International Journal of Information Technology and Knowledge Management
July-December 2011, Volume 4, No. 2, pp. 673-678

OPTIMIZED METHOD FOR INDEXING THE HIDDEN WEB DATA

Priyanka Gupta[1], Komal Bhatia[2] & Kalpna Gupta[3]

Published Methods of Indexing the hidden web database use different indexing techniques like distributed indexing and noble indexing techniques. The goal of this research is to extract the data from various hidden web databases and this data in integrated form will be stored in large repository with no duplicate records. Here, we propose an optimized method for indexing the hidden web database. This research uses Map-Reduce Framework for indexing the Data downloaded by the Siphone++. Basically, the idea behind using this Map-Reduce framework as Indexer is to make strong links between the WebPages using clustering of nodes. It is a highly scalable technique. Here, with minimum number of queries we get the maximum hidden web data. The beauty of Map Reduce system is that it stores intermediate results as a separate list. Thus while processing complete result, this intermediate result is used which reduces the complexity and saves time. User can find the desired information by submitting the queries to the search interfaces. Then Search engine fetches the indexed data stored in Highly Distributed File System.

1.1 Proposed Work

In this Thesis, we propose an optimized method for indexing the hidden web database. The basic model is depicted in Figure 1.1. Section 1.2. discusses first component Siphone++ as a hidden web crawler. It is the advance version of Siphone with new framework for Indexing. Section 1.2.1 discusses the Adaptive component of Siphone++. Section 1.2.2 outlines the Heuristic approach of Siphone++ to siphon the data behind the Search Interfaces. It issues the queries that have high coverage of data from hidden database.

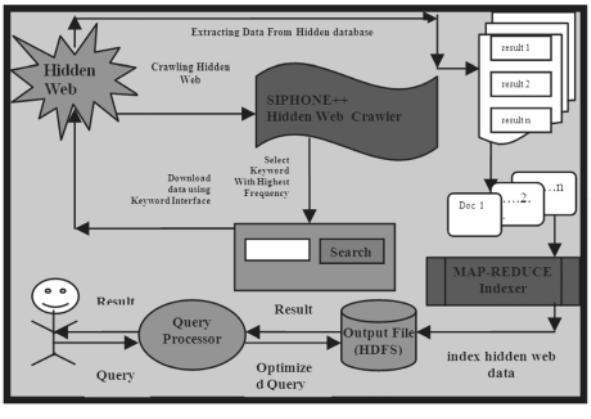

Fig. 1.1: Proposed Optimized Model for Indexing

Section 1.3 proposes Map-Reduce Indexer for indexing the hidden web Data. Map Reduce Indexer uses an algorithm which is used to make search among clusters of machines which Google uses to fetch data for web searches. Its main aim is processing and generating large data. The primary job of Map Reduce is to select the node on cluster, where the jobs need to run and how balancing is to be done. Here, with minimum number of queries we get the maximum hidden web data. Map-Reduce Indexer uses an algorithm that is used by Google's web servers to process large amount of data and index it so that it can be referred in web searches. User can find the desired information by submitting SQL queries to fetch data from hidden sources and send it back to user with desired results.

1.2 Using SIPHONE++ as Hidden Web Crawler

Reference [1] presents a crawler for automatically retrieving Web content hidden behind keyword-based interfaces. Being able to retrieve and index, this content has the potential to uncover hidden information and help users find useful information that is currently out-of-reach for search engines. Unlike multi-attribute forms, keyword-based interfaces are simple to query, since they do not require detailed knowledge of the schema or structure of the underlying data.

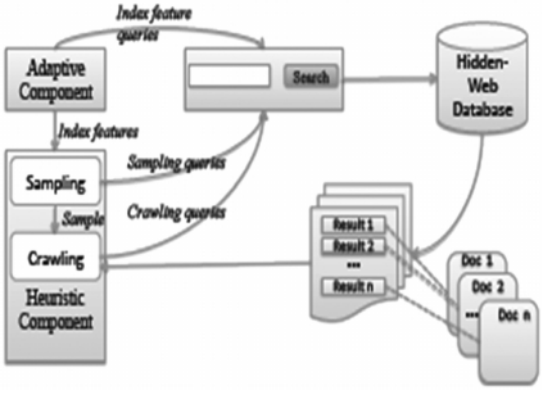

Fig. 1.2: Architecture of the Siphon++

[1]Department of Computer Science, Ajay Kumar Garg Engineering College and Gautama Budh, Technical University, Ghaziabad
[2]Department of Computer Science, YMCA Institute of Engineering Faridabad
[3]Department of Computer Science, G. L. Bajaj Engineering College, Gautama Budh, Technical University, Greater Noida
E-mail: fascinatingpriya@gmail.com

It is thus possible to create automatic and effective solutions to crawl these interfaces. We propose a new crawling strategy which adapts to the features of the indexes underlying the search interface.

