Abstract  The search engine vocabulary is normally stored in alphabetical order so that it can be searched with a binary search. If the vocabulary is large, it can be represented as a 2-level B-tree and only the root of the tree is held in memory. The leaves are retrieved from disk only when required at runtime. In this paper, we investigate and address issues associated with the 2-level B-tree structure for vocabulary lookup. Several compression algorithms are proposed and compared. The proposed algorithms not only perform well when the leaves are stored on disk, but also when the whole vocabulary is stored in memory.

Keywords  Inverted Files, Vocabulary Compression.

1 Introduction

Inverted files [16, 15] are the most widely used index structures in Information Retrieval (IR). An inverted file typically contains two parts, a vocabulary of unique terms extracted from a document collection and a list of postings (normally a pair of ⟨document number, term frequency⟩) for each of the vocabulary terms. Due to the large size of the postings, various techniques have been developed to improve the processing of them, including better compression algorithms [12], impact ordering [9, 2] and pruning [2, 13, 5]. With the applied techniques, the postings can be efficiently loaded and processed. What becomes of interest is the time taken to search through the vocabulary in order to locate the postings for the terms.

The vocabulary is normally stored in alphabetical order so that it can be searched with a binary search. If the vocabulary is large, it can be represented as a 2-level B-tree and only the root of the tree is held in memory. The leaves are retrieved from disk only when required at runtime. Three steps are required to locate the postings for the terms; (1) A binary search of the root gives the disk location of the leaf. (2) One disk-seek and one disk-read load the leaf into memory. (3) A search through the leaf gives the location of the postings.

There are three efficiency issues related to the 2-level B-tree structure. First, disks are an order of magnitude slower than main memory and it can take a long time to read the leaves. Second, the size of the leaves has a big impact on the performance of disk I/O. If the size is large, it can take a long time to read. However, large leaves might benefit from disk caching provided by operating systems [4]. On the other hand, small leaves are fast to read, but cannot benefit much from the caching. The last issue is how the leaves should be compressed. Algorithms with better compression ratios might be slow to decompress at runtime, due to the complexity of the algorithms.

In this paper, we investigate and address issues associated with the 2-level B-tree structure for vocabulary lookup. A number of compression algorithms are proposed and compared. As shown in the results, the proposed algorithms not only perform well when the leaves are stored on disk, but also when the whole vocabulary is stored in memory. The embedfixed algorithm provided the best trade-off between compression ratio and fast lookup.

2 In-memory Algorithms

Manning et al. [8] discuss storage efficiency for in-memory vocabulary compression and provide three data structures. This section gives a brief discussion of those structures.

In fixed, vocabulary terms are sorted in alphabetical order and stored in fixed-width blocks as shown in Figure 1(a). The example shown in the figure allocates 24 bytes for each term, followed by an 8-byte postings pointer (the location of the postings for the term). Terms are null-terminated (shown as “ø”) for fast string comparison. This fixed structure is clearly not very space efficient, since the average length of a term is expected to be small. Also terms longer than 24 characters have to be truncated. In order to alleviate these problems it is better to store terms in variable-width structures.

The string structure stores the vocabulary terms as a long string of characters as shown in Figure 1(b). The vocab pointers are used to identify the beginning of each term and the end of each term is specified by the next vocab pointer. As strings are not null-terminated
special string comparison using string length has to be used.

The blocked-\(k\) structure further extends string by grouping terms into blocks of size \(k\) and assigning a vocab pointer to each group. An example of blocked with \(k = 2\) is shown in Figure 1(c). An exception is that the last block has only one term since there are only five terms in the illustrated example.

The blocked-\(k\) structure provides better compression ratio by reducing the number of vocab pointers required. However it takes \(O(\log_2(\frac{n}{k}) + k)\) to search instead of \(O(\log_2 n)\) for fixed and dictionary-as-string, where \(k\) is the blocking factor and \(n\) is the total number of terms in the vocabulary.

3 Fast Lookups with 2-level B-tree

For large data collections and systems with limited resources, it is not feasible to store the whole vocabulary in memory. In this section we discuss how to efficiently compress the vocabulary in terms of both storage space and speed of search using a 2-level B-tree. The root of the tree is stored in memory. The leaves are stored on disk and retrieved only when required.

