A framework for dynamic indexing from hidden web

Hasan Mahmud[1], Moumie Soulemane[2], Mohammad Rafiuzzaman[3]

[1] Department of Computer Science and Information Technology, Islamic University of Technology, Board Bazar, Gazipur-1704, Bangladesh.

[2] Department of Computer Science and Information Technology, Islamic University of Technology, Board Bazar, Gazipur-1704, Bangladesh.

[3] Department of Computer Science and Information Technology, Islamic University of Technology, Board Bazar, Gazipur-1704, Bangladesh.

Abstract

The proliferation of dynamic websites operating on databases requires generating web pages on-the-fly which is too sophisticated for most of the search engines to index. In an attempt to crawl the contents of dynamic web pages, we've tried to come up with a simple approach to index these huge amounts of dynamic contents hidden behind the search forms. Our key contribution in this paper is the design and implementation of a simple framework to index the dynamic web pages and the use of Hadoop MapReduce framework to update and maintain the index. In our approach, from an initial URL, our crawler downloads both the static and dynamic web pages, detects form interfaces, adaptively selects keywords to generate most promising search results, automatically fill-up search form interfaces, submits the dynamic URL and processes the result until some conditions are satisfied.

Keywords: *Dynamic web pages, crawler, hidden web, index, hadoop.*

.

1. Introduction

Web mining is an application of data mining which aims to discover useful information or knowledge from the web hyperlink structure, page content and usage log. Based on the primary kind of data used in the mining process, web mining tasks are categorized into four main types: (1) Web usage mining, (2) Web structure mining, (3) Web user profile mining and (4) Web content mining [7, 14]. In the past few years, there was a rapid expansion of activities in the web content mining area. This is not surprising because of the phenomenal growth of the web contents and significant economic benefits of such mining. Given the enormous size of the web, the indexed web contains at least 13.85 billion pages [9]. Many users today prefer to access web sites through search engines. A number of recent studies have noted that a tremendous amount of content on the web is dynamic. According to [8] Google, the largest search database on the planet, currently has around eight billion web pages which are already indexed. That's a lot of information. But it's nothing compared to what else is out there. Google can only index the visible web, or searchable web which refers to the set of web pages reachable purely by following hypertext links. But the invisible web or deep web [4, 5, 16, 17, 21], "hidden" behind search forms is estimated to be 500 times bigger than the searchable web. However, a little of this tremendous amount of high quality dynamic contents are being crawled or indexed and in particular, most of them are ignored.

In this paper the focus is on the automatic indexing of dynamic web contents which are the part of deep web. It is same as web content mining as we're extracting the words included in web pages. Here we've tried to come up with a simple approach to crawl the textual portion of dynamic contents hidden behind search forms with the following techniques:

- **Dynamic content extraction:** Extraction of structured data, hidden behind the search forms of Web pages, such as search results. Extracting such data allows one to provide services, so search engines will be benefited if we index dynamic contents of the web pages as most of the time their crawlers avoid those pages.

- **Form detection:** Web form with single general input text field is considered. A site like in [13] uses one

single generic text box for form submission. Forms with more than one binding inputs will be ignored.

- **Selection of searching keywords***:* Although the Web contains a huge amount of data, not always an optimized search result is generated for a given keyword. Here the method developed for selecting a candidate keyword for submitting a query will try to generate an optimized search result.

- **Detection of duplicate URLs:** Sometimes two different words may generate same URL twice, which will decrease the efficiency if the same URL is crawled again and again. Detection of duplicate URLs and ignoring them is another try-out of this paper work.

- **Automatic processing:** There is an automation process for crawling. That is recognizing suitable forms, generating keyword for searching, putting the word in the search bar and making or updating an index for the search results; all of these operations will be fully automatic without any human interaction.

This research work only encompasses dynamism in content, not dynamism in appearance or user interaction. For example, a page with static content, but containing client-side scripts and DHTML tags that dynamically modify the appearance and visibility of objects on the page, does not satisfy our definition as well as our objective.

Section 2 of this paper contains some aspects of dynamic web pages including the existing techniques for dynamic web page indexing. Then the proposed approach is presented in Section 3, the framework in Section 4 and its delimitations are discussed in Section 5. The conclusion in Section 6 includes some directions for future work.

2. Defining some aspects of dynamic web pages

Considering the tremendous amount of high quality content “hidden” behind search forms and stored in large searchable electronic database, “Dynamic Web Page” has become a buzzword of current web mining technology. But before continuing our research on dynamic web pages, we’ve to know what a Dynamic Web Page is. A Dynamic Web Page is a template that displays specific information in response to queries. Its *'dynamism'* lies in its resonance and interactivity, both in client-side scripting and server-side scripting. Dynamic web pages can change every time they are loaded (without anyone having to make those changes) and they can change their content based on what user does, like clicking on some text or an image.