1.2.1 Adaptive Component

The Adaptive Component (AC) detects the index features by issuing probe queries against the search interface. We call these queries index feature queries (see Figure 1.2). Two techniques commonly used for compacting search engine indexes are: stop word removal (e.g., the removal of prepositions and articles); and stemming, i.e., reducing words to their root (stem) form. As the Heuristic Component (HC) searches for high-coverage queries, if stop words are present in the index, they should be included in the queries since they are likely to appear in many documents in the database. On the other hand, if they have been removed from the index, they should be ignored, since they will lead to queries that have no results.

1.2.2 The Heuristic Component

The Heuristic Component is responsible for defining a policy for submitting queries. The HC first builds a sample of the database by issuing a set of queries (Sampling phase). Next, it selects most frequent words in the documents in this sample to crawl the database (Crawling phase), assuming they also have a high frequency in the actual database/ index.

Sampling

The goal of the Sampling phase is to assemble a representative sample of the database. The performance of Siphon++ heavily depends on the quality of the sample. The only way to acquire this information is through the keyword search interface that serves as the entry point to the database. Reference[2] developed sampling-based algorithm that automatically discovers keywords which result in high recall; and use these keywords to build queries that siphon all available results in the database (or, as many as possible).

Using Sampling to Retrieve Hidden Data

In order to automatically retrieve data (results) through a keyword-based interface, it is necessary to determine which queries to issue and which keywords to use. The ideal approach would be to determine a small number of queries that retrieve all the results. Instead of blindly issuing queries, Barbosa and Freire propose a sampling-based approach to discover words that result in high coverage [2].

Siphoning Hidden-Web Data through
Keyword-Based Interfaces
Algorithm 1 SampleKeywords(URL,form)

```
1:  page = get(URL) ;
2:  //retrieve the first set of results
3:  initialKeywordList=generateList(page) ;
4: word=nextHigherOccurrency (initialKeyword List) ;
5: resultPage=submitQuery(form,word);
6: while resultPage==errorPage do
7: word=nextHigherOccurrency(initialKeyword List) ;
8: resultPage=submitQuery(form, word) ;
9: end while
10: //initialize keyword list
11: acceptStopword=checkStopword(form) ;
12: if acceptStopword then
13: keywordList=generateListWithStopWords
    (resultPage);
14: else
15: keywordList=generateList(resultPage) ;
16: end if
17: //iterate and build list of candidate keyword/
    occurences
18: numSubs = 0 ;
19: while numSubs < maxSubs do
20: // randomly selects a word
21: word = selectWord(keywordList) ;
22: resultPage = submitQuery(form, word) ;
23: //adds new keywords, and updates the frequency of
    existing keywords
24: if acceptStopword then
25: keywordList+ = genListWithStopWords (result
    Page) ;
26: else
27: keywordList += genList(resultPage) ;
28: end if
29: numSubs++ ;
30: end while
31: return keywordList ;
```

As described in Algorithm 1, probe queries are issued to learn new keywords from the contents of the query results and their relative frequency with respect to all results retrieved in this phase. The first step is to retrieve the page where the search form is located, select and submit a keyword (lines 1–16). Once a 'results page' is successfully retrieved, the algorithm builds a list of candidate keywords by iteratively submitting a query using a selected word (line 22); and using the results to insert new high-frequency words into the candidate set, as well as to update the frequencies of existing keywords (lines 23–28).

Crawling

After sampling, it selects most frequent words in documents in this sample to crawl the database, assuming they also have a high frequency in the actual database/index. The Crawling phase is responsible for

(1) issuing queries to the database;

(2) retrieving the result pages and extracting from them the links to the target documents; and

(3) downloading the documents from the database.

