A Hybrid Model for a Search Engine

Shikha Mehta, Ankush Gulati, and Ankit Kalra

Abstract—**The large size and the dynamic nature of the Web highlight the need for continuous support and updating of Web based information retrieval systems such as search engines. Due to resource constraints, search engines usually have difficulties in striking the right balance between time and space limitations. In this paper, we propose a simple yet effective model for a search engine. We suggest a hybrid design which brings together the best features that constitute a search engine. To give an overview, the whole mechanism of how a search engine works is provided. Further, the model is discussed in detail. We then demonstrate how the proposed model, which embodies features like Fingerprinting, Compressor, Importance number and Refresher can improve the efficiency of a simple search engine if applied on a large scale.**

Index Terms—Crawler, Web search engine, refresh policy, search methods, indexing, Distributed information systems.

I. Introduction

THE World Wide Web has grown from a few thousand pages in its early days to more than two billion pages at present. A large number of analyses have been made on the size of the web. Conclusions are drawn that the web is still growing at an exponential pace [2]. Moreover, the web is not structured at all and finding your desired page on the web without a search engine can be a painful task. That is why; search engines have grown into by far the most popular way for navigating the web. In fact, search engines were also known as some of the brightest stars in the Internet frenzy that occurred in the late 1990s.

Engineering a search engine is a challenging task. Search engines rely on massive collections of web pages that are acquired with the help of web crawlers, which traverse the web by following hyperlinks and storing downloaded pages in a large database that is later indexed for efficient execution of user queries. Many researchers have looked at web search technology over the last few years, including crawling strategies, storage, indexing, and ranking techniques as a complex issue. The key to a search engine is, that it needs to be equipped with an intelligent navigation strategy [7], i.e. enabling it to make decisions regarding the choice of subsequent actions to be taken (pages to be downloaded etc). In this paper, we propose an architecture which can be helpful in improving the efficiency of search engines. Our main goal is to improve the quality of web search engines and build an architecture that can search the ever growing web data in a better way.

The rest of the paper is organized as follows. We begin with a definition of a basic search engine in section 2, giving an overview of how does a search engine work and what are its key components. In Section 3, we investigate the need of an advanced search engine. Section 4 suggests some possible solutions for those problems. In Section 5, we the present a 'hybrid' model for search engines- *Swift* which incorporates the best features possible model and discuss every module in detail. Finally, we discuss the scope and future work possible in this direction in Section 6 and outline our conclusion in Section 7.

II. Working of a Search Engine

Internet search engines are special sites on the Web that are designed to help people find information stored on other sites. A search engine basically consists of four parts. Figure 1 shows a basic search engine model [2] :

- *Crawlers*: They search the Internet -- or select pieces of the Internet -- based on important words.
- *Repository*: It stores the complete HTML of every relevant page crawled by the Crawler.
- *Indexer*: It creates an index of the pages they find, on the basis of their linking with other pages.
- *Searcher*: It allows users to look for words or combinations of words in the local repository. The complete working of a search engine is dependent on the flow of data among the above mentioned modules.

A. Crawler

A Web crawler (also known as spider) is a program, which automatically traverses the web by downloading documents and following links from page to page. Crawlers are mainly used by web search engines to gather data for populating the repository. It starts with a few seed pages and then uses the external links within them to attend to other pages. The process repeats with the new pages offering more external links to follow.

We may think that the job of a crawler is over when all the pages have been fetched to the repository once, but there is another important task that a crawler has to perform, refreshing. The Web is not a static collection of pages. It is a dynamic entity evolving and growing every minute. Hence there is a continual need for crawlers to help applications stay current as new pages are added and old ones are deleted,

