# SPEEDING UP KEYWORD SEARCH FOR SEARCH ENGINES

Sanjay Baberwal & Ben Choi

Computer Science, College of Engineering and Science
Louisiana Tech University, Ruston, LA 71272, USA

pro@BenChoi.org

## ABSTRACT

*In the current information age, the dominant method for information search is by providing few keywords to a search engine. Keyword search is currently one of the most important operations in search engines and numerous other applications. In this paper we propose a new text indexing technique for improving the performance of keyword search. Our proposed technique not only speeds up searching operations but also the operations for inserting and for deleting keywords, which are particularly important for the ever increasing and dynamic changing databases such as that for search engines. We propose to partition all keywords into search trees based on the first character and the length of the keywords. Our partitioning scheme creates a much more even distribution of keywords and results in a 32% speedup in the worst cases and a 1% speedup in the average cases in comparing to one of the leading text indexing techniques called burst tries. In addition, our proposed technique stores document indexes only at the leaf nodes of the search trees and results in efficient algorithms for searching, insertion, and deletion of keywords. We successfully integrated the technique into our Information Classification and Search Engine system and showed its potential and feasibility.*

**Keywords**: Information System and the Internet, Search Engine, Information Storage and Retrieval, Text Indexing

## 1. INTRODUCTION

The search engine has become a necessity for searching information in the Internet. For a search engine to give quick and relevant results, it must have an efficient inverted index. An inverted index is a collection of distinct words, each of which represents its existence in the documents by maintaining a list of references to the documents. Beside efficient searching operation, the indexing techniques for search engines must also have efficient insertion and deletion operations. As new words are constantly being added into the Web, insertion operations are often needed. Unlike other dictionary data structures, efficient deletion operations are also needed for search engines. The delete operation is important as many of the web pages introduce spam words like "wxcscpsr". The web pages with the spam tokens take advantage of a loop hole and try to gain higher ranking. Thus, when those web pages are removed from the Web, the indexed words may also need to be deleted.

Owing to the size of the internet, the collection of distinct words far exceeds 200,000, which does not even taking into account of spam words and proper names [2]. Many text-based indexes were developed and various techniques like hash tables and binary search tree have given adequate results to the text search for small compilations of words, but have their own limitations when more words are added. Due to the collision of entries in hash tables or the increase of nodes in search trees, as the word count increases, the text search usually results in a longer search time.

Besides the large number of words appeared in the Internet, there is the problem caused by the uneven distribution of words based on the first character of the words (see Figure 1). For example, there are about 25,000 words started with character 'S' in comparing to about 400 hundred words started with 'X'. Existing indexing schemes build their search trees simply based on the first character or first two characters of the words. Since searching a large tree usually requires more time than a smaller one, these results in more time required for searching common words.

In this paper, we propose a simpler solution. We observed that words arranged according to its length depict a parabolic shape (Figure 2). For example, the number of words that have nine letters is more than 30,000 in

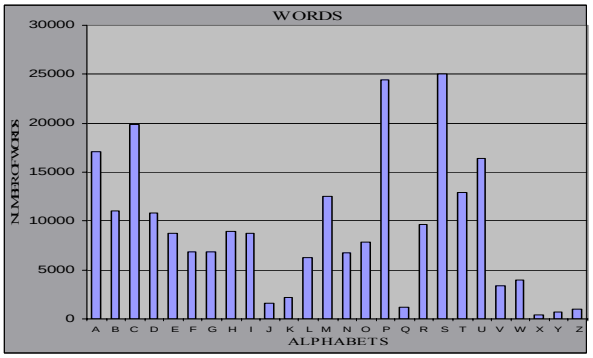


*Figure 1: Pattern of the skewed distribution of English Lexicon. The number of words starting with 's' is nearly 25,000.*

This research was funded in part by a grant from the Center for Entrepreneurship and Information Technology (CEnIT), Louisiana Tech University.

