owners to mark parts of the site as off-limits to automated clients such as our fetcher.) Otherwise, the fetcher blindly marches down the fetchlist, writing down the resulting downloaded text. Fetchers output WebDB updates and Web content. The updates tell WebDB about pages that have appeared or disappeared since the last fetch attempt. The Web content is used to generate the searchable index that users will actually query. Note that the WebDB-fetch cycle is designed to repeat forever, maintaining an up-to-date image of the Web graph. Once the Web content is obtained, Nutch can get ready to process queries. The indexer uses the content to generate an inverted index of all terms and all pages. The document set is divided into a set of index segments, each of which is fed to a single searcher process.

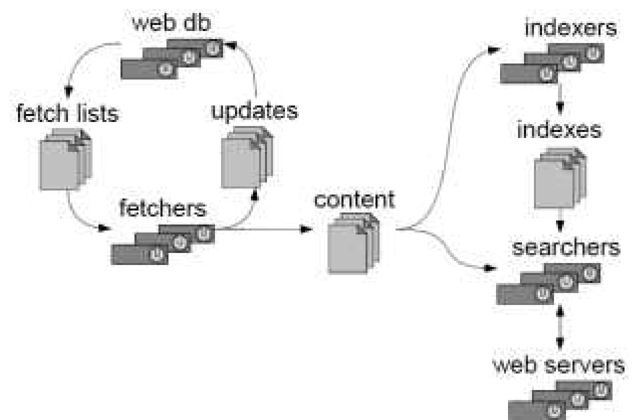

Figure 5.1 The Nutch Architecture

The current set of index segments are then distributed over an arbitrary number of searcher processes, allowing easily scalability with the query load. Further, an index segment can be copied to multiple machines and searcher can be run over each one; that allows more good scaling behavior and reliability in case one or more of the searcher machines fail.

Each searcher also draws upon the Web content from earlier, so it can provide a cached copy of any Web page. Finally, a pool of Web servers handle interactions with users and contact the searchers for results. Each Web server interacts with many different searchers to learn about the entire document set. In this way, the Web server is simultaneously acting as an HTTP server and a Nutch-search client. Web servers contain very little state and can be easily reproduced to handle increased load. They need to be told only about the existing pool of

searcher machines. The only state they do maintain is a list of which searcher processes are available at any time; if a given segment's searcher fails, the Web server will query a different one instead.

5.3 The Nutch Api[13]

All components listed above use the nutch API. The users can utilize the API via two approaches, which depends on the task at hand :

(i) Through the nutch Shell Script for administrative tasks, such as creating and maintaining indexes
(ii) Through the Search Web Application, in order to perform a search using keywords

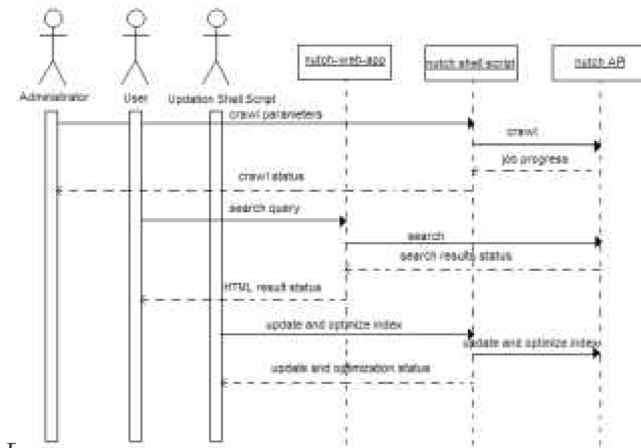

Figure 5.2 The Nutch sequence diagram

The sequence diagram in Figure 5.2 shows how each of these components interact in implementing a Nutch based search application.

6. Carrot² Component

6.1 About Carrot²

Carrot² is an Open Source Search Results Clustering Engine. It can automatically organize (cluster) search results into thematic categories.Carrot² provides an architecture for acquiring search results from various sources (YahooAPI, GoogleAPI, MSN Search API, eTools Meta Search, Alexa Web Search, PubMed, OpenSearch, Lucene index, SOLR), clustering the results and visualizing the clusters. Currently, 5 clustering algorithms are available that are suitable for different kinds of document clustering tasks. Due to its flexible architecture, high quality and a friendly BSD-like license, Carrot² has been successfully used in a number of commercial and research applications and resulted in a number of interesting publications.

6.2 The Carrot² Architecture

Carrot² is based on a pipeline of components of three types: input components, filter components and visualization components. The task of input components is to provide search results for clustering based on a user query. Filter components transform the results in some way (e.g. apply clustering, case normalization), and the visualization components render the transformed results for the user.

