Web Spider Performance and Data Structure Analysis

Sadi Evren SEKER

Department of Computer Engineering, Istanbul University, Istanbul, Turkey
academic@sadievrenseker.com

Abstract. *The aim of this study is performance evaluation of a web spider which almost all search engines utilize during the web crawling. A data structure is required to keep record of pages visited and the keywords extracted from the web site during the web crawling. The paper first goes into the detail of possible data structures for a web spider and critics all possibilities depending on their time and memory efficiencies. Furthermore the possibilities are narrowed into tree variations only and a tree is selected from each tree data structure family. Finally, a search engine is implemented and all the tree alternatives from each of the tree data structure family are also implemented and the performance of each alternative is benchmarked.*

Keywords: Web Spider, Web Crawling, Web Indexing, Benchmarking, Data Structures

1 Introduction

In the date of this study, there are 2 kind of possible web sources for the Internet surfers. A user trying to access information on the Internet can either use the directories or the search engines. Directories are hierarchical index lists of sites; they list sites by topic. They are widely used and in many cases offer an extremely great source of information. However, they have few problems: [1]

- Hierarchies are very vulnerable. Data and its classifications change constantly. This also leads to changes in hierarchy. A good example of this is DMOZ [2], world's largest directory. Several subcategories are created, removed or deleted each day.
- Most directories rely on human intelligence and are manually edited. They can never compete with search engines in amount of information. However, quantity is never as important as quality.

This paper concentrates on the search engine architecture rather than the hierarchical indexing. Anatomy of a search engine can be demonstrated as Fig.1.

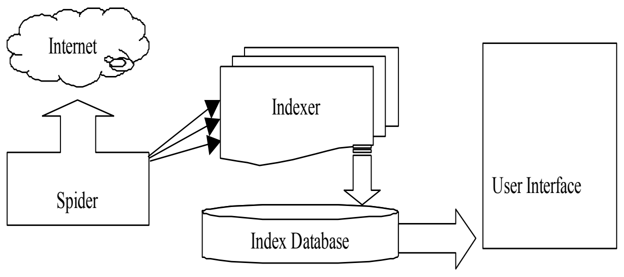


Fig. 1. A sample view of a web spider and its components

From the Fig.1 a spider gets connects to the Internet and supplies information for indexer which is responsible to keep the information for queries. This information can be kept in a database or can stay in memory for faster results. Finally, a user gets connect to the search engine through a user interface and queries the data in the indexer.

One of the most crucial points of a search engine is the indexer and the data arrangement during the data storage and querying.

This paper has been published in SWWS12, Proceeding of Semantic Web and Web Services Conference, 2012,ISBN:1-60132-232-1, Volume 1, Pages: 73-77

The indexer implemented during this study can do:

- Web Parsing: is extracting the pure text from HTML tags,
- Extracting Keywords: is creating a set of keywords and removing duplicates and also cleaning stop words (which are the words does not effect the search like "and, or, a, an, etc."),
- Inverted Indexing: creating an index from the keywords to the web site links instead of keeping keywords in each web site.

The number of queries in an indexer is greatly higher than the number of insertions or updates. This requires a data structure with better query performance required in the indexer. This paper critiques the data structures just after discussing the alternative data structures. In the first chapter this discussion will be ignited and the narrowing alternatives and implementation and benchmarking will go on to the next chapters.

2 Data Structure of Indexer

Indexer is the core data structure in the whole project. The most complex and the most critical point is the implementation of the Indexer. There are several implementation possibilities. It is possible to implement a hash indexer or a tree as an indexer. The problem can be separated into three parts the performance of lookup in the data structure, the performance of update and the performance of memory management. Besides the memory issues, since it is the hardware update as a second choice, we have to concentrate on the time performance. The discussion gets the performance of lookup or update of the tree.

In a real living search engine, the probability of lookup queries would be much more than the queries of the insert or update. Besides the number of queries, the users are directly affected from the search queries, the worst update or worst insert query is not felt by the search engine users. So we have following assumptions in the indexer data structure design phase:

- Memory effectiveness can be sacrificed to time performance

Search queries are much more important than the update or insert queries

So according to the above criteria, we have listed all possible trees in the data structures world in the analysis phase. This section covers the possible tree implementations.

By the definitions on analysis phase, the trees can be grouped into 3 categories.