Visitors find information in a dynamic site by using a search query. That query can either be typed into a search form by the visitor or already be coded into a link on the home page - making the link a pre-defined search of the site's catalog. In that later case, the portion of the link containing the search parameters is called a 'Query String' .This query is then used to retrieve information from the huge database which is hidden behind the search forms. This whole operation is depicted in Fig.1 below.

2.1 Problems with dynamic pages

All dynamic pages can be identified by the “?” symbol in the URLs, such as

http://www.mysite.com/products.php?id=1&style=a

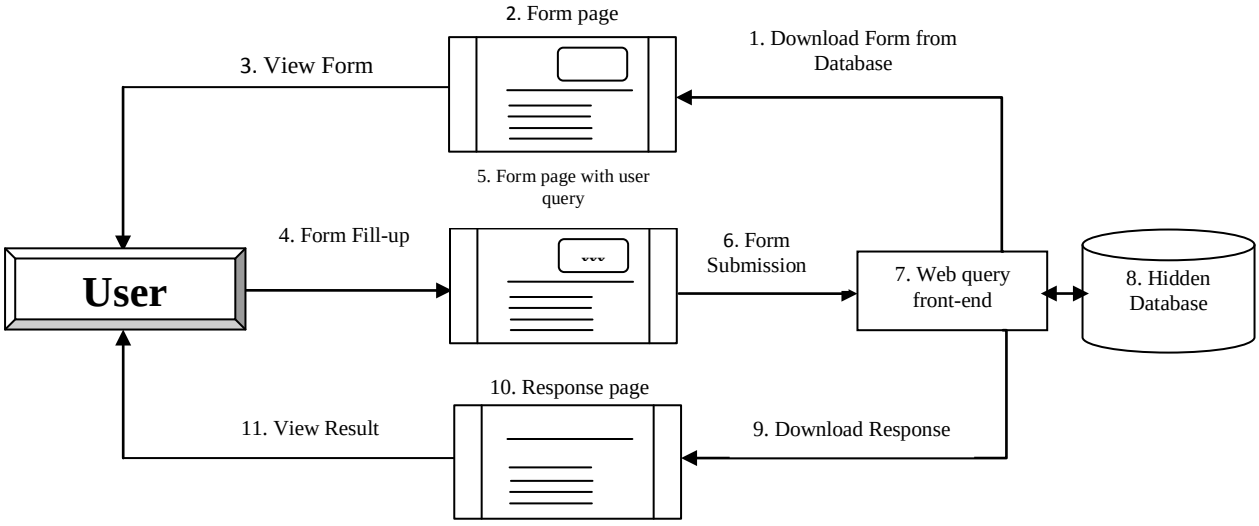

Fig.1: User interactions with search forms to retrieve information from hidden database

Search engines like Google can crawl and index dynamic pages which don't have more than 2 parameters in the URL (the example above has two parameters separated by the "&" symbol). Even so, Google may not crawl dynamic pages for various reasons. Among them, the most common are:

- **Trap:** A spider trap happens when a search engine's web crawler becomes snared in an infinite circle of a website's coding, which produce endless numbers of documents; web pages that are heavily flooded with characters, which may crash spiders programs**.**
- **Insufficient query information:** Search engine spiders have a much tougher time with dynamic sites. Some get stuck because they can't supply the information the site needs to generate the page.
- **Security and privacy issues:** Many of these sites require user authentication and bypassing it automatically may cause violation of privacy or law.
- **Policy preferences:** Some search engines deliberately avoid extensive indexing of these sites.
- **Costly:** It needs expertise in contradiction to the static one that is simple and straight forward.

Moreover, Google will not follow links that contain session IDs embedded in them as in [1].

2.2 Crawling dynamic pages

A web crawler is a relatively simple automated program, or script that methodically scans or "crawls" through Internet pages to create an index of the data it's looking for. Alternative names for a web crawler include web spider, web robot, bot, crawler, and automatic indexer.

When a search engine's web crawler visits a web page, it "reads" the visible text, the hyperlinks and the content of the various tags used in the site, such as keyword rich Meta tags. Using the information gathered from the crawler, a search engine will then determine what the site is about and index the information. 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. Web crawling is done on all text contained inside the hypertext content, tags, or text. The operation of a hidden web crawler is shown in Fig.2.

In practice common crawler algorithms must be extended to address the following issues like in [11]:

- **Speed:** In real life if some HTTP request takes one second to complete, some will take much longer or fail to respond at all. Normally a simple crawler can fetch no more than 86,400 pages per day. At this rate, it would take 634 years to crawl 20 billion pages in a single computation manner. That's why in practice, crawling is carried out using hundreds of distributed crawling machines. Now-a-days Hadoop MapReduce is used over these distributed systems to overcome the bottleneck of a single server computation with higher data processing speed.
- **Politeness:** Unless care is taken, crawler parallelism introduces the risk that a single Web server will be bombarded with requests to such an extent that it becomes overloaded. That's why crawler algorithms are designed to ensure that only one request to a server is made at a time. In order to serve this need a politeness delay is inserted between requests.