3.1 Prior Algorithms

It is easy to convert Manning’s structures into 2-level B-tree structures and he suggests this. The header structure (the root of the 2-level B-tree) is constructed to allow fast lookup using binary search in order to locate the leaf. The leaf structure is built with the consideration of disk properties so that leaves can be stored efficiently on disk and read quickly from disk.

Two fields are required for each entry in the header structure. The first field stores the first term stored in each leaf. The header terms are null-terminated for fast string comparison. The second field stores the disk offset to the leaf on disk. The size of the leaf pointer is dependent on the vocabulary file size and might be either 4 or 8 bytes. Throughout our discussion, we use 8 bytes for the leaf pointer. The leaf count entry at the beginning of the header stores the number of leaves. The left part of Figure 2(a) shows a typical header. On average, it takes \(O(\log_2(\text{lc}))\) to locate the leaf in the header, where \(\text{lc}\) is the number of leaves.

In order to optimise disk I/O, a leaf should be a whole number of physical disk sectors. The sector is the smallest unit a disk can read/write. If leaves are sector aligned, the time taken to read a leaf should be minimised. During construction of the leaf, many terms as possible are inserted into the leaf and the remaining bytes padded with zeros.

In order to convert the original fixed structure into a 2-level B-tree, each entry in the structure should be given a specific size so that terms do not cross sector boundaries. As shown in Figure 1(a) in our experiments, each entry is assigned 32 bytes, 24 bytes for storing the term and the other 8 bytes for the postings pointer. The size of a disk sector is normally 512 bytes, which is evenly divisible by 32. There is no need to store a header, instead the header can be constructed at startup time.

Figure 2(a) and Figure 2(b) show the original string and blocked represented as a 2-level B-tree. The header on the left in the Figure 2(a) is that describe above. The leaf structures are identical to the original structures encoded using string and block-k respectively, the leaf length entry is added to the beginning of the structure and stores the number of entries in the leaf.

3.2 New Algorithms

Our new embed algorithm is based on the assumption that similar data types should be grouped together as CPUs are good at caching and fast at reading data of a fixed length. The data types used in the leaf structure are word-aligned as CPUs read/write data in units of words. A vocab pointer is assigned to each term stored in the leaf and the stored terms are not compressed, thus no de-serialisation is required. As shown in Figure 2(c), embed leaf first stores the postings pointers, followed by the vocab pointers, then null-terminated strings and finally the number of entries in the leaf. The header structure of embed is the same as the header structure in 2-level string. The leaf structure of embed is similar to 2-level string with the exceptions that (1) elements are re-ordered for faster CPU processing and (2) terms are

![Figure 1: The in-memory vocabulary structures from Manning et al. [8]. In blocked-k, k has a value of 2.](image-url)
null-terminated so that standard intrinsic string comparison can be performed.

The search time of embed is $O(\log_2(l_c))$, where $l_c$ is the number of leaves in the header, plus $O(\log_2(l_l))$, where $l_l$ is the number of entries in the leaf. The performance of embed is efficient as it allows binary search in the header and leaf structures to locate the term (no deserialisation is required). However, embed is not space efficient. Terms inside each leaf are not compressed. Two new algorithms are introduced to overcome this problem, embedfront and embedfixed.

The embedfront algorithm uses front coding [14, p. 122] to compress terms in each leaf. Front coding takes advantage of the fact that adjacent terms in sorted alphabetical order tend to share a common prefix. Two integers (we use one-byte integers) are used; one to track the size of the common prefix between the current term and the previous one, and another is the size of the suffix for the current term. As shown in Figure 2(d), “term,terms,termstr,termsstr,them” is front encoded as “4term41s52r71s13hem”. The first term has a value of zero for the common prefix count and there is no need to store this.

Since there is no way to access the terms directly without de-serialisation, there is no need to store the vocab pointers for each term. The header structure of embedfront is identical to the header structure in 2-level string.

The embedfront algorithm not only compresses terms more effectively but also does not need vocab pointers. However, the search time in the leaf requires $O(\log_2(l_l))$ or a de-serialisation and a binary search of $O(l_l + \log_2(l_l))$.

### 3.2.1 Embedfixed

The embed algorithm allows fast lookup, but provides no compression of terms. On the other hand, embedfront uses front coding to compress the terms stored in leaves, but de-serialisation is required for lookup. An intermediate solution, e.g. embedfixed, is proposed to provide a trade-off between embed and embedfront.