Algorithm 2

```
ConstructQuery(keywordList,form)
    1: numSubs=num
    Words=totalBefore =
    totalCurrent = 0 ;
    2: query = queryTemp = null; 3 :
    while (numSubs < maxSubs) &&
    (numWords < maxTerms) do
    4: //selects word with
    highestfrequency
    5: word=nextElement(listOfOccurrency) ;
    6: queryTemp =
    addWordToQuery(query, word) ;
    7: page=submit (form, queryTemp) ;
    8: totalCurrent=getNumberOfResults(page, form) ;
    9: if(totalBefore *
    minimumIncrease)<=
    totalCurrent then
    10: query = queryTemp ;
    11: totalBefore = totalCurrent ;
    12: numberWords++ ;
    13: end if
    14: numSubs++ ;
    15:end while
```

The candidate (high-frequency) keywords are input to Algorithm 2, which uses a greedy strategy to construct the query with the highest coverage. It iteratively selects the keyword with highest frequency from the candidate set, and adds it to the query if it leads to an increase in coverage. Note that although the query construction phase can be costly, once a high-coverage query is determined, it can be re-used for later searches, e.g., a hidden-Web search engine can use the query to refresh its index periodically.

1.3 Using Map-Rezduce Indexer for Indexing

The current indexing approaches such as CLucene and single-pass based indexing use inefficient data-structures for merging two index segments (segment represents inverted index for a subset of documents in a collection). The data structures used in such approaches require two expensive operations to merge between two index segments, namely: merge-sort of terms and data re-organization of document and postings data (list of positions for the occurrence of a term in a document). Efficient parallel merge sort can help here but expensive data reorganization is unavoidable. Since these data-structures lead to expensive merge they are not suitable for real-time indexing with high throughput. Expensive merge leads to load-imbalance and hence poor scalability in distributed indexing algorithm that uses separate nodes to generate segments and to merge them. Hence, one needs to re-design the index data-structures and ensure they enable an efficient merge. For distributed indexing, the algorithm should lead to scalable indexing throughput and low indexing latency, while at the same time sustaining search response time and throughput. This is a challenging problem due to inherent trade-offs between index size, indexing throughput and search response time and throughput.

We focus on indexing throughput maximization with same or better search performance and around two times the index size of typical indexing approaches. We present the design of innovative distributed data-structures and algorithm that meet the design goals by enabling efficient index merge that leads to improved load balance and hence better scalability.

It is known that for a single cluster, search is not scalable when performed over more than certain number of nodes in a system depending on workload and system configuration. MapReduce is a framework for processing huge datasets on certain kinds of distributable problems using a large number of computers (nodes), collectively referred to as a cluster (if all nodes use the same hardware) or as a grid (if the nodes use different hardware). Computational processing can occur on data stored either in a file system (unstructured) or within a database (structured). The advantage of Map Reduce is that it allows for distributed processing of the map and reduction operations.

1.3.1 Logical View

The Map and Reduce functions of Map Reduce are both defined with respect to data structured in (key, value) pairs. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$\text{Map}\ (k_1, v_1) \rightarrow \text{list}\ (k_2, v_2)$$

The Map function is applied in parallel to every item in the input dataset. This produces a list of (k_2, v_2) pairs for

each call. After that, the Map Reduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys.

The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce (k_2, list (v_2)) $\rightarrow$ list (v_3)

Each Reduce call typically produces either one value v_3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired 'result list'.

Example

The canonical example application of Map Reduce is a process to count the appearances of each different word in a set of documents:

```
void map(String name, String
document):
    // name: document name
    // document: document
contents
    for each word w in document:
        EmitIntermediate(w, "1") ;
void reduce(String word,
Iterator partialCounts) :
    // word: a word
    // partialCounts: a list of
aggregated
partial counts
    int result = 0 ;
    for each pc in
partialCounts :
    result += ParseInt(pc) ;
    Emit(AsString(result)) ;
```

Here, each document is split into words, and each word is counted initially with a "1" value by the Map function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to Reduce, thus this function just needs to sum all of its input values to find the total appearances of that word.

1.3.2 Design of MAP-REDUCE Indexer

Software architect, Ricky Ho presents model on Map-Reduce Indexing as shown in Figure 1.3 [3]. This model provides an overview of previous efforts in distributed and real-time text indexing and explains a generalized Map-Reduce framework. Our scalable approach in memory indexing techniques can be implemented as part of the run-time of this framework, which improves the performance of this system for large-scale indexing. This provides scalable distributed indexing algorithm without trade-off on search time.