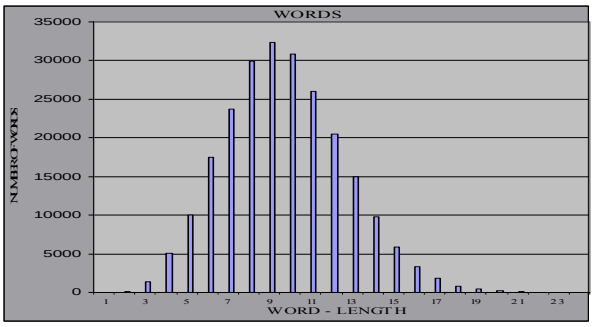


*Figure 2: The words arranged according to its length depicting a parabolic shape. The number of 9 letter words is more than 30,000.*

comparing to a few words started with one or two characters. We decided to combine two criteria for creating a new partitioning scheme.

We propose to partition all words into search tress based not only on the first character but also the length of the words. Our technique, called hash tries, breaks down the whole collection of distinct words into 517 trees and thus reduces the character inspection by segregating the words more uniformly. Our search algorithm begins by finding the appropriate tree based on the first character and the word length and then traverses the tree matching the query string characters with the nodes visited. As the results, in comparing to a leading indexing scheme called burst tries [2], our search algorithm has 32% speedup in the worst cases and a 1% speedup in the average cases. In addition, our proposed technique stores document indexes only at the leaf nodes of the search trees and results in efficient algorithms for searching, insertion, and deletion of keywords.

### 1.1 ORGANIZATION

This paper is organized into five sections. Section 2 provides related research on text indexing techniques. Section 3 describes our proposed approach including our searching, insertion, and deletion algorithms. Section 4 provides testing and performance results. Finally, Section 5 gives the conclusions and provides the future work.

## 2. RELATED RESEARCH

Many indexing techniques has been introduced such as binary search trees, hash tables, red black trees, splay trees, AVL trees, and burst tries. The evolutionary process from binary search trees (BST) to burst tries has been lengthy. A binary search tree having datum and two child pointers in each node has problem that if the data that is inserted is already sorted then BST would become a linear link list. This results in that its performance deteriorates and causes a high searching cost. The shape of the BST is therefore determined by the insertion order of the words. One variant of BST is called AVL tree. This tree has a property that limits the difference of heights of its left and right sub-tree to not more than one, ensures rebalancing. Similarly, red-black trees and splay trees [23], both variants of BST are self-balanced for avoiding nodes forming skewed distribution. However it has been shown that they are no faster than the standard BST. Another disadvantage comparing to BST, in case of splay trees, is that it requires three comparisons at each level, and necessitates an additional space for an extra pointer reference.

Hash tables [17] are faster than any tree structures, but its performance comes with a price. The search can become sequential if the data set is large and the hash table is comparably small. This leads to a longer search time. Also, if the hash table is proportionally larger than the dataset, then it accounts for the wastage of memory. Moreover many hash functions for strings are inefficient, thereby increasing the complexity even more.

Tries [5, 7, 8] are tree structures that have nodes for each character of a string. High memory requirements have led to two main modification of tries: reduction in node size that uses array or link list implementation, and reduction in the number of nodes that is coupled with a technique called Patricia [3, 4, 6, 12], in which the nodes that have no branching are collapsed to form a single node for saving storage space. LC-trie [7] and LPC-trie use level compression in addition to the path compression used by Patricia. Their maintenance, however, is complex. Burst tries [2] is a data structure that has an access trie holding the prefix of the words and the remaining string is stored in BST or other data structures that befits as container. Burst tries by employing heuristics and dynamically reorganize the tree structure is currently one of the leading text indexing techniques [2].

## 3. OUR PROPOSED APPROACH: HASH TRIES

Our proposed approach, called Hast tries, partition all words into search trees based on the first character and the length of the words. This results in creating 517 search trees. The number of words contained in each of the search trees is plotted in Figure 3 and a more detail look of the search trees that start with character A is shown in Figure 4. Among these search trees, the larger tree contains not more than 4000 words. This is a significant reduction comparing to the number of words starts with S is about 25,000.

Our partitioning scheme creates a unique and beneficial property for each search tree. Since all the words in each search tree have the same length, the number of leaves of the tree represents the number of words and all leaves exist at the same depth (see for example Figure 5). Thus, for an inverted index, only leaf notes of the search tree contain document indexes.