The embedfixed algorithm uses a simple but effective coding method for compressing terms and allows fast lookup without de-serialisation. Instead of allocating leaves as multiples of disk sectors and squeezing as many terms as possible into each leaf, a leaf in embedfixed only contains terms with the same common prefix. The leaf size thus depends on the number of common prefix characters and how many terms share that common prefix. Additionally and uniquely, only the suffixes of the terms are stored in the leaf and are null-terminated for fast string comparison. Essentially, our algorithm is a form of Trie [6, Section 6.3, p. 412].

The construction of the header is also different from the previous algorithms. Instead of storing the whole term (of varying lengths) in the header, only the characters of the common prefix are stored, and all common prefixes are of the same length. There are several advantages to constructing the header structure this way:

1. For the same number of leaves, the structure has a smaller footprint compared with the previous header structure since only partial terms are stored.
2. During lookup, a shorter string comparison is performed to locate the leaf containing the term.
3. If the partial terms
are treated as integers, integer comparison can be used for fast lookup instead of standard string comparison.

Essentially, embedfixed splits terms into two parts, the common prefix and the rest of the term (the suffix). For example, the term “termstr” with common prefix “term” will be split as “term” and “strø”. The common prefix “term” will be stored in the header and the suffix “strø” will be stored in the leaf.

Figure 3 shows the complete structure of embedfixed. In the header, 4 bytes are allocated for the common prefixes and 8 bytes for the leaf pointers. The leaf count and max leaf len entries at the beginning of the header structure store the number of leaves and the size of the largest leaf respectively. The max leaf len entry is used to allocate one fixed-sized buffer for reading leaves throughout the execution of the search engine. In the leaf, the 8-byte postings pointers are stored first, followed by the 4-byte vocab pointers, then the null-terminated suffixes and finally the number of entries in the leaf. The vocab pointers are used to avoid de-serialisation of the suffix string. With the illustrated terms of “term,terms,termstr,termstrs,them”, the header will have common prefix “term” and the terms are compressed as “strøstrøstrøstrø” in the leaf. The last term “them” does not share a common prefix with “term” and has to be stored in a different leaf (not illustrated).

There is a case that a leaf may contain only one term. This happens when the current term does not share a common prefix with the next term. For example, the terms “termstr,worm” have no common prefix. The first four characters of the term “termstr” will be stored in the header since the header has a length of 4 byte for the common prefix. The suffix “strø” will be stored in the leaf by itself. In this case, the vocab pointer is no longer needed. The leaf length entry is still required to indicate that it is a leaf containing a single term.

4 Experiments

We conducted all our experiments on a system with dual quad-core Intel Xeon E5410 2.3 GHz, DDR2 PCS300 8 GB main memory, Seagate 7200 RPM 500 GB hard drive and running Linux with kernel version 2.6.30. The hard drive has a model number of ST3500320AS and 512 byte sectors [11].

The collections used were the INEX 2009 Wikipedia collection [10] and the uni-gram corpus from the Web 1T 5-gram version 1 [3]. The Wikipedia collection has about 2.6 million documents, 2,348,343,176 total words and 11,393,924 unique words. The Web 1T corpus was generated from about 1 trillion terms of text from public web pages. It contains English word n-grams (uni-grams to 5-gram) and their frequency counts. Throughout the experiments, only the uni-grams was used. There are 13,588,391 words in the uni-gram data set.

The Million Query Track queries from TREC 2007 [1] were used for throughput evaluation. The track has a total of ten thousand queries with a total of 41,671 terms. The average query length is about 4 terms.

Instead of using a real search engine, a simulation program was written and used for the evaluation. The advantage of using a simulation is that it is easy to implement and evaluate the algorithms. At the same time, the simulation program can provide the correct performance since it mimics how a real search engine process queries for vocabulary lookup. Through out the experiments, only the vocabulary of the inverted files was built as the postings are not touched in the experiments.

The program has two executables, build-dictionaries and search-dictionaries. The build-dictionaries executable uses the parser from the search engine described in [13, 5] and takes a number of parameters including (1) which vocabulary structure to build, (2) the number of disk sectors for the leaf size and (3) the value of k for the blocked-k structure.