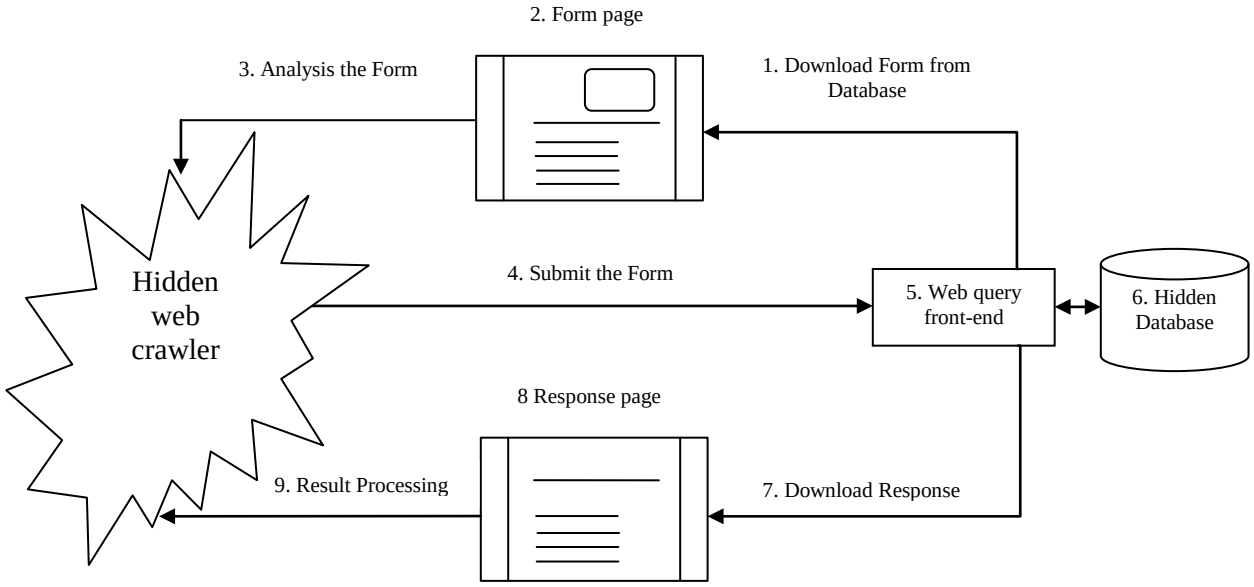

Fig.2: Hidden web crawler

- **Excluded content:** Before fetching a page from a site, a crawler must fetch that site's robots.txt file to determine whether the webmaster has specified to crawl some or the entire site.

- **Duplicate content:** Identical content is frequently published at multiple URLs. But when the page includes its own URL, a visitor counter or a date; more sophisticated fingerprinting methods are needed. Crawlers can save considerable resources by recognizing and eliminating duplication as early as possible because unrecognized duplicates can contain relative links to whole families of other duplicate content.

- **Continuous crawling:** Carrying out a full crawling with fixed intervals would imply slow response to important changes in the Web. For example, submitting the query "current time New York" to the GYM (Google, Yahoo, Microsoft) engines reveals that each of these engines crawls the ***www.timeanddate.com/worldclock*** site every couple of days. However, no matter how often the search engine crawls this site, the search result will always show the wrong time. That's why continuous crawling without any certainty or limit is avoided in most of the current search engines

2.3 Indexing dynamic pages

Indexes are data structures permitting rapid identification of which crawled pages contain like particular words or phrases. To index a set of web documents with the words they contain, we need to have all documents available for processing in a local repository. Creating the index by accessing the documents directly on the Web is impractical for a number of reasons. Collecting "all" web documents can be done by browsing the Web systematically and exhaustively and storing all visited pages. This is done by a crawler and is used by search engines. A similar operation is depicted in Fig.3.

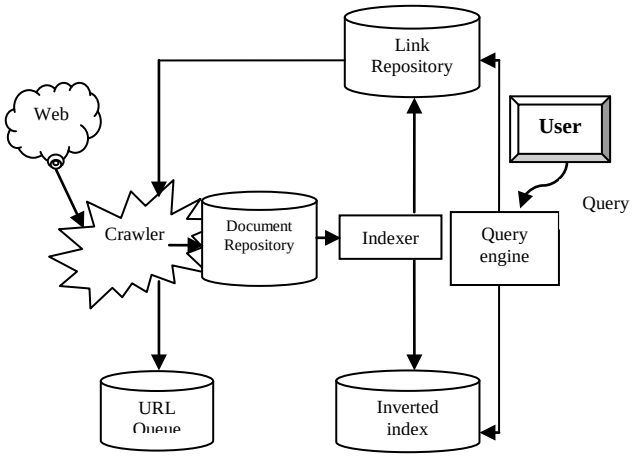

Fig.3: Crawler used in search engines

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. Search engine architectures vary in the way indexing is performed and in methods of index storage to meet the various design factors. Types of indices data structures include:

A. Inverted indices
B. Forward indices

A. 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. TABLE 1 is a simplified illustration of an inverted index.

B. Forward Index

The forward index stores a list of words for each document. TABLE 2 is a simplified form of the forward index.