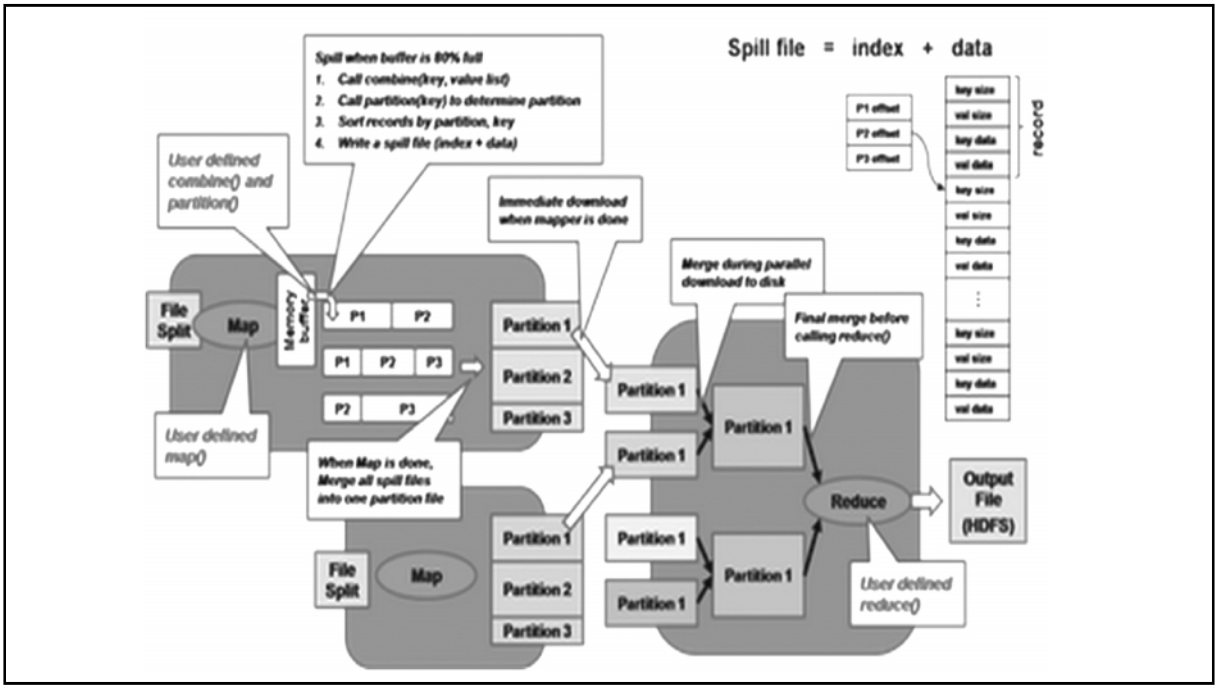

Fig. 1.3: Map Reduce Framework

Abstract Processing Model: The general processing flow is as follows.

1. Input data is "split" into multiple mapper process which executes in parallel.
2. The result of the mapper is partitioned by key and locally sorted.
3. Result of mapper of the same key will land on the same reducer and gets consolidated there.
4. Merge sort happens at the reducer, so all keys arriving the at the same reducer are sorted.

Within the processing flow, user-defined functions can be plugged-in to the framework. It requires the following sequence.

- map(key1, value1) -> emit(key2, value2)
- reduce(key2, value2_list) -> emit(key2, aggregated_value2)
- combine(key2, value2_list) -> emit(key2, combined_value2)
- partition(key2) return reducer number.

The Mapper will parse the words in the document to emit [word, doc] pairs along with other metadata such as the place, where in the document, this word occurs. In Text index, we are not just counting the actual frequency of the terms but also adjust its weighting based on frequency distribution, Therefore common words will have less significance when they appear in the document.

1.5. Query Using SQL Model

The SQL model can be used to extract data from the data source. It contains a number of primitives.

Projection / Filter

This logic is typically implemented in the Mapper

1. result = SELECT c1, c2, c3, c4 FROM source WHERE conditions

Aggregation / Group by / Having

This logic is typically implemented in the Reducer

2. SELECT sum (c3) as s1, avg(c4) as s2 ... FROM result GROUP BY c1, c2 HAVING conditions.

Data Joins: Joining 2 data sets is a common operation in Relational Data Model and has been quite prevalent in RDBMS implementation.

Indexed join: An index (e.g. B-Tree index) is built for one of the data sets (say data set 2 which is the smaller one). This process will scan through data set1 and lookup the index to find the matched records of data set 2.