In order to minimise the interference from the Linux kernel during the evaluation, one of the CPU cores is configured off the management of the kernel and the search-dictionaries process was assigned to that particular core using the CPU affinity system calls [7, p. 172]. search-dictionaries also takes a number of parameters, including (1) which vocabulary to search, (2) whether the nodes are stored on disk or the whole vocabulary is loaded into memory, (3) the number of sectors for the leaf node size, (4) the value of k for the blocked-k structure and (5) which query file to use for batch mode processing.

5 Results

Four sets of experiments were conducted to test the 2-level vocabulary algorithms of fixed, string, blocked-k with a value of 2, 4 and 8, embed, embedfront and embedfixed. Unless specified, the number of characters stored in the header are 4 bytes long for embedfixed. All search results were average over twenty runs. The gettimeofday() function was used for the timing.

5.1 Experiment One

The first set of the experiments examined the storage space of the proposed 2-level structures with various leaf sizes. The leaf sizes were 1, 2, 4, 8, 16 and 32 sectors and the corresponding length in bytes are 512, 1024, 2048, 4096, 8192 and 16384 (the disk has a sector size of 512 bytes).

Figure 4(a) shows the results. In both collections, fixed and embedfixed showed static storage space as the size of the leaf did not depend on the number of sectors. fixed required about 349 MB to store the Wikipedia collection and about 337 MB for the unigram data set.

1But the ATIRE search engine currently under development at University of Otago now uses essentially the embedfixed algorithm and so it has been tested in a search engine.
The total storage space and the wastage of the storage space are shown in Figure 4. The left figure shows the results using the Wikipedia collection [10] and the right figure shows the results using the uni-gram data set in the Web 1T corpus [3].

embedfixed used about 196 MB and 197 MB respectively in the Wikipedia collection and the uni-gram data set. The storage space steadily decreased as the leaf size increased for string, blocked-2, blocked-4, blocked-8, embed, and embedfront in both collections. When the leaf size increased from 1 to 8 sectors, blocked-8 had the biggest drop in the storage required from 244 MB to 201 MB in the Wikipedia collection and from 232 MB to 193 MB in the uni-gram dataset.

Overall, embedfront was the most storage efficient and used about 62% and 60% less storage than fixed in the corresponding collections. In most cases, embedfixed performed better than the other algorithms except that it used about 60 MB and 50MB more storage than embedfront in the corresponding collections.

There are two reasons why embedfront used less storage space than embedfixed. First, the 4-byte vocab pointer is not used in embedfront. For a leaf stored with 100 terms, 400 bytes are saved (100 pointers of 4 bytes each). Second, embedfront computes the common prefix dynamically between each adjacent terms, while the common prefix in embedfixed is preset. For example, for “term,terms,termstr,termstrs”, embedfront computes the common prefixes of “term”, “terms” and “termstr”, while shorter common prefix of “term” is used for embedfixed. For terms sharing long common prefixes, embedfront provides better compression.

Figure 4(b) shows the wasted storage space due to internal fragmentation in leaves. Apart from fixed and embedfixed, as the size of the leaf increased more storage space was wasted. However, the wastage is small compared with the total storage space.

There is an apparent contradiction when comparing Figure 4(a) and Figure 4(b). As the number of sectors increased, the wastage increased, however the total storage space decreased for string, blocked-k, embed and embedfront. A common factor for the reduction of the storage space is a reduction in the number of leaves and hence the size of the header is smaller. Furthermore, block-k also benefitted from combining two or more odd sized leaves into even sized leaves and embedfront from better compression of terms stored in larger leaves.

5.2 Experiment Two

The second set of the experiments examined the search performance when the header of the vocabulary structures was loaded into memory and only the required leaves were retrieved from disk. The simulation program did not cache I/O, but depended on the Linux kernel to do so. Before each run of the experiments, the disk cache was flushed.

Figure 5(a) shows the I/O time, taken to read the required leaves from disk. The I/O time for embedfixed is approximately constant regardless of the number of sectors to store each leaf. For other algorithms, the
Figure 5: The performance of the algorithms when leaves were stored on disk. The left figures shows the results using the Wikipedia collection [10] and the right figures shows the results using the uni-gram data set in the Web 1T corpus [3].
I/O time decreased with the increasing number of sectors. For both collections, embedfixed performed the best, embedfront took more time than embedfixed but performed better than other algorithms, and fixed performed the worst.