TABLE 1: Inverted index of words w.r.t. their URLs

Inverted Index Word	***URL***
the	*URL 1, URL 3, URL 4, URL 5*
cow	*URL 2, URL 3, URL 4*
says	*URL 5*
moo	*URL 7*

Table 2: Forward index of words contained in URLs

Forward Index URL	***Words***
URL 1	*the, cow, says, moo*
URL 2	*the, cat, and, the, hat*
URL 3	*the, dish, ran, away, with, the, fork*

2.4 Existing techniques for dynamic web page indexing

One of the deep web crawler architecture is proposed in [2] where a task-specific, human-assisted approach is used for crawling the hidden web. There are two basic problems related to deep web search,

- **Firstly** the volume of the hidden web is very large and
- **Secondly** there is a need of such type of crawlers which can handle search interfaces efficiently, which are designed mainly for humans.

In this paper a model of task specific human assisted web crawler is designed and realized in HiWE (hidden web exposure). The HiWE prototype built at Stanford which crawl the dynamic pages is designed to automatically process, analyze, and submit forms, using an internal model of forms and form submissions. HiWE uses a layout-based information extraction technique to process and extract useful information. The advantages of HiWE architecture is that its application/task specific approach allows the crawler to concentrate on relevant pages only and with the human assisted approach automatic form filling can be done. Limitations of this architecture are that it is not precise with response to partially filled forms and it is not able to identify and respond to simple dependency between form elements. Recently [5] studied the problem of automating the retrieval of data hidden behind simple search interfaces that accept keyword-based queries but did not focus on the detection of search interfaces.

A technical analysis of some of the important deep web search interface detection techniques is done to find out their relative strengths and limitations with reference to current development in the field of deep web information retrieval technology [3] .We found this analysis crucial for the detection of the search interface and it can be a good starting ground for anyone interested in this field. Reference [4] proposed some ways to select keywords for query such random, generic-frequency and adaptive. Meanwhile other usual approach to dynamic indexing is to remove query strings from dynamic URL's, adding dynamic links to static pages, making dynamic links look like static using mod_rewrite available in web server like apache. We also have some paid inclusion programs ,these programs, are premium services for indexing dynamic sites include those of AltaVista, *Inktomi* and *FAST*, to name a few . Also there are Deep web search tools enhance deep Web searching, including *BrightPlanet*, *Intelliseek's Invisible Web*, *ProFusion*, *Quigo*, *Search.com*, and *Vivisimo*. Irregularities highlighted in the existing techniques have leaded us to the proposal of the following approach.

3. Proposed approach

In order to index the dynamic contents hidden behind the search forms, we've come up with an approach which contains the following steps:

3.1 Web pages collection
3.2 Form interface recognition
3.3 Keyword selection
3.4 Query submission
3.5 Result Processing
3.6 Updating the index

3.1 Web pages collection

This part is essentially a static crawling, given the initial URL, the crawler recursively fetches all pages that are linked by it (don't make it recursive, unless you are using functional languages; just use a queue of URLs to be fetched). Test it on the set of web pages created at the beginning. If a page is linked many times, it must be downloaded once. Static crawling is depicted in Fig.4

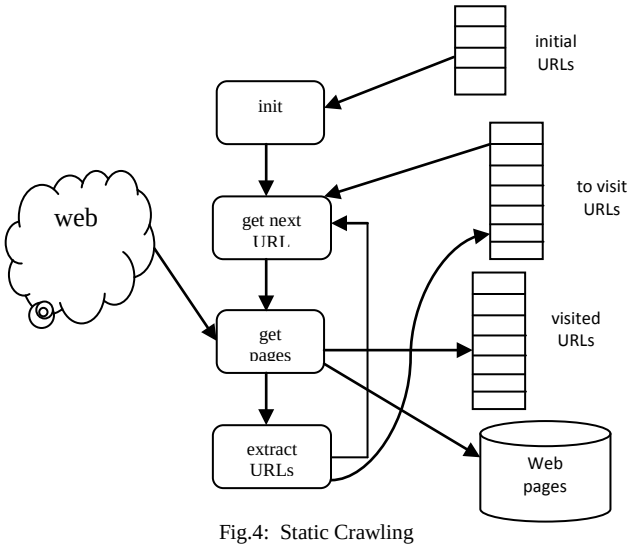

Fig.4: Static Crawling

```
<form action="MAILTO:someone@example.com" method="post"
enctype="text/plain">
Name:<br /><input type="text" name="name" value="your name" /><br />
E-mail:<br /><input type="text" name="mail" value="your email" /><br />
Comment:<br /><input type="text" name="comment" value="your comment"
size="50" />
<br /><br />
<select name="cars">
<option value="volvo">Volvo</option><option value="saab">Saab</option>
<option value="fiat">Fiat</option><option
value="audi">Audi</option></select>
<br /><br /><br /><br /><br /><br />
<input type="submit" value="Send"><input type="reset" value="Reset">
</form>
```

Fig.5: HTML form tag markup for sample Form input controls

3.2 Form interface recognition

Recognizing a web form and its fields is a key point of this approach. A standard HTML web form consists of form tags [6], a start tag <form> and an end tag </form> within which the form fields reside. Forms can have several 'input controls', each defined by an <input> tag and some values considered as domain for those input controls. Input controls can be of a number of types, the prominent ones 'text boxes', 'check boxes', 'selection lists' and 'buttons' (submit, reset, normal or radio).Each of the field is having attributes like label, name and values. The form tag also has attributes like 'method' i.e.: 'get' and 'post' and 'action' which identify the server that will perform the query processing in response to the form submission.