A search tree is created having each of it nodes contain one character (see for example Figure 5). Each node also contains two pointers, one for child node and another for sibling node except for the root node that contains a child node and a leaf node contains only a sibling node. As shown in Figure 5, a child pointer is a pointer pointing downward. A sibling pointer is a pointer pointing rightward.

Each path from the root node to a leaf node represents a word that is formed by taking the first character from depth

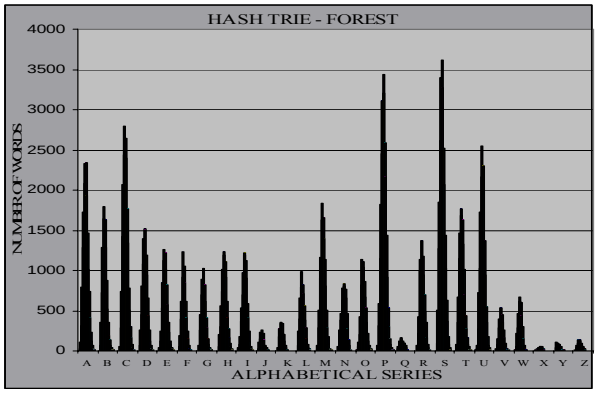


*Figure 3: Forest formed by segmentation of words according to its first character and length. Each stroke in the alphabetical series is a trie tree. The highest number of words that any hash trie tree has is not more than 4000.*

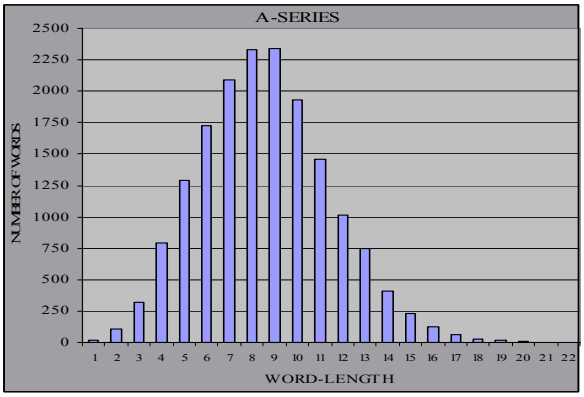


*Figure 4: A close look at the A-series depicts the distribution of words and the maximum number of words in A-series tree is not more than 2500.*

0 and second character from depth 1 and so on till the depth at the leaf. For example, as shown in Figure 5, the left-most path straight down forms the word "best", and the word "bear" is formed from taking 'b' from depth 0, 'e' from depth 1, 'a' from depth 2, and 'r' from depth 3. Hereafter, we will call this type of search trees as hash tries.

These structures of the search trees (hash tries) facilitate searching, insertion, and deletion algorithms as described in following.

## 3.1 SEARCHING

Searching a hash trie is a simple operation requiring traversal along the sibling nodes and if any character matches, it traverses along the child nodes. After determining the length and first letter of query string, search begins with locating an appropriate hash trie and then traversing to the leaf. Referring to the example in Figure 5, if the query string is "blue", the hash trie having the 'b' as root and depth of 4 is located. The search begins with matching the characters of the query string with the nodes visited. The nodes visited include "b → e → i → l → u → e". Each time a node matches with the query string character, the search goes deeper. If there is no branching then it's a straight path downwards. For a successful search, the least number of comparisons required is equal to the query string length. The search continues till it traverses to the leaf of the tree for a successful search and if any of the characters of the query string is not found in the hash trie, it returns back flagging its failure. The search strategy is outlined as follows:

SearchHashTrie

*Locate the hash trie tree based on the query string first character and length. Because the first character always matches with the root, the current node is the child of the root and current depth is 1.*

FOR i = 1 to n-1 *//where n is the length of the query string.*

IF query string $i_{th}$ character matches with the current node THEN go one level deep

ELSE traverse along the same level till it matches with the sibling node.

IF query string $i_{th}$ character does not match with any of the sibling node the search terminates, return false.

The successful search returns with the vector of document indexes.

