An Indexing Technique for Efficient Retrieval from Large Dictionaries

K. Narayana Murthy

Department of CIS
University of Hyderabad
Hyderabad, 500 046 INDIA
Phone: 3010 512 Extn. 4056
FAX: 91-040-3010145
email: knmcs@uohyd. ernet. in

ABSTRACT

This paper is about efficient retrieval from large dictionaries. Dictionaries are often too large to fit into the main memory of a computer. Good indexing schemes are required to make dictionary access efficient. In this paper we present a specific indexing technique for efficient retrieval from large dictionaries. Our scheme includes a non-dense TRIE index stored in main memory and a dense index file stored in secondary memory. To look up any word in the dictionary, only a few character comparisons and a few word comparisons would be required. For both kinds of comparisons, upper bounds can be specified. We can also easily fine tune this indexing scheme to get good performance for any given dictionary and given machine configuration.

1. **Introduction:**

Dictionaries form an essential part of many important applications in computer science. Dictionaries are widely used in text processing applications such as spell checking and grammar checking. Dictionaries are an also integral part of most of the applications in computational linguistics (also called Natural Language Processing or NLP) including parsing, morphological analysis, and machine translation. In fact linguistics has come to assign an increasingly central role to the lexicon in the recent past. Words of a language form a very important part of the knowledge of language. Dictionaries enlist all the words of a language and associate phonological, morphological, syntactic, semantic and other relevant pieces of information with these words. Dictionaries are store houses of valuable information and they have a key role to play in all these applications. The development of large computerized lexical knowledge bases has emerged as probably the most urgent, expensive, and time consuming task facing linguistics and NLP [3]. The importance of computational tools for the design, development and maintenance of such lexical knowledge bases cannot be over emphasized [2,4].

In this paper we focus on only one specific problem - given a large dictionary, how do we achieve efficient look up? For other related issues see [1,5,6]. By 'large' dictionary, we mean any dictionary which is too big to fit entirely into the main memory of a computer. Here we assume that the dictionary resides in the secondary memory. The dictionary can then be viewed as a file of records. There will be one field containing a word from the language in question and this field would be the key field. There can be any number of fields in the record specifying relevant information associated with that word. We do not require that the dictionary be sorted. This makes it easy to add new entries -additions can simply be made at the end. We assume that the primary way we would like to access the dictionary is by looking up a specified word for the associated information.

2. **Primary Dense Index:**

Our scheme includes two levels of indexing. Firstly, there will be a dense index file residing in secondary memory. The records in this file would consist of a key field containing a word and a location filed specifying the location of that word in the dictionary file. The index file is sorted on the key filed, in the alphabetical order of the words.

If this index file could fit into the main memory of a computer, we could have used binary search to locate specific words. The worst case time complexity of locating a given word in the index would then be $O(\log_2 n)$ where n is the size of the dictionary, that is, the number of entries in the dictionary. However, a dense index file is also often too large to fit entirely into the main memory. We therefore assume that this index is also stored in the secondary memory as a file.

The index file is created from the given unsorted dictionary by sequentially scanning the dictionary entries and noting down the head words and the corresponding locations in the dictionary file. The index file is then sorted on the words. We need to use an external sorting algorithm to do this. In our current implementation we have used two-way merge sort algorithm. Initially, chunks of the index file are sorted in main memory and then these sorted chunks are merged pair-wise in stages until the entire index is sorted. The size of the chunks to be initially sorted in main memory can be specified by the user depending upon the main memory available. If bigger chunks can be sorted initially, the number of passes required by the merge-sort would reduce and make the whole sorting process much faster.

3. **Secondary TRIE Index:**

We also use another level of indexing - we will have a non-dense TRIE index residing the main memory. A TRIE, for a set of words defined over an alphabet $\sum$ of size n, is simply a search tree where each node contains n fields. The fields in each node correspond to the n symbols from the alphabet, one symbol for each field. Each field logically points to all the words with a specified prefix. The prefix is simply the concatenation of all the symbols corresponding to the nodes in the path from the root to the current node. Thus a TRIE is simply a collection of pointers corresponding to words having specified prefixes.