In this study, we focus on the forms with one input control binding to a generic text box. Forms with multiple input controls will be ignored. Fig.6 shows such a form with several input controls and Fig.5 shows the piece of HTML markup that was used to generate this form. Whenever our crawler will encounter forms like this they will be discarded from the crawling operation.

But if the crawler encounters a form tag like depicted in Fig.7 it will consider the form as eligible for crawling and will proceed with its operation. A general search form with single input, often on the top-right of the web page is used in this approach.

Further, as per the HTML specification, forms using ***post*** method for form submission are used whenever submission of the form results in state changes or side effects (e.g. for shopping carts, travel reservation and login). For these reasons we restrict our attention to those forms which are using ***get*** method to submit the form as they tend to produce contents suitable for indexing.

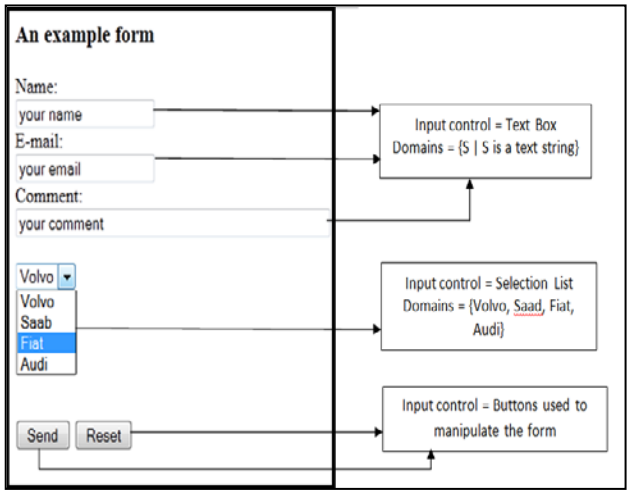

Fig.6: Simple labeled form with several control inputs

*<**form** action=http://www.iut.com/department*
method="get">
<input name=keyword type=text/>
Input name=searching type=submit value=search/>
*</**form**>*

Fig.7 HTML form tag markup for a considerable sample form

3.3 Keyword selection

The selection of the most appropriate and relevant value for the input field that can maximize the search is challenging, even though the generic text field generally can accept any keyword. How should a crawler select the queries to issue, given that the goal is to download the maximum number of unique documents from a textual database? Finding the answer for this question is another approach that we've tried to cover here.

In order to solve this question we could select our initial keyword from a dictionary and use it for query submission. But generating a dictionary and searching a keyword within it will be both time and space consuming. Since we want to cover all possible languages, we can't start with from a dictionary of terms. After all an English dictionary will never contain a word which can be used as a keyword for query submission in a Chinese search form. In this case our approach suggests the following aspects:

- **Initial value:** At the very first, keywords are selected from the static content of the web page having the search form interface.
- **Adaptive:** After the generation of the 1st set of results, promising keywords are selected from the successfully retrieved pages. Here keywords for query submission in a search form are selected adaptively from itself.
- **Multilingualism:** By selecting the searching keywords from the web page instead of a predefined location like dictionary or repository our approach also supports multilingualism.

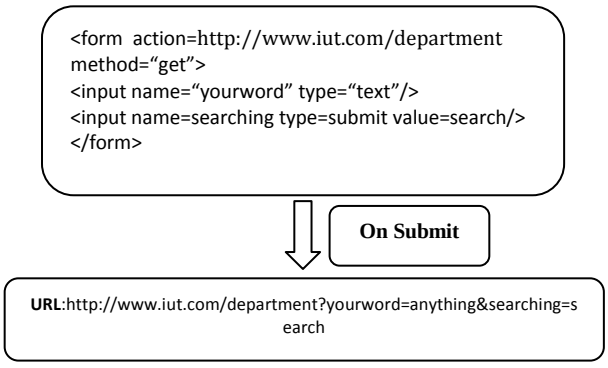

Fig.8: Dynamic URL on querysubmission

- **Limit:** At most **max** submissions per form will take place to prevent the crawler from falling in a trap (infinite loop). Where *max* is a given number representing the maximum number of queries.