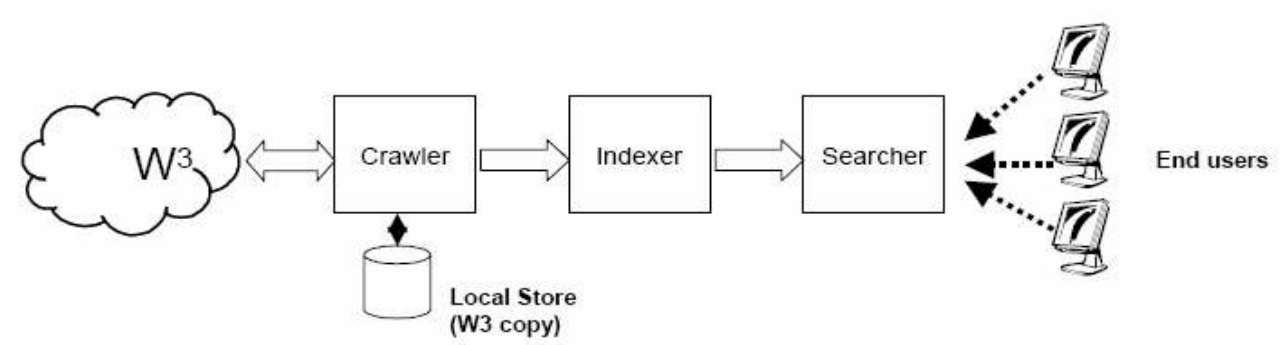

Figure 1 – Model of a Basic Search Engine

moved or modified.

In technical terms, crawling can be viewed as a graph search problem [5]. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes. The process of fetching a page and extracting the links within it is analogous to expanding a node in graph search.

B. Local Repository

Everything the spider finds goes into the second part of the search engine, the repository. The repository stores and manages a large collection of 'data objects' in this case web page. All the pages that a crawler crawls and finds relevant are downloaded and stored in the repository. The repository acts as the local cache for this information retrieval system. Whenever, a user searches for a keyword, the searcher module looks into the repository and prints the results.

C. Indexer

An Indexer is a program that "reads" the pages, which are stored in the repository. Even though, each web database has a different indexing method (Brin and Page 1998), the indexer mostly decides what the web site is about and how the website is linked to the rest of the web. It reads the repository, decompresses the documents, and parses out all the links in every web page and stores important information about them to determine where each link points from and to, and the text of the link.

Furthermore, the indexer also does the job of ranking pages on the basis of their importance in the result set. The ranking module consists of a rank distribution algorithm which assigns a random rank to a web page and then computes the rank of other web pages. The algorithms that are commonly used for the purpose of ranking are HITS, Page Rank algorithm and many more [14].

D. Searcher

This is the program that sifts through the millions of pages recorded in the database to find matches to a specific search.

The searcher works on the output files from the indexer. It accepts user queries, runs them over the index, and returns computed search results to the issuer. The searcher is run by a web server and uses the Page Ranks to answer queries.

III. NEED OF AN ADVANCED SEARCH ENGINE

The enormous growth in information that we want to publish on the web has created the need and space for more advanced information retrieval systems to help fetch the information effectively. Many reasons can be cited for the need of an advanced search engine.

- **Complex Structure of the web:** The internet has been aptly named as the Web because of its structure. The web is not organized and its complicated structure creates a lot of problems in effective management of data on the web. The hypertext documents are linked with each other through hyperlinks within them. This gives the user, the ability to choose what he will see next. Interestingly, there can be various links on different pages leading to the same document. A simple crawl mechanism may lead us to a voluminous database with a high degree of redundant data. Thus the crawler needs to have a good crawling strategy [10], i.e., a strategy for deciding whether to download the next page or not, by selecting only one of the many paths available for the same page and hence, avoid data redundancy. It may not seem to be an important issue but when the size and the structure of the web are taken into account, this problem can have deadly consequences on the effectiveness of the search engine.

- **Dynamic nature of the web:** It is an important factor for large-scale Internet search engines. We can broadly classify the issue into three cases :
    1. *Pages Added:* The web is growing in size, and most studies agree that it grows at an exponential rate. This poses a serious challenge of scalability for search engines that aspire to cover a large part of the web. Pages are added everyday and it is the responsibility of the search engine to continuously grow and update its database about the latest link structure of the web.
    2. *Pages Updated:* Apart from the newly

created pages, the existing pages are continuously updated [14]. Newer and more relevant information is added and the older ones are removed. Websites like news portals, etc are updated almost every minute and if the search engine's database is not updated with the current information, it is of no use to the user. Thus, the search engine must store a fresh copy of the pages stored.

3. *Pages Deleted:* Finally, the problem of unavailability of pages also needs to be addressed. The less relevant pages are removed from time to time by the servers. The search engine must keep a check on the availability of the pages it has stored in its local database and remove there links if they are no more hosted.