A TRIE over an alphabet $\sum$ of size n would have n^l nodes at level l, taking the level of the root node as level 0. Each node requires space for storing n pointers. The depth of the TRIE would not depend upon the size of the dictionary at all but only on the size of the words in the dictionary. The depth of the TRIE equals ml, the length of the biggest word in the dictionary. There will be a total of $O(n^{ml})$ nodes in the TRIE. Hence a dense TRIE index would have a space complexity of $O((n^{ml}) * n)$.

Thus a dense TRIE index is not a feasible indexing scheme for any realistic dictionary. One solution could be to restrict the depth of the TRIE to some pre-specified value. However, the distribution of words with different prefixes in a given language is rarely uniform. Thus a uniform depth cut off is not desirable. Here we employ a variable depth TRIE. The TRIE nodes are expanded further if and only if the number of words in the given dictionary having the prefix corresponding that node exceeds a specified TRIE-Split threshold value. Thus if there are very few words starting with z, we may cut off the z field at the root itself. There are many words starting with 'str', say, and we may here need to go to the 4th level or beyond. A single parameter, the TRIE-Split threshold (TST), controls the creation of the TRIE and for any given dictionary automatically determines for what prefixes it is better to split the TRIE node further and when it is not so. For any given

dictionary and any specified main memory size, we can determine the Threshold value which gives the best results.

To access any word in the dictionary, initially we search in the TRIE. This would involve only character comparisons. Either we get a pointer to the word in the index file straight away or we get a starting point in the index file where words with a prefix of the given word begin. We would also know how many such words are there in the index file. This number will never be more than TRIE-Split Threshold. Now we can either go to the specified location in the index file and start searching sequentially for the full word or bring the relevant set of words from the index file to main memory and then do binary search to locate the full word in the index file. This binary search would require log n_2 word comparisons where n is the number of words to be so searched. In the worst case this n would be TRIE-Split Threshold. If TRIE-Split Threshold is about 16 or less, it would normally be advisable to do sequential search in the index file rather than resort to binary search in the main memory. Once the word is located in the index file, we will know the exact location of that word in the dictionary file and hence we can go there and access all the relevant information. Thus accessing the dictionary requires a few character comparisons and a few word comparisons both of which are determined and limited by a single parameter - the TRIE-Split Threshold value. For any given dictionary and given main memory size, a suitable value for this threshold can be easily determined to get the best possible performance under this scheme.

4. **Conclusions:**

It may be noted in comparison that other indexing techniques including B-Trees and B^+-Trees involve comparing full words. In our scheme, searching for a given word of length m involves at most m character comparisons and at most ($\log_2$ TST) word comparisons where TST is the TRIE Split Threshold value. Also, by adjusting the TST value, one can easily fine tune the indexing

scheme for optimum performance on any given machine for any given dictionary. Similarly, a single parameter - the chunk size, can be used to efficiently sort the entries in the dictionary in secondary memory for creating the index file. We have used this scheme of indexing for indexing not only English dictionaries but also dictionaries of Indian Languages, where the character set itself is quite large.

References

1. A. Sivasankara Reddy, K. Narayana Murthy, Vasudev Varama, "Object Oriented Multipurpose Lexicon", Intnl. Jnl. of Communication, pp 69-83, Vol VI, No. 1&2, Jan-Dec 1996

2. Basant Kumar Sahu, Pramod S. Nagpure, "Text Processing Tools for Natural Language Processing Applications", MCA thesis, Department of CIS, University of Hyderabad, 1995.

3. Bran Boguraev, Ted Briscoe (Eds), "Computational Lexicography for Natural Language Processing", Longman, 1989.

4. Gerald Salton, "Automatic Text Processing", Addison-Wesley, 1989.

5. M. Srinivasa Rao, K. V. S. Prasad, "Software Tools for Electronic Dictionaries", M. Tech. Thesis, Department of CIS, University of Hyderabad, 1995.

6. P. Nageswara Rao, D. Raja Shekar, "Software Tools for Bilingual Bi-directional Electronic Dictionaries", M. Tech. Thesis, Department of CIS, University of Hyderabad, 1996.