## 3.2 INSERTION

Insertion algorithm begins with finding an appropriate hash trie by computing insert string first letter and its length. It involves creating a new path from root to the leaf. If there is no root then a root is created along with the path. A pictorial example (Figure 6) showing the state of the tree after the word "bear" is inserted. Since "be" is common, no additional node is added. At the third level, a sibling node is created, thus forming an additional path down to the leaf for the new word "bear". The strategy for inserting new words is outline below:

InsertString

*IF there is no tree then create a root along with the path to the leaf with the input string, otherwise locate hash trie tree by input string first character and length. Because the first character always matches with the root, the current node is the child of the root and current depth is 1.*

FOR i = 1 to n-1 *//where n is the length of the input string.*

IF input string $i_{th}$ character matches with the current node THEN go one level deep

ELSE traverse along the same level till it matches with the sibling node.

IF input string $i_{th}$ character does not match with any of the sibling node then create a path to the leaf with the remaining characters of the input string.

## 3.3 DELETION

To delete a word from a hash trie, we first make sure the word is there in the trie before the deletion process begins. The deletion algorithm is first illustrated by two examples as shown in Figure 7. To delete the word "bill" from the

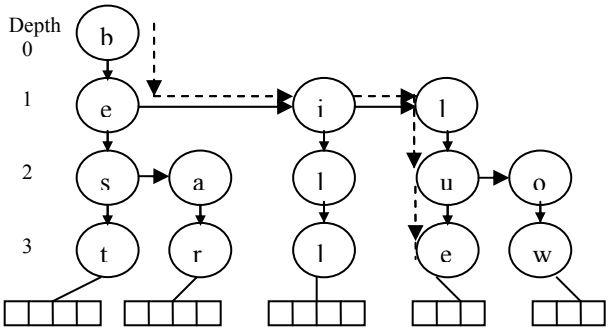


*Figure 5: An example of hash trie tree structure. All words in this tree start with 'b' and have length of 4. The number of leaves represents the number of words in the tree and all leaves exist at same depth. The dotted line is the search path to the leaf. Each leaf node has a vector of document indexes.*

hash trie (left tree of Figure 7), the path containing nodes, i, l, and l is removed. The sibling pointer for node e in depth 1 is updated to point to node l. To delete the word "blue", the path containing nodes u and e is removed. The child pointer for node l in depth 1 is updated to point to node o in depth 2.

Deletion algorithm maintains two states: current node, and previous link. At the beginning, the current node is set to the root node and the previous link is set to null. For deleting the word "bill" as an example, the current node is node b in depth 0. The algorithm checks whether the child node of current node matches character i. In this case, the child node is the node e at depth 1 and does not match character i. Now, the algorithm updates the current nodes to the node e at depth 1 and the previous pointer to the pointer from node b to node e. It continues to find a node to match the character i. Now, it follows the sibling pointer to node i and updates the current node to be node i and the pervious pointer to be the pointer from node e to node i. Since the current node matches the character i, it check downward. It finds that the child node of the current node does not contain any sibling node. It continues to check downward and find a leaf node. Thus, there is a straight down path from current node to a leaf node. Now it can remove this path. It updates the previous pointer to point to the sibling pointer of the current node. It removes the current node and all the descendent nodes to the leaf node. In this case nodes i, l, and l are removed. The strategy for deletion is outline as follows.

DeleteString
*A path from current node to a leaf node is defined to be a straight down path if each node along the path does not have any sibling node.*
Set current node to be the root node
Set previous link to be null
FOR i = 1 to n-1 *//where n is the length of the query string.*
IF query string $i_{th}$ character matches with the current node
THEN IF straight down path exist
THEN remove the straight down path and return
ELSE go one level deep
ELSE traverse along the same level till it matches with the sibling node.
Update current node and previous link

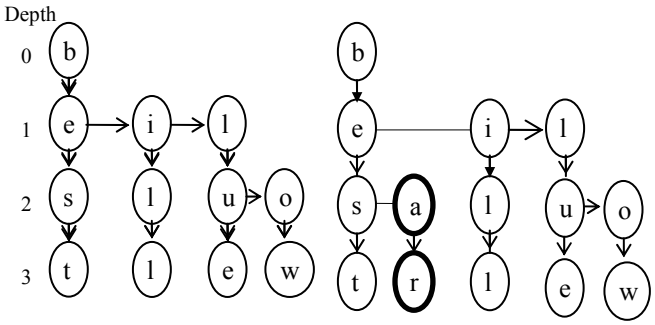


*Figure 6: The states of hash trie before and after the insertion of the word "bear". It creates a path branching out at the third level.*