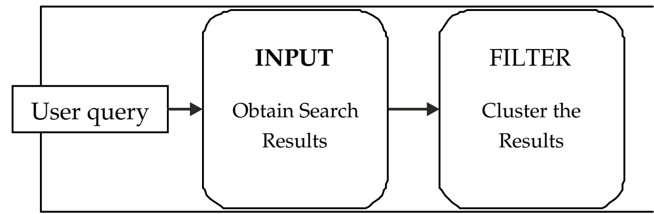

Figure 6.1 Flow diagram of Carrot²

Ready-to-use input components are available in Carrot² for reading search results from :

- Yahoo! search engine (using Yahoo Web Seach API)
- Google search engine (using Google API)
- MSN search engine (using MSN Search Web Services)
- eTools Meta Search
- Alexa Web Search
- PubMed
- OpenSearch-compliant search engine
- Lucene index (requires access to the index files)
- A local XML file

6.3 Clustering Algorithm [2]

The algorithm used in Carrot2 is a clustering algorithm called Lingo in which special emphasis is placed on the quality of group labels. The main idea behind the algorithm is to reverse the usual order of the clustering process: Lingo first identifies potential cluster labels using a dimensionality reduction technique called Singular Value Decomposition (SVD), and only then assigns documents to these labels to form proper thematic groups.

When designing a web search clustering algorithm, special attention must be paid to ensuring that both content and description (labels) of the resulting groups are meaningful to humans. The majority of open text clustering algorithms follows a scheme where cluster content discovery is performed first, and then, based on the content, the labels are determined. But very often intricate measures of similarity among documents do not correspond well with plain human understanding of what a cluster's "glue" element has been. To avoid such problems Lingo reverses this process—human-perceivable cluster label is created and only then assign documents to it. Specifically, frequent phrases from the input documents are extracted, hoping they are the most informative source of human-readable topic descriptions. Next, by performing reduction of the original term-document matrix using SVD, any existing latent structure of diverse topics in the search result is discovered. Finally, group descriptions are matched with the extracted topics and assign relevant documents to them.

6.3.1 Preprocessing

Stemming and stop words removal is very common operations

in Information Retrieval. Interestingly, their influence on results is not always positive — in certain applications stemming yielded no improvement to overall quality. Be as it may, current experiments show that preprocessing is of great importance in Lingo because the input snippets are automatically generated summaries of the original documents and hence are usually very small (one or two sentences). Although SVD is capable of dealing with noisy data, without sufficient preprocessing, the majority of discovered abstract concepts would be related to meaningless frequent terms. The aim of the preprocessing phase is to prune from the input all characters and terms that can possibly affect the quality of group descriptions. Three steps are performed: text filtering removes HTML tags, entities and non-letter characters except for sentence boundaries. Next, each snippet's language is identified and finally appropriate stemming and stop words removal end the preprocessing phase. Stop words are used as potential indicators of a document's language. Other methods, such as n-gram language detection could be used alternatively. For stemming English documents Porter's algorithm was used.

6.3.2 Frequent phrase extraction

Frequent phrases are defined as recurring ordered sequences of terms appearing in the input documents. Intuitively, when writing about something, the subject-related keywords are repeated to keep a reader's attention. Obviously, in a good writing style it is common to use synonymy and pronouns and thus avoid annoying repetition. Lingo can partially overcome the former by using the SVD-decomposed term document matrix to identify abstract concepts—single subjects or groups of related subjects that are cognitively different from other abstract concepts.

To be a candidate for a cluster label, a frequent phrase or a single term must :

(i) appear in the input documents at least certain number of times (term frequency threshold),
(ii) not cross sentence boundaries,
(iii) be a complete phrase
(iv) not begin nor end with a stop word.

A complete phrase is a complete substring of the collated text of the input documents such that it cannot be "extended" by adding preceding or trailing elements, because at least one of these elements is different from the rest.

6.3.3 Cluster Label Induction

Once frequent phrases (and single frequent terms) that exceed term frequency thresholds are known, they are used for cluster label induction. There are three steps to this: term-document matrix building, abstract concept discovery, phrase matching and label pruning. The term-document matrix is constructed out of single terms that exceed a predefined term frequency threshold. Weight of each term is calculated using the standard term frequency, inverse document frequency, terms appearing in document titles are additionally scaled by a constant factor. In abstract concept discovery, Singular Value Decomposition method is applied to the term-document matrix to find its orthogonal basis.

The Phrase matching and label pruning step, where group descriptions are discovered, relies on an important observation that both abstract concepts and frequent phrases are expressed in the same vector space—the column space of the original term-document matrix. Thus, the classic cosine distance can be used to calculate how "close" a phrase or a single term is to an abstract concept.