Merge join: It pre-sorts both data sets so that they are arranged physically in increasing order. This step is realized by just merging the two data sets: (a) Locate the first record in both data set 1 and set 2, which is their corresponding minimum key (b) In the one with a smaller minimum key (say data set 1), keep scanning until the next key is found which is bigger than the minimum key of the other data set (i.e. data set 2). Call this the next minimum key of data set 1. (c) Switch position and repeat the whole process until one of the data sets is exhausted.

Hash / Partition join: Partition the data set1 and data set 2 into smaller size and apply other join algorithm in a smaller data set size. A linear scan with a hash() function is typically performed to partition the data sets such that data in set1 and data in set 2 with the same key will land on the same partition.

Semi join: This is mainly used to join two sets of data that are stored at different locations. The goal is to reduce the amount of data-transfer such that only the full records appear in the final joint result. (a) Data set 2 will send its key set to machine holding Data set 1. (b) Machine holding Data set 1 will do a join and send back the records in Data set 1 that matches one of the send-over keys. (c) The machine holding data set 2 will do a final join to the data send back.

General reducer-side join

This is the most basic one. It records from data set 1 and set 2 with the same key. This will land on the same reducer, which will then do a cartesian product. The downside of this model is that the reducer needs to have enough memory to hold all records of each key.

1.7 Conclusion and Future Scope

In this paper we examined the problem of siphoning data hidden behind keyword-based search interfaces. We index the downloaded data by using Map-Reduce Indexer. We proposed a simple and completely automated strategy that can be quite effective in practice, leading to high coverages of data. In order to provide a comprehensive solution to the problem, techniques are needed to automatically obtain this information and dynamically tune the algorithm.

Map Reduce is useful in a wide range of applications and architectures including: "distributed grep, distributed sort, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning, translation." Most significantly, when Map Reduce was finished, it was used to completely regenerate Google's index of the World Wide Web, and replaced the old ad hoc programs that updated the index and ran various analyses. MapReduce's stable inputs and outputs are usually stored in a distributed file system.

The fact that such a simple strategy is effective raises some security issues. As people publish data on the Web, they may be unaware of how much access is actually provided to their data. Our preliminary study suggests some simple guidelines to be followed to make it harder for information to be hijacked from search interfaces. For example one must avoid indexing stopwords.

An interesting direction for future work is to characterize search interfaces in a better way, and devise techniques that guarantee different notions and levels of security. As part of the future work, we plan to study efficient security algorithm for keyword interfaces.

REFERENCES

[1] Karane Vieira, Luciano Barbosa, Juliana Freire, Altigran Silva, Siphon++: A Hidden-Web Crawler for Keyword-Based Interfaces. Proc.ACM, California, USA,pp.978-991, October 2010.

[2] Luciano Barbosa, Juliana Freire, "Siphoning Hidden-Web Data through Keyword-Based Interfaces", Proc. of Journal of Information and Data Management, pp 133-144, February 2010.

[3] http://4.bp.blogspot.com

[4] Wu, W., Doan, A., Yu, C., and Meng, "W. Modeling and Extracting Deep-Web Query Interfaces", Advances in Information and Intelligent Systems. Springer Berlin, Heidelberg, pp 65-90., 2009.

[5] Wu, W., Yu, C., Doan, A., and Meng, W. 2004, "An Interactive Clustering-based Approach to Integrating Source Query Interfaces on the Deep Web", Proc. of the ACM International Conference on Management of Data Paris, France, pp.13-18, June 2004.

[6] Zhang, Z., He, B., and Chang, K. C. 2004, "Understanding Web Query Interfaces: Best-Effort Parsing with Hidden Syntax", In Proc. of the ACM International Conference on Management of Data Paris, France, pp.107 – 118, June 2004.

[7] Anupam, V., J. Freire, Kumar, B., and Lieuwen, D, "AutomatingWeb navigation with theWebVCR", Proceedings of the International World Wide Web Conferences. Amsterdam, The Netherlands, pp. 503–517, 2000.

[8] Bhalotia, G., Hulgeri, A., Nakhe, C, Chakrabarti, S., and Sudarshan, "S. Keyword Searching and Browsing in Databases Using BANKS", Proceedings of the International Conference on Data Engineering. San Jose, USA, pp. 431–440, 2002.

[9] Byers, S., J. Freire, and Silva, C. T, "Efficient Acquisition of Web Data through Restricted Query Interfaces", Proceedings of the International World Wide Web Conferences, Poster session. Hong Kong, China, pp. 184–185, 2001.