## 4. TESTING AND PERFORMANCE RESULTS

### 4.1 TESTING

Test data for our performance analysis contains 234,884 English words, which is downloaded from puzzlers.org [25]. The word distribution based on the first character of the words is plotted in Figure 1. The number of words starts with 'S' being the highest, followed by 'P', which is about 25,000 words. The word distribution based on the length of the words is plotted in Figure 2. The top three entries are the 8, 9, and 10-letter words, each of which has about 30,000 words. The combined distribution based on the first character and the length of the words is shown in Figure 3, in which the highest is only about 4,000.

We designed our testing to compare the search performance of our hash tries with that of one of the leading text indexing techniques, called burst tries [2]. For our performance analysis to be machine independent, we counted the number of comparisons necessary to find a word by using hash tries or by using burst tries. This is done for the worst cases and on average cases in each search trees. To insure fair comparisons, care has been taken to ensure binary search trees used as containers of burst tries is balanced; otherwise the trees would tend to form a linear list if the data that is entered is already sorted.

### 4.2 PERFORMANCE RESULTS

The worst case complexity is calculated by finding a word, for which a search takes the maximum number of character comparison compared to all other words that exists in a tree of the forest. The result, summarized in Figure 8, is that on average, the maximum number of character comparisons required by hash trie is 0.68 times the average number of maximum comparisons required by burst trie. Hash trie takes 31.69 character comparisons whereas

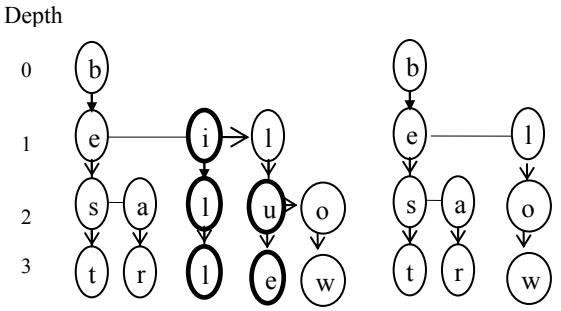


Figure 7**:** The states of hash trie before and after the removal of the word "bill" and "blue" resulting in deletion of the path containing the string "ill" and "ue".

burst trie takes 46.45 character comparisons for the worst case. So hash trie is 32% more efficient than burst trie in the worst cases. It is observed that a hash trie forms 517 trees whereas for the same dataset burst trie forms 358 trees. So it can be argued that skewness still obstructs the burst trie performance.

The burst tries stores all the words starting with 'qu' in one single tree. There are few words which don't start with 'qu' in the 'Q' series. This results in the maximum character comparison required in the 'qu' tree being 77. However, the overall average for 'Q' series dropped. Hash trie performs better than burst trie in all cases where skewed distribution, which is the characteristic of the English lexicon, is present. Since hash trie takes in a word according to its first character and word length, skewness is reduced as the words with the same prefix but with different length, exist in a different tree altogether.

Suppose a query is issued to find a particular word: 'abcxyz'. We also assume that it is not present in the index at all. The burst tries would search all the way to the leaf to confirm that it doesn't exist. However, the hash trie can flag its non-existence whenever the character it is looking for is not present. So each time a non-existent word is searched in a burst trie, its performance is the worst.