The priority in keyword selection is calculated based on the term frequency F_{tf} in our approach as it determines the importance of a word in a set of documents. The term frequency F_{tf} is a measure of how often a word is found in a collection of documents. Suppose a word 'W_i' occurs 'n_p' times within a web page 'P' and there are total of Np words (including the repeated ones) on that page. Then the term frequency,

$$F_{tf} = n_p / Np. \quad (1)$$

But if we fail to obtain a convenient keyword in that given page, the choice is taken in the repository or at last in the worst case from the dictionary. The selected keywords destined to the query should be compared against the stop words list as these words used to be more frequent.

3.4 Query submission

After selecting the keyword, there is another challenge in submitting the query in the search form automatically, i.e. without any human interaction. Whenever a keyword for a query submission will be selected it'd automatically be submitted in the search form to generate more search results. The action would be something similar depicted in Fig.8.

In this way, whenever a form is submitted a dynamic URL is generated and sent to the database. How many time the query should be submitted and when should it stop? Of course it shall stop when *max* numbers of queries have been submitted.

3.5 Result processing

When our crawler submits a form for processing, different results are possible.

1) The returned page will contain all the data behind the form.

2) The returned page may contain data, but not showing all the data for the query in a single page. Instead, there may be a "next" button leading to another page of data, such as the. In this case, the system will automatically gather all the data on all "next" pages (actually not all, up to a certain limit to avoid a Trap) into a single query result.

3) The query might return data, but only part of the data behind the form because the query is just one of many possible combinations of the form fields. In this case the only returned portion will be processed.

4) The query may return a page that not only contains data, but also contains the original form. Here whatever the result is generated we'll gather information as much as possible.

5) The query may return a page that has another different form to fill in. For this case we'll start with the resultant form from the beginning.

6) Some other error cases might involve a server being down, an unexpected failure of a network connection, or some other HTTP errors.

7) The query may go and return the same form requesting for required field to be filled or to be filled with consistent data. Usually this kind of form contains JavaScript.

8) Successive queries may return redundant result, it is therefore important for similarity detection be verified amount successive queries. After all this, the result should be crawled and indexed.

3.6 Updating the index

After the processing the result an initial index will be created. But as this is a continuous process more and more pages will be crawled to extract more words and will be added to the index in times. As a result a continuous updating of the index is required here which will eventually exceed the capacity of a normal single storage device. That's why multiple storage device is needed and in order to do this we've used "Hadoop-MapReduce" to do the job. Hadoop is an Apache software foundation project as in [20]. It's a scalable, fault-tolerant system for data storage and processing and a framework for running applications on large clusters. Hadoop includes:

- *HDFS* - a distributed file system and
- *Map/Reduce* - offline computing engine.

HDFS splits user data across servers in a cluster. It uses replication to ensure that even multiple node failures will not cause data loss. HDFS breaks incoming files into blocks and stores them redundantly across the cluster. In this approach we're using this splitting and reducing technique to handle the huge amount of index.