- **Vast Ocean of data (WWW) to be used as the database to search from:** Size of web cannot be calculated in less than petabytes and crawling the entire web can be a cumbersome task. According to a study, interestingly, the highly relevant content is found very deep in the web. Hence, it can be seen as a problem where we have the limitation of both space and time. In the context of space, we need a local database of size that can store a copy of almost each and every page crawled by the crawler. Taking the time restriction into account, we need to have an efficient algorithm which can search the giant database fast enough to give the desired results in ample time.

IV. Proposed Solution

On the basis of our study of the problems a basic search engine faces, we have come up with a solution which addresses most of the issues discussed in the previous section.

We explored that crawlers consume the maximum amount of resources [9]: network bandwidth to download pages, CPU to evaluate and select URLs, and disk storage to store the text and links of fetched pages as well as other persistent data. Hence, there is a need to improve the working of the crawler considerably.

Issues like **complex structure of the web** can be resolved by using special techniques such as *URL matching* and *Content matching* [10] where in all the pages downloaded by the crawler are first inspected for there content and compared with the copies available in the local database to avoid redundancy.

Similarly, the **large amount of data** can be handled efficiently if it is classified on the basis of some parameters. For example, we can divide the entire database on the basis of content and store pages related to one category in one database and other category in another and so on. We can further reduce the amount of space required to store the data by applying some common compression- decompression techniques [9] on the database.

Also, the **time constraint** can be handled with good *indexing techniques* and we can provide quality search results using a *rank distribution algorithm* [14].

Lastly, there is the issue of **dynamic nature of the web**. This problem can not be easily sorted out. Some smart and effective methods are needed if this issue has to be dealt with. Continuous changes in the web have to be matched with powerful **refreshing techniques** [2]. The local database should be consistently updated with the latest copies of updated pages.

V. Design Of The Proposed Model

In this section we give a detailed presentation of the design of a 'Hybrid' model. *Swift* is a distributed and scalable architecture which is extensible as well. The entire model has been designed in a way that new modules can be added any time for further improvements. Figure 2 shows the complete design of *Swift*.

The key features which make *Swift* a 'Hybrid' model are Compressor-Decompressor, Fingerprint matcher, Importance value calculator, Ranking Algorithm, Focused Crawlers and Smart Refresher algorithm in a Distributed setup.

In the proposed model, the crawling process starts with a few URLs provided. It generates a repository of hundreds and thousands of pages from them and further, refreshes the local database from time to time. The content matching and database updation mechanisms follow the crawling operation. This complete process continues in the background repeatedly. When the user searches for a keyword, the *Searcher* and the *Indexer* modules use the *repository* in its current state as the database to search from. Figure 2 shows the flow of data among the components of *Swift*.

A. Features

- **Distributed Architecture:**

The design proposed is based on a completely distributed architecture. The distribution of jobs to agents is an important problem, crucially related to the efficiency of the system. Therefore, each task must be performed in a fully distributed fashion, that is, no central coordinator can exist. Every agent interacts with either some agent or the *Repository* for taking the input or giving the processed output. Even the *Refresher* and *Extractors* are further distributed for applying the Focused Crawling approach.

- **Fingerprinting:**

Every time a page is downloaded by an *Extractor*, a 64-bit key is generated by applying MD5 algorithm [10] on the contents of the document. We call this key, a fingerprint of this page. This fingerprint can be used by both the *Refresher* and the *Content Seen Tester*. For each newly collected document, if we verify its fingerprint against the fingerprint of the previously collected documents, we can

certainly reduce the data redundancy problem to a large extent and hence, address the space limitations.