The average complexity is calculated by finding the average number of character comparisons that a search requires to find any word that exists in a tree and then finding the grand average of all the averages of the tree that

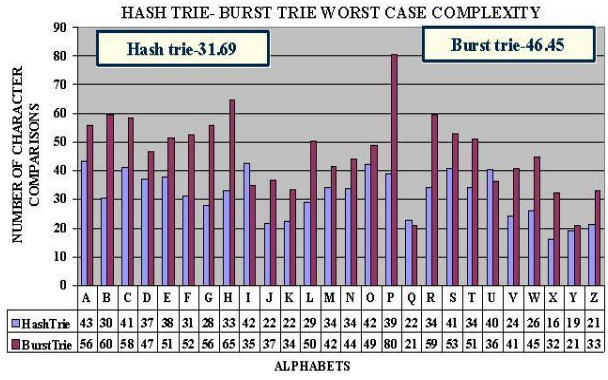


*Figure 8: The hash tries worst case complexity is 32% more efficient than the burst trie.*

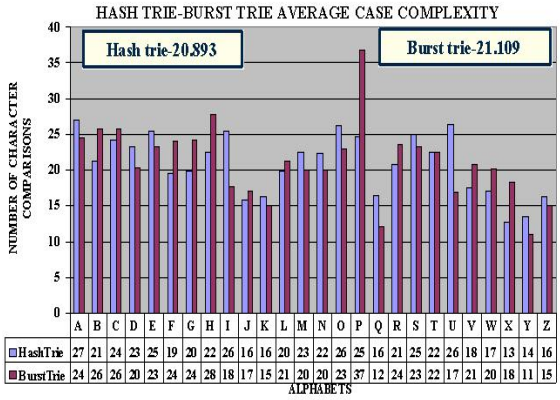


*Figure 9: The average case complexity of hash tries and burst tries is nearly equal. Burst tries has twice as much deviation as hash tries.*

belong to a particular alphabetical series. On average both techniques accomplish nearly equal performance (summarized in Figure 9). Hash tries does perform marginally better than burst trie. It takes about 0.99 times the average number of comparisons required by burst tries. Hash tries takes 20.893 character comparisons whereas burst trie takes 21.109 character comparisons for average cases.

Since the measure by average doesn't show the result distribution, standard deviation is taken into account. The standard deviation, in this context, displays a variance in search times. It is found that burst tries deviation is more than twice that of a hash tries. Burst tries deviates from the mean by 10.444 whereas hash trie deviates mean by 5.717 character comparisons. It is also inferred that the average search of burst tries has more than twice as many instances that fall above the mean than the hash tries. In other words, the hash tries has a more consistent search capability in terms of average cases.

## 5. CONCLUSION AND FUTURE WORK

In this paper, we propose a new text indexing technique called hash tries which reduces the character inspection by distributing words according to its first character and its length into 517 search trees. Experiments were conducted to compare hash tries with burst tries on same dataset by calculating the maximum number of character comparisons required in the worst cases and in the average cases. The results express that hash tries show a 32% improvement over burst tries in the worst case comparisons. For the average cases the hash tries is 1% better than burst tries. In addition, searching a non-existence word in hash tries does not go all the way to leaf as does the case for burst tries. Hash tries is also simple in construction, where all document indexes are stored in the leaf nodes. We also have successfully used hash tries as indexing technique for our Information Classification and Search Engine system [25-30] and demonstrated its potential and suitability for such applications.

We showed that partitioning techniques such as the one proposed in this paper can improve the performance of text indexing. However, new words, including spam words and proper names, are constantly being created and put in the Internet. To keep up with future load, additional criteria for partitioning may be needed. The future research in this direction is to introduce partitioning scheme that no only use the length and the first character of the words, but the length and the first two characters of the words. Since keyword search remains one of the most important operations in information retrieval, further research in this area remains important.

## REFERENCES


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[2] Steffen Heinz, Justin Zobel, and Hugh E. Williams, *Burst Tries: A Fast, Efficient Data Structure for String Keys.* ACM Transactions on Information Systems, 20(2): 192-223, 2002.

[3] M. Al-Suwaiyel and E. Horowitz. *Algorithms for trie compaction*. *ACM Transactions on Database Systems*, 9(2):243-263, 1984.

[4] A. Andersson and S. Nilsson. *Improved behaviour of tries by adaptive branching.* Information Processing Letters, 46(6):295-300, June 1993.

[5] J.-I. Aoe, K. Morimoto, and T. Sato. *An efficient implementation of trie structures*. Software Practice and Experience, 22(9):695-721, September 1992.