The performance difference between fixed, string, blocked-k, embed and embedfront showed the impact of the total file size on disk I/O. The smaller the file size, the less time taken to read from disk.

The embedfixed algorithm performed the best, even though it had a larger file size than embedfront. This is due to the fact that embedfixed accessed a smaller number of leaves than embedfront and the leaves were larger. A large leaf can contain more terms than a small leaf and be better cached by the operating system. For example in the Wikipedia collection, embedfront had 280,399 leaves when the leaf was 2 sectors. On the other hand, embedfixed had 314,871 leaves but only 169,187 of them contained more than one term. 169,187 large blocks is smaller than 280,399 blocks, and they were read and cached earlier in the experiments. Figure 5(b) shows the total number of bytes read from disk. When the leaf had a size of 2 sectors, embedfixed read 560 MB and 950 MB more than embedfront respectively in the collections.

Figure 5(c) shows the search time. As the leaf size increased from 1 to 32 sectors, the search time taken by embedfront grew exponentially on the log scale of the number of sectors (linearly in linear scale). The performance gap between embedfront and the other algorithms was caused by the de-serialisation in order to access the individual terms stored in leaves during the lookup.

As shown in Figure 5(d), the total time was dominated by the I/O time. Figure 5(d) is almost identical to Figure 5(a) because the CPU time was order of magnitude lower than the I/O time. Overall, embedfixed showed the best performance.

5.3 Experiment Three

The third set of the experiments examined the search performance when the whole Vocabulary was loaded into memory. The results show the same pattern as shown in Figure 5(c). embedfront grew linearly as the leaf size increased. Figure 6 shows a detailed plot of the results from Figure 5(c), but without embedfront. In both collections, The time to search for string, blocked-2, blocked-4, blocked-8 and embed increased as the leaf size increased, while fixed and embedfixed showed static performance.

5.4 Experiment Four

The last set of the experiments explored various lengths of the common prefix in the header for embedfixed. Figure 7(a) shows the required storage space as the size of the common prefix increased from 1 to 10 bytes. In both collections, the required storage space decreased from 1 to 4, and increased from 7 to 10. The embedfixed algorithm used the least storage space when when the common prefix had a length of 4.

Figure 7(b) shows the search performance when the whole vocabulary was loaded into main memory. In the Wikipedia collection, embedfixed showed the best performance when the common prefix had a length of 5 or 6. While in the uni-gram data set, embedfixed had the best performance when the common prefix had a length of 8. This is due to how terms are distributed among the header and leaf structures. For a header with long common prefix, more terms are squeezed into the header and more characters have to be compared in order to locate the leaf. For a header with shorter common prefixes, fewer terms are squeezed into the header and less characters are compared, however it takes longer to locate the term in the leaf.

6 Conclusion and Future Work

In this paper, we have investigated and addressed the related issues associated with the 2-level B-tree structure for storing the vocabulary of a search engine. We have conducted experiments on a number of algorithms, including 2-level fixed, string, blocked-k, embed, embedfront and embedfixed. Our new embedfixed algorithm provides a simple but effective encoding method
by storing terms with the same common prefix in the same leaf, and the shared common prefix in the header structure. The new encoding method allows fast lookup without de-serialisation, with an average search time of $O(\log_2(ls))$, where $ls$ is the number of leaves in the header, plus $O(\log_2(ll))$, where $ll$ is the number of entries in the leaf.

In terms of compression ratio, embedfront showed the best performance in both the Wikipedia collection and the uni-gram data set. The embedfixed algorithm used about 60 MB and 50 MB more storage space than embedfront in the corresponding collections. While in terms of vocabulary lookup performance, embedfixed showed the best performance in both sets of the experiments when only the header structure was loaded into memory and when the whole vocabulary was loaded into memory.

However, the performance of vocabulary lookup is dependent on how disk data is cached in memory. An extreme case is that systems do not cache disk data at all. For such systems, the algorithms with smaller leaf sizes will perform the best since less data is read from disk. In this case, the embedfront algorithm with a leaf size of 1 sector should be used. The opposite extreme is that the whole vocabulary can be loaded into memory at startup time, thus disk I/O is eliminated during lookup. In this case, the embedfixed algorithm should be used. In future experiments, we will explore the performance of the algorithms on systems with limited resources, for example smart phones.

Acknowledgements

Thanks Dylan Jenkinson for his early contribution to this work.

References