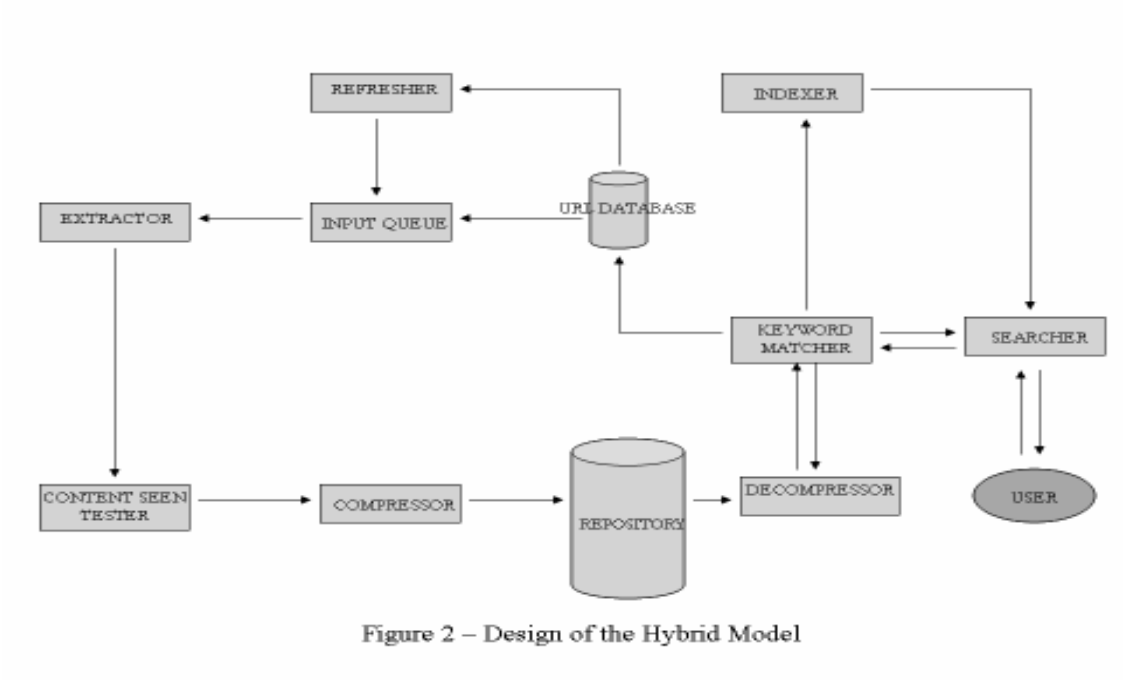

Figure 2 – Design of the Hybrid Model

- **Compression-Decompression:**

We can further reduce the amount of space required to an astonishing degree by using a few simple data compression techniques on the documents before storing them in the *repository*. During the testing phase of this module, we could reduce the size of the local repository by about 60% of the original size. Thus, this feature if taken into account can cause serious improvements.

- **Heterogeneous crawling:**

As the size of the Web grows, it becomes imperative to parallelize a crawling process, in order to finish downloading pages in a reasonable amount of time [13]. This feature suggests a crawler cluster with dedicated machines for crawling the web heterogeneously on the basis of the content.

- **Smart Refreshing techniques:**

Even though there is an established protocol, Robot Exclusion Protocol [7] that can be used to get information about the page. very few websites actually implement this protocol and incorporate it in their pages. We follow the approach suggested by Risvik and Michelsen [2]. This approach uses a relatively simple algorithm for adaptively computing an estimate of the refresh frequency for a given document. Basically, this algorithm decreases the refresh interval if the document was changed between two retrievals and increases it if the document has not changed [2]. This is used as input to the scheduler, which prioritizes between the different documents and decides when to refresh each document. To decide whether the page has changed since the previous crawl, we apply a smart technique. On retrieving the document, we calculate its fingerprint and match it with its existing fingerprint. If they verify, it suggests that page has not changed and thus, we discard the new page. But if they do not verify, the old page is replaced by the new one and its fingerprints are updated in the *URL database*.

- **Importance value:**

A small yet effective feature that gives weight age to the user's choices. Whenever a page is accessed by the user, its *importance value* is raised depicting that the page is a useful one. By default, it is a standard integer value assigned to every URL which gets incremented by 1 every time the URL is visited by the User.

- **Ranking algorithm:**

Due to the Web's size and the fact that users typically only enter one or two keywords, result sets are usually very large. Hence the ranking module has the task of sorting the results such that results near the top are the most likely to be what the user is looking for. In our model, we incorporate the Page Rank algorithm. The crawled portion of the Web is modeled as a graph with nodes and edges. Each node in the graph is a Web page, and a directed edge from node A to node B represents a hypertext link in page A that points to page B [14]. Page Rank is a link analysis algorithm that assigns a numerical weighting to each node of the graph generated with the purpose of "measuring" its relative importance.

B. Working

The working of the model can be explained as follows. To begin with, the *Input Queue* takes a list of seed URLs as its input from the *URL Database* and the *Extractors* repeatedly execute the following steps. Remove a URL from the queue, download the corresponding document, and extract any links

contained in it. With the help of the *Content Seen Tester*, it is ensured that no extracted file is encountered before and there does not exist a copy of the same in the *Repository*. After the document has passed the *Content Seen Test*, its *Fingerprint* is saved in the *URL Database*. The document is then sent to the *Compressor* module which compresses the document and stores it in the *Repository*.