- B-tree family
- Spatial Access family (a special form of tree Access)
- Binary tree family

Besides the above tree families, in this study we have also concentrated suffix trees because of their importance and reputation on the search engines.

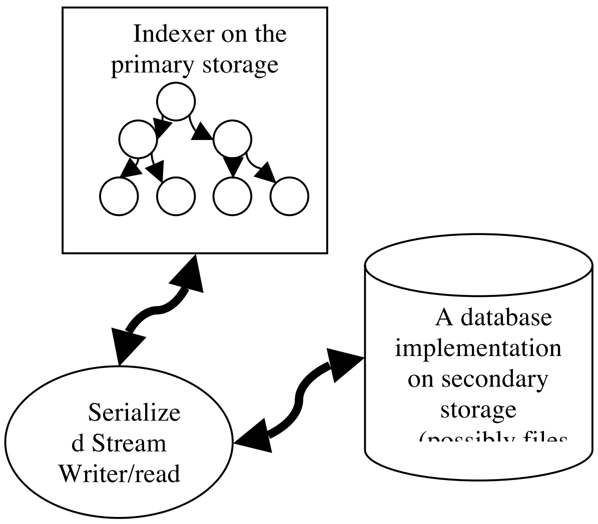
So this study will mainly cover these 4 type of tree implementations. Also the special case of the tree structures gives better results. For example, the AVL tree implementation yields better result than the most of the binary tree implementations. The reason of better results from AVL is the balancing of the tree. For example holding n nodes in an ordinary binary tree and AVL tree yields same worst cases O(log n) in time complexity of algorithm or the O(n) in memory complexity of the algorithm. But the AVL tree uses memory more efficient since the tree is kept in balance. So in the comparison of the AVL tree and an unbalanced binary tree, AVL yields always better results.

The same results can be applied to the k-d tree versus b-tree relation. The k-d tree implementation gives a great variety of indexing over the classical b-tree implementation. The complexity of k-d tree in the search is $O(n^{1-1/d} + k)$, where d is the number of dimensions and the k is the number of reported points. On the other hand the complexity of a classical b-tree query is only O(log n). So most of the cases the performance of k-d tree yields better results.

On the other hand the suffix tree implementations are built over several tree implementations. Most of the cases suffix tree can be built over a balanced search tree. The balanced search tree implementation gives the best result on the most of the cases. The complexity of suffix tree implementation over a balanced search tree implementation is O(log x) for the insertion and lookup where x is the number of alphabets in the language. On the other hand the complexity of traversal is O(1) which is a great speed up for the indexer of the search engine. So the next step would be an implementation of suffix tree over k-d tree or AVL tree structures.

2.1 Data structure of index database

Index database is responsible of managing huge indexes in the memory. Since the amount of ram is limited and the uptime of computers is not reliable, the index database is responsible of keeping the index data into the secondary storage (because of the limitations on the project). The primary focusing data unit is the indexer in the search engine. The indexer database and other data structures are classified as the secondary targets. For the time being, the simplest solution for the index databases it the implementation of a simple file database holding objects in it.



Indexer on the primary storage
A database implementation on secondary storage
Serialize d Stream Writer/read

Fig. 2. Deployment of indexer database

The stream structure gives the ability of keeping serializable objects in the files. So the search engine will dump the index file in the memory to the hard disk in given time periods. The biggest problem about this implementation is the difficulties in the dividing tree into sub parts. This operation is extremely important while the index size is greater than the primary memory. Besides of dividing into sub parts the index database should keep a track of the priorities in the memory and keep the higher priority in the memory always while selecting between low hit and high hit accesses.

Also another type of implementation is the division of whole tree in to parallel or distributed computers. This approach has many benefits besides the increasing primary memory size. The computation and indexing can be divided between the computers as well.

3 Test and Performance Evaluations

This section will cover the tests, debugging and also the benchmarking and appropriate of several indexer implementations.

The basic time measurement tool in JAVA is taking the current system time by using the system library. Unfortunately in my testing environment the results of currentTimeMilis() function from the system library did not yield good results for the time measurement. There were lots of 0 results between the starting and ending time of tree accesses.

Because of these unstable results I have switched to the getting nano second function from the system library again. This function is nanoTime() from the java.lang package.