[6] J.-I. Aoe, K. Morimoto, M. Shishibori, and K.-H. Park. *A trie compaction algorithm for a large set of keys.* IEEE Transactions on Knowledge and Data Engineering, 8(3):476-491,1996.

[7] R. A. Baeza-Yates and G. Gonnet. *Fast text searching for regular expressions or automaton searching on tries*. Jour. of the ACM, 43(6):915-936, 1996.

[8] J. Clement, P. Flajolet, and B. Vallee. *The analysis of hybrid trie structures*. In Proc. Annual ACM-SIAM Symp. on Discrete Algorithms, pages 531-539, San Francisco, California, 1998. ACM/SIAM.

[9] R. de la Briandais. *File searching using variable length keys*. In Proc. Western Joint Computer Conference, volume 15, Montvale, NJ, USA, 1959. AFIPS Press.

[10] P. Flajolet. *On the performance evaluation of extendible hashing and trie searching*. Acta Informatica, 20:345-369, 1983.

[11] E. M. McCreight. *A space-economical suffix tree construction algorithm*. Jour. of the ACM, 23(2):262-271, 1976.

[12] D. R. Morrison. *Patricia: a practical algorithm to retrieve information coded in alphanumeric*. Jour. of the ACM, 15(4):514-534, 1968.

[13] M. V. Ramakrishna and J. Zobel. *Performance in practice of string hashing functions*. In R. Topor and K. Tanaka, editors, Proc. Int. Conf. on Database Systems for Advanced Applications, pages 215-223, Melbourne, Australia, April 1997.

[14] R. Ramesh, A. J. G. Babu, and J. Peter Kincaid. *Variable-depth trie index optimization: Theory and experimental results.* ACM Transactions on Database Systems, 14(1):41-74, 1989.

[15] I. H. Witten and T. C. Bell. *Source models for natural language text.* Int. Jour. on Man Machine Studies, 32:545-579, 1990.

[16] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, California, second edition, 1999.

[17] J. Zobel, S. Heinz, and H. E. Williams. *In-memory hash tables for accumulating text vocabularies*. Information Processing Letters.

[18] W. Szpankowski. *Patricia tries again revisited*. Jour. of the ACM, 37(4):691-711, 1990.

[19] W. Szpankowski. *On the height of digital trees and related problems*. Algorithmica, 6:256-277, 1991.

[20] W. Szpankowski. *Average case analysis of algorithms on sequences*. John Wiley and Sons, New York, 2001.

[21] M. Regnier and P. Jacquet. *New results of the size of tries*. IEEE Transactions on Information Theory, 35(1):203-205, January 1989.

[22] D.D. Sleator and R.E. Tarjan. *Self-adjusting binary search trees*. Jour. of the ACM, 32:652-686, 1985.

[23] S. Nilsson, M. Tikkanen. *Implementing a dynamic compressed trie*. In Mehlhorn K editor, Proceedings of the 2nd Workshop on Algorithm Engineering (WAE'98), 25-36, Aug 20-22, 1998

[24] Puzzler.org, ftp://puzzlers.org/pub/wordlists/web2.txt

[25] B. Choi and X. Peng, "Dynamic and Hierarchical Classification of Web Pages," *Online Information Review*, Vol. 28, No. 2, 2004.

[26] B. Choi and R. Dhawan, "Agent Space Architecture for Search Engines," IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004.

[27] B. Choi and Q. Guo, "Applying Semantic Links for Classifying Web Pages," Developments in Applied Artificial Intelligence, IEA/AIE 2003, *Lecture Notes in Artificial Intelligence*, Vol. 2718, pp. 148-153, 2003

[28] Z. Yao and B. Choi, "Bidirectional Hierarchical Clustering for Web Mining," IEEE/WIC International Conference on Web Intelligence, pp. 620-624, 2003.

[29] X. Peng and B. Choi, "Automatic Web Page Classification in a Dynamic and Hierarchical Way," IEEE International Conference on Data Mining, pp. 386-393, 2002.

[30] B. Choi, "Making Sense of Search Results by Automatic Web-page Classifications," WebNet 2001: World Conference on the WWW and Internet, pp.184-186, 2001.