6.3.4 Cluster Content Discovery

In the cluster content discovery phase, the classic Vector Space Model is used to assign the input documents to the cluster labels induced in the previous phase. In a way, we re-query the input document set with all induced cluster labels. The assignment process resembles document retrieval based on the VSM model. Let us define matrix Q, in which each cluster label is represented as a column vector. Let C = QTA, where A is the original term-document matrix for input documents. This way, element cij of the C matrix indicates the strength of membership of the j-th document to the i-th cluster. A document is added to a cluster if cij exceeds the Snippet Assignment Threshold, yet another control parameter of the algorithm. Documents not assigned to any cluster end up in an artificial cluster called Others.

6.3.5 Final cluster formation

Finally, clusters are sorted for display based on their score, calculated using the following simple formula: Cscore = label score × kCk, where kCk is the number of documents assigned to cluster C. The scoring function, although simple, prefers well-described and relatively large groups over smaller, possibly noisy ones.

7. Proposed Work

This project aims at creating a web search engine that performs :

(i) Web crawling, to create a copy of all the visited pages to provide fast searches.
(ii) indexing, to collect, parse, and store data to facilitate fast and accurate information retrieval
(iii) Web searching, based on queries consisting of keywords.
(iv) Display, showing results in a decreasing order of relevance to the search query keywords.
(v) Clustering, where the displayed results must be presented in groups of related web pages, called clusters.

Phase	Objectives
I	Get Nutch running on a Linux platform
II	Crawl and index sample pages Test intermediate results using sample queries Implement clustering techniques to improve search results
III	Perform final testing on results to search queries Test cluster labels

Table 1 Project Phases

8. Conclusions

Although it is true that the industry standard search engines such as Google and Yahoo probably cost a lot of money to operate, many Web search engines may not serve nearly as much traffic and need not search nearly so many pages. In a world with lots of deployed search engines, the vast majority will serve small audiences. The costs are also well within reach for research groups, governmental departments, and small- to medium-size companies.

Governments, universities, and nonprofits are terrific candidates for this search engine. These organizations often have special obligations that for-profit companies don't (e.g., a seniors' organization might want to offer search with a special usability focus), so having the source code to Nutch is a huge advantage. Further, these groups often don't have lots of cash to spend on solutions.In addition, by clustering the results we looked at a clustering algorithm that generalized different dimensionality reduction techniques and choose the one that would perform best in terms of clustering quality and computational efficiency. Thus, the search results can be viewed with and without clustering in a side-by-side manner.

9. Future Work

This project can be extended in two main ways :

(i) **the search engine** - extending Nutch by writing new plugins at its extension points.

(ii) **clustering** - implementing and comparing other clustering techniques

9.1 The Search Engine

As seen earlier, the Nutch based search engine has a plugin system that offers many extension points. On these extension points, plugins can be written to add new features such as an improved indexer, a parser that can handle more document types or a more efficient crawling engine.More details on how to extend the search engine can be found at Nutch's Plugin Central[14] page.

9.2 Clustering

The clustering algorithm used in Carrot^2 was carefully chosen as one that would perform best given the size of our project. However, several other clustering algorithms are available in Carrot^2 which can be implemented based on the quantity of search results per query and their performances can be analyzed and compared.The clustering algorithms supported by Carrot^2 can be found in the project's algorithms[15] page.

10. References

1. *Doug Cutting, "Nutch: Open Source Web Search", 22 May 2004, WWW2004, New York.*
2. *Stanisław Osiński, Jerzy Stefanowski, Dawid Weiss: Lingo: Search Results Clustering Algorithm Based on Singular Value Decomposition. Advances in Soft Computing, Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM´04 Conference, Zakopane,*
3. *Stanislaw Osinski and Dawid Weiss. Lingo: A Concept-Driven Algorithm for Clustering Search Results. Accepted for IEEE Intelligent Systems, 2004.*
4. *S. Kotsiantis, P. Pintelas, Recent Advances in Clustering: A Brief Survey, WSEAS Transactions on Information Science and Applications, Vol 1, No 1 (73-81), 2004.*
5. *Nielsen NetRatings: August 2007 Search Share Puts Google On Top, Microsoft Holding Gains, SearchEngineLand, September 21, 2007.*
6. *Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, Tefko Saracevic (2001). "Searching the web: The public and their queries". Journal of the American Society for Information Science and Technology 52 (3): 226-234.*
7. *Soumen Chakrabarti, Mining the Web – Discovering Knowledge from Hypertext data, Morgan Kaufman, 2003. ISBN 1-55860-754-4.*
8. *Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze (2007), Introduction to Information Retrieval, Ch. 19.*
9. *PC Technology Guide: Guides/Storage/Hard Disks. PC Technology Guide. 2003. Verified Dec 2006.*