This second try resulted a valuable numbers and I have added these outputs in 3 different global cumulative variables. Each of these variables holds one of the tree operations. The results are also displayed into the screen when the print times button is clicked.

```
          long
temp=System.nanoTime();

          trie.addString(key,address)
;
          temp-=System.nanoTime();
          trieTime += temp;
          temp=System.nanoTime();
          avl.insert(key,address);
          temp-=System.nanoTime();
          avlTime += temp;
          keyURL a[]= new keyURL[4];
          temp=System.nanoTime();
          bpt.add(new
keyURL(key,address));
          temp-=System.nanoTime();
          bptTime+=temp;
```

Fig. 3. Coding of benchmarking

In Fig.3 code piece demonstrates the calculation of running time of each of the tree operations. The variable “temp” is created and filled up with the system time in the first line. After the creation of this variable the add function of the “trie“ tree is called. The return of the function is also the calculation of the next system time and getting difference from the temp variable. The same operation is repeated for the “bplustree” and the “avl” tree implementations.

Please note that the above code is in a function and called every time when an insertion operation is needed in the tree. So the variables in the above code will keep the cumulative time of each of the tree insert operations.

Also similar to the above insertion operations, the time for each search operation is calculated again. The time measurement of the search operations is same and the value of the search time is added to the cumulative variables holding the time for each data structure.

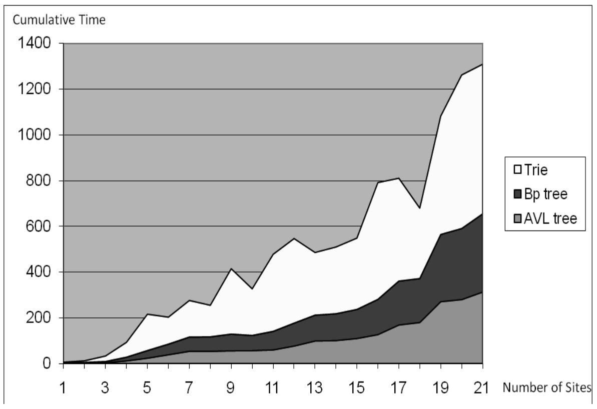


Fig. 4. Time Efficiency of the data structures

Fig.4 holds the tests run over 21 sites with 5 keyword search from each site. The sites are tested by time manner and the cumulative time value is displayed on the y axis of the Fig.4. The dataset of the above graph can be demonstrated as Table 1.

Table 1. A sample view of cse.yeditepe.edu.tr domain search with 5 keywords.

SiteName / Keyword	Time of Trie	Time of AVL	Time of BPT
Site: cse.yeditepe.edu.tr	539082176	552533245	483070 868
Keyword: Faculty	+536991	+523530	+51961
Keyword: Exchange	+5565055	+7971125	+28216
Keyword: Studying	+7286401	+7262935	+31009
Keyword: Application	+1673956	+1623670	+84926

The graph is built over the above tables for each of the 21 web sites. So the web site is first indexed with three different tree data structures and than the keywords are tested as the above sample.

4. Conclusion

This project covers a basic web spider implementation with various indexer possibilities. The test results have shown us the best possible tree implementation for the search engines is the Trie implementation. Its nature also gives the signal of such a result and I have tested this case via this project. Also the bplus tree and AVL has yielded worse results than the Trie but they are very close to each other.

Acknowledgement

This study was supported by Scientific Research Projects Coordination Unit of Istanbul University. Project number YADOP-16728.

References

[1] Koulutus- and Konsultointipalvelu KK Mediat, from SEOGuy.com 2004
[2] Open Directory Project (Directory of Mozilla), 2007
[3] Main source for information on the robots.txt Robots Exclusion Standard and other articles about writing well-behaved Web robots. www.robotstxt.org , 2007
[4] Metasearch.com , a search engine working over the currently implemented search engines. 2007
[5] Sergey Brin and Lawrence Page, The Anatomy of a Large-Scale Hypertextual
Web Search Engine, Computer Science Department, Stanford University, 1999
[6] National Institute of Standards and Technology nist.gov, 2007
[7] TUSSE (Turkish Speaking Search Engine) , http://www.shedai.net/tusse, 2008[8] G.M.
[8] Adelson-Velsky and E.M. Landis, An algorithm for the organization of information, Soviet Mathematics 3 (1962), pp. 1259–1263.