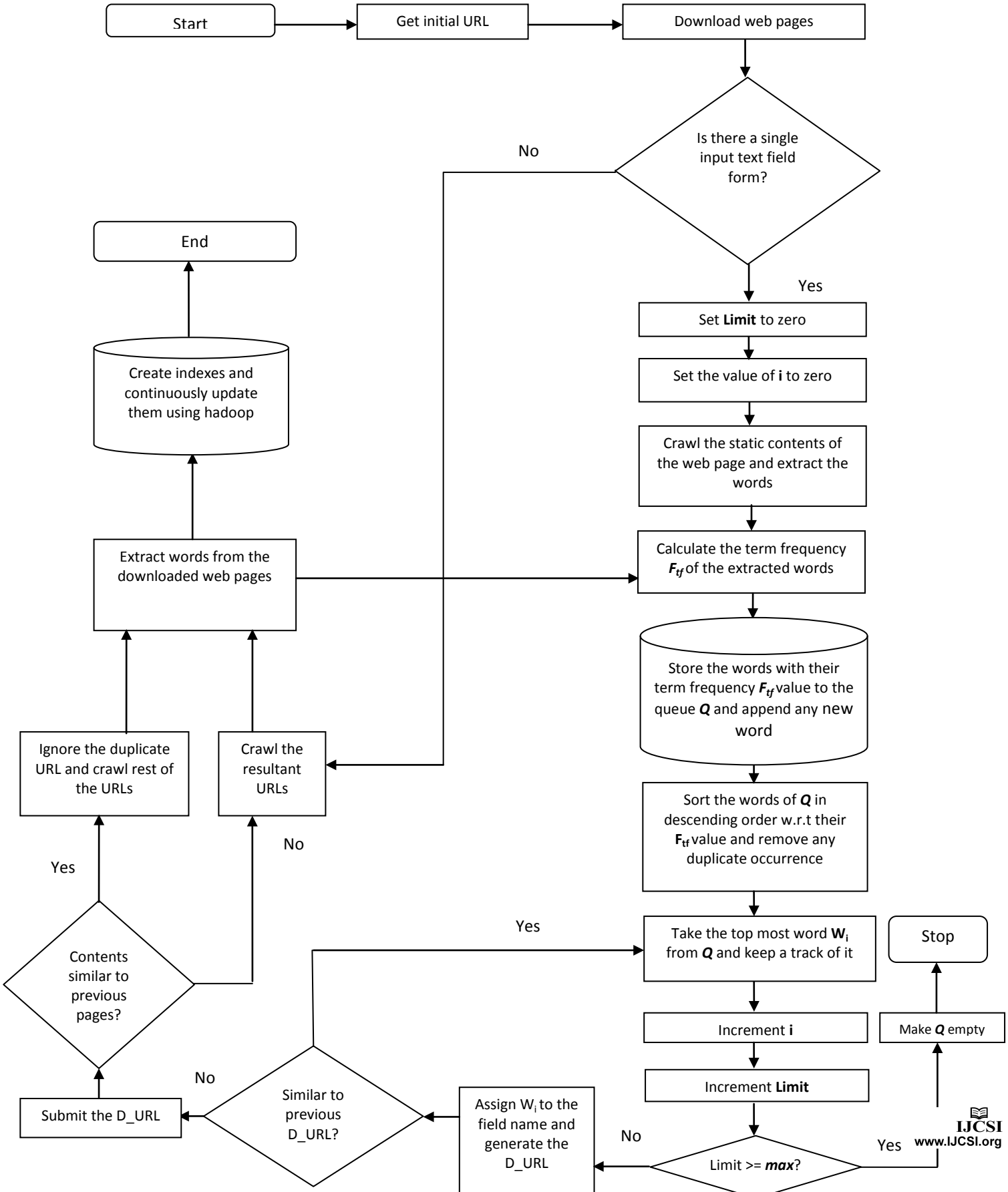
Start
Get initial URL
Download web pages
Is there a single input text field form?
No
Yes
End
Set Limit to zero
Create indexes and continuously update them using hadoop
Set the value of i to zero
Crawl the static contents of the web page and extract the words
Extract words from the downloaded web pages
Calculate the term frequency F_{tf} of the extracted words
Store the words with their term frequency F_{tf} value to the queue Q and append any new word
Ignore the duplicate URL and crawl rest of the URLs
Crawl the resultant URLs
Sort the words of Q in descending order w.r.t their F_{tf} value and remove any duplicate occurrence
Yes
No
Contents similar to previous pages?
Take the top most word W_i from Q and keep a track of it
Stop
Increment i
Make Q empty
Increment Limit
No
Similar to previous D_URL?
Assign W_i to the field name and generate the D_URL
Submit the D_URL
No
Yes
Limit >= max?
IJCSI
www.IJCSI.org

Table 3: legend of the framework

Notations	***Meaning***
Q	*Contains crawled words*
F_{tf}	*Term frequency*
max	*Maximum number of submission=10*
W_i	i^{th} *word*
Limit	*Current number of submission*
i	*Word index*
D_URL	*Dynamic URL*

4. General Framework

According to Fig.9, the web crawler starts with an initial URL to download the page. After that the downloaded web page is crawled and is checked to see if it contains a single input text field for query submission or not. If not, it will simply be crawled and the words and URLs are indexed. If yes, the *limit* and variable 'i' is used respectively to count the number of submissions and searching keywords. After that this web page containing the form is crawled and the term frequency F_{tf} of words extracted from it are calculated and stored in a queue Q. Words in the queue are sorted in descending order of their F_{tf} and the duplicates are removed. The top most word from Q is submitted to the form through a dynamic URL. This process is repeated till the limit reaches the maximum limit *max* when Q is emptied and the crawler stops. Contents of web pages retrieved are detected to see if they are similar to previous ones and the duplicates are deleted, duplicates URLs can be filter as in [18]. These new web pages are then crawled again, sent to the index for updating and term frequency is calculated and submitted as previously until a certain amount of web pages are downloaded. More details can be observed from the Fig.9 and TABLE 3.

5. Delimitations of our approach

In our approach we are not concerned with the following aspects:

1) **Dealing with the form unless it is in the standard format:** If the code is not properly written in a suitable form, our parser will not be able to conveniently extract information from the web page containing that form. Therefore the presence of the form may not be detected.

2) **Handling form that doesn't support passing parameters via URL:** As in [12] the get method append its parameters to the action in the URLs in the form of a dynamic URLs format that are often clearly visible (e.g. http://jobs.com/find?src=bd&s=go). In contrast the post method parameters are sent in the body of the HTTP request and its URL is just simple making it difficult for us to deal with it (e.g., http://jobs.com/find).

3) **Forms with multiple elements:** Because we're focusing in only single input form, any form other than this kind will not be considered for submission.

4) **Forms that span across several pages:** This is the case where the same form is extended over multiple continuous pages.

5) **Forms with JavaScript embedded:** Usually input fields of this type of form have a lot of restriction such the type of input, the format, the length, the syntax. Because we are not going to handle all these, we just prefer to ignore them and discard the form.

6) **Forms that a single input is not a text field:** The single input under consideration must be a text field type

7) **Forms with personal information** indication such as username, password, E-mail will not be considered for privacy raison.