Alongside, the refresher module works on refreshing the populated Repository. It takes the already visited URLs from the URL Database and downloads them. The refresher then applies the refreshing algorithm with the help of Fingerprinting mechanism and updates the local Repository with the fresh copies of existing URLs.

On the other hand, when a user searches for a specific keyword(s), the Searcher module fires a query to the Keyword Matcher. The Keyword Matcher requests the Decompressor to decompress the documents stored in Repository one by one. It searches for the requested keyword(s) within each page it gets from the Decompressor and forwards the results to the Indexer. The Indexer further generates a graph of resulting documents and calculates a rank for each document. The Searcher fetches the results from the Indexer and displays the results in an ascending order of the ranks calculated. Finally, for all the links that are accessed by the User, an update is sent by the Searcher to the URL database to increment its Importance Value.

VI. Future Work

The size of the web is clearly a big challenge, and future evolution of web dynamics raises clear needs for even more intelligent models. One important dimension to be worked upon is the search quality. Search quality means being able to intelligently manipulate the query and fetch results that are as close as possible to the desired output. Features like keyword lexicon can be incorporated in the existing model.

Another important research direction is to study more sophisticated text analysis techniques [8]. At the same time, the "Deep Web" is most likely growing at a rate much higher than the current "indexable" web. There is no unified and practical solution to aggregate the deep web on a large scale.

VII. Conclusion

We have presented Swift, a fully distributed, scalable, incremental and extensible model. We believe that Swift introduces new ideas in intelligent information systems, in particular the search engines. Swift is an ongoing project, and our current goal is to successfully implement the proposed model. We have described the architecture and the operation of Swift in detail. Also, we have discussed the working of a search engine and highlighted how problems arise in all components of a basic search engine model. Swift copes with several of these problems by its key properties like Fingerprinting, Importance Value, Compression-Decompression, Smart Refresher Techniques and Ranking algorithms in a distributed environment. The overall architecture that we have described in this paper is quite simple and does not represent very novel ideas. The system architecture is relatively simple and hence, easy to grow.

References

[1] Qiang Zhu , "An Algorithm OFC For The Focused Web Crawler" in *Proceedings of the Sixth International Conference , Hong Kong, Aug .2007*

[2] Knut Magne Risvik , Rolf Michelsen, *"*Search Engines and Web Dynamics*"*.

[3] Qingzhao , Tan,Prasenjit Mitra , C.Lee Giles, "Designing Clustering-Based Web Crawling Policies for Search Engine Crawlers"

[4] Altigran S. da Silva , Eveline A. Veloso , Paulo B. Golgher "CoBWeb – A Crawler for the Brazilian Web".

[5] Gautam Pant , Padmini Srinivasan , Filippo Menczer " Crawling the Web"

[6] Vladislav Shkapenyuk , Torsten Suel " Design and Implementation of a High-Performance Distributed Web Crawler ".

[7] Younes Hafri , Chabane Djeraba " Dominos : A New Web Crawler's Design ".

[8] Brian Pinkerton " WebCrawler : Finding What People Want ".

[9] Monica Peshave " How Search Engines Work And a Web Crawler Application ".

[10] Allan Heydon < Marc Najork " Mercator : A scalable , Extensible Web Crawler" June 1999.

[11] Junghoo Cho , Hector Garcia-Molina " Parallel Crawlers ".

[12] Carlos Castillo , Mauricio Marin , Andrea Rodriguez " Scheduling Algorithms for Web Crawling ".

[13] Paolo Boldi , Bruno , Massimo Santini , Sebastiano Vigna " UbiCrawler : A scalable Fully Distributed Web Crawler ".

[14] Arvind Arasu , Junghoo Cho , Hector Garcia-Molina , Andreas Paepcke , Sriram Raghavan "Searching the Web".

[15] Behnak Yaltaghian , Mark Chignell " Re-ranking Search Results using Network Analysis - A case study with Google".

[16] Behnak Yaltaghian , Mark Chignell " Effect of Different Network Analysis Strategies on Search Engine Re-Ranking ".

[17] Michelangelo Diligenti , Marco Gori , Marco Maggini " Web Page Scoring Systems for Horizontal and Vertical Search".

[18] Thomas Mandl " Implementation and Evaluation of a Quality Based Search Engine ".