6. Conclusion

In this paper we have studied how to use a hidden web crawler as an approach to dynamic web indexing. We have proposed a complete and automated framework that may be quite effective in practice, leading to an efficient and higher coverage. We have tried to make our design as simple with fewer complexities as possible. Towards the achievement of our goal, we've already developed a prototype for dynamic web page indexing using java, and the website used is [13]. Our future work will include a complete implementation, evaluation and analysis of this approach. We'll also try to compare the performance in both java platform and Hadoop MapReduce.

References

[1] Dan Sisson. *Google SEO secrets, the complete guide*, pp.26–28, 2006.

[2] S. Raghavan, H. Garcia-Molina. Crawling the Hidden Web, in: Proc. of the 27th Int. Conf. on Very Large Databases (VLDB 2001), September 2001.

[3] Dilip Kumar Sharmal, A.k.Sharma2.Analysis of techniques for detection of web search interfaces, *2YMCA University of Science and Technology, Faridabad, Haryana,*

India,http://www.csi-india.org/web/csi/studentskorner-december10, *accessed on June*, 2011.

[4] A.Ntoulas, Petros Zerfos, Junghoo Cho, Downloading Textual Hidden Web Content through Keyword Queries, JCDL '05. Proceedings of the 5th ACM/IEEE-CS Joint Conference, 2005.

[5] Luciano Barbosa, Juliano Freire, siphoning hidden-web data through keyword-based interfaces, Journal of Information and Data management, 2010.

[6] http://www.w3schools.com/html/html_forms.asp, *accessed* on, June 2011

[7] Wiley, *Data Mining the Web Uncovering Patterns.(2007)*.DDU.[0471666556].

[8] Pradeep, Shubha Singh, Abhishek, NewNet- Crawling Deep Web, IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.5, pp. 129-130, May 2010.

[9] http://www.worldwidewebsize.com/, *accessed on June*, 2010.

[10] J Bar-Ilan - Methods for comparing rankings of search engine results-2005,http://www.seo-jerusalem.com/googles-best-kept-secret/,http://www.search-marketing.info/search-algorithm/index.htm, *accessed on June*, 2010.

[11] David Hawking, Web Search Engines-1, pp. 87-88, 2006.

[12] Jayant Madhavan, David Ko, Luc jaKot, Vignesh Ganapathy, Alex Rasmussen, Alon Halevy. “Google's Deep-Web Crawl”, Proceedings of the International Conference on Very Large Databases (VLDB), 2008.

[13] http://www.dmoz.org/, *accessed on June*, 2010.

[14] Brijendra Singh, Hemant Kumar Singh."Web Data Mining Research: A Survey", IEEE, 2010.

[15] http://www.ncbi.nlm.nih.gov/pubmed, *accessed on June*, 2010.

[16] C.H.Chang, M.Kayed, M.R.Girgis, K.F.Shaalan,” A survey of web information extraction systems”. IEEE Transactions on Knowledge and Data Engineering 18(10), pp.1411–1428, 2006.

[17] P.Wu, J.R.Wen, H.Liu, W.Y.Ma,"Query selection techniques for efficient crawling of structured web sources". In: Proc. of ICDE, 2006.

[18] Wang Hui-chang, Ruan,Shu-hua, Tang,Qi-jie."The Implementation of a Web Crawler URL Filter Algorithm Based on Caching”. Second International Workshop on Computer Science and Engineering, IEEE, 2009.

[19] Jeffrey Dean, Sanjay Ghemawat ."MapReduce: Simplified Data Processing on Large Clusters". To appear in OSDI, 2004 http://labs.google.com/papers/mapreduce.html.

[20] http://hadoop.apache.org/, *accessed on june*, 2010.

[21] King-Ip Lin, Hui Chen. "Automatic Information Discovery from the “Invisible Web”", Information Technology: Coding and Computing (ITCC’02), IEEE, 2002.

Hasan Mahmud has received his Bachelor degree in Computer Science and Information Technology (CIT) from Islamic University of Technology (IUT), Bangladesh in 2004. After that he had joined as a faculty member in Computer Science and Engineering (CSE) department at Stamford University Bangladesh.

He did his Master of Science degree in Computer Science (Specialization on NetCentric Informatics) from University of Trento (UniTN), Italy in 2009. He had received University Guild Grant Scholarship for the two years (2007-2009) Master’s study and also awarded with early degree scholarship. He has 5 research papers published in different international journals. He is currently working as an Assistant Professor in the department of Computer Science and Information Technology (CIT), IUT, Bangladesh. His current research interests are on web mining, Human Computer Interaction, and Ubiquitous Computing.

Moumie Soulemane did his Higher Diploma in Computer Science and Information Technology (CIT) with specialization in web technology from the Islamic University of Technology (IUT), Bangladesh in 2010. Currently he is the final year student for the B.Sc. degree in Computer Science and Information Technology (CIT) at the same university.

Mohammad Rafiuzzaman is currently in the final year of bachelor degree in Computer Science and Information Technology (CIT) at the Islamic University of Technology.