In-memory Hash Tables for Accumulating Text Vocabularies

Justin Zobel Steffen Heinz Hugh E. Williams
Department of Computer Science, RMIT University
GPO Box 2476V, Melbourne 3001, Australia
jz@cs.rmit.edu.au, steffen@mds.rmit.edu.au, hugh@cs.rmit.edu.au

Keywords Data structures, hashing, splay trees, text databases, index construction.

Introduction

Searching of large text collections, such as repositories of Web pages, is today one of the commonest uses of computers. For a collection to be searched, it requires an index. One of the main tasks in constructing an index is identifying the set of unique words occurring in the collection, that is, extracting its vocabulary. This vocabulary is used during index construction to accumulate statistics and temporary inverted lists, and at query time both for fetching inverted lists and as a source of information about the repository.

In the case of English text, where frequency of occurrence of words is skewed and follows the Zipf distribution [8], vocabulary size is typically smaller than main memory. As an example, in a medium-size collection of around 1 Gb of English text derived from the TREC world-wide web data [2], there are around 170 million word occurrences, of which just under 2 million are distinct words. The single most frequent word, "the", occurs almost 6.5 million times—almost twice as often as the second most frequent word, "of"—while there are more than 900,000 words that occur once only.

In this paper we experimentally evaluate the performance of several data structures for building vocabularies, using a range of data collections and machines. Given the well-known properties of text and some initial experimentation, we chose to focus on the most promising candidates, splay trees and chained hash tables, also reporting results with binary trees. Of these, our experiments show that hash tables are by a considerable margin the most efficient.

We propose and measure a refinement to hash tables, the use of move-to-front lists. This refinement is remarkably effective: as we show, using a small table in which there are large numbers of strings in each chain has only limited impact on performance. Moving frequently-accessed words to the front of the list has the surprising property that the vast majority of accesses are to the first or second node. For example, our experiments show that in a typical case a table with an average of around 80 strings per slot is only 10%–40% slower than a table with around one string per slot—while a table without move-to-front is perhaps 40% slower again—and is still over three times faster than using a tree. We show, moreover, that a move-to-front hash table of fixed size is more efficient in space and time than a hash table that is dynamically doubled in size to maintain a constant load average.

Candidate data structures

The task we are investigating is accumulation of the vocabulary of a large text collection. As discussed above, such vocabularies typically contain millions or tens of millions of distinct words. This volume of data can be managed in the memory of a current machine—say 256 Mb for a large desktop computer—but greatly exceeds typical CPU cache size of around 1 Mb. Thus each random memory access can involve a memory fault and a delay of perhaps 10 processor cycles.

Binary search trees. In the average case, a simple binary search tree (or BST)—even without rebalancing or other enhancements—could well be efficient for this task. Common words should occur close to the start of the corpus and thus should be placed high in the tree. These frequently-accessed words should be retained in the CPU cache. Most failed comparisons involve only the first characters of the string and require only a few operations. Searches for rare or new strings are more costly, however, so the performance in practice depends on the distribution of words. In other work [7] we have found BSTs to be approximately as efficient in practice as other tree structures. Each node requires two pointers in addition to the stored string itself.

Splay trees. Splay trees are a variant of BSTs, in which the node accessed in a search is moved to the root of the tree through a series of rotations. Splaying has two beneficial effects: the worst case over a series of searches and insertions is only a constant factor worse than the average case, and it tends to keep frequently-accessed nodes near to the root, and thus should be suitable for skew data sets such as word occurrences in text.

An efficient implementation requires that each node store three pointers in addition to the stored string, making splaying the most space-intensive of the data structures we consider. Another drawback of splaying is the cost of reorganising the tree, with around three comparisons and six assignments for each level. We have found that a practical heuristic that addresses this problem is to only rotate at every nth access, with say $n = 11$ [7].

Hash tables. For in-memory variable-length string management, the most efficient form of hash table is to use chaining, in which the table is a fixed-length array of pointers to linked lists of nodes; each node contains a string and a pointer to the next node. For this task, common words are likely to occur early in the text and should therefore be found towards the front of the linked list for that hash value. Given a sufficiently large hash table and the likely skew access pattern, even as the number of stored terms grows to be much larger than the hash table average access costs should not be excessive.

Access costs can be reduced by doubling the table size when the load factor reaches some fixed threshold (which is a computationally cheap way of estimating average access costs). We compare below a variety of fixed-size hash tables to dynamic hash tables with size doubling.

A crucial element of hashing is choice of hash function. Many hash functions for strings are highly inefficient, such as the "radix" hash function for strings in a recent edition of a popular textbook [6], which uses two multiplications and a modulo for each character in the string. Much greater efficiency, and equally good behaviour, can be obtained from "bit-wise" hash functions based on operations such as shift and exclusive-or [5].

Other structures. Further candidate structures for this task include tries, B-trees, AVL trees, skip lists, and T-trees. Tries have the potential to be fast, but are extremely space-intensive. In preliminary experiments we have been unable to use tries to accumulate vocabularies of more than a few hundred thousand words; in contrast, the other structures were easily able to accrue the vocabularies of around 10,000,000 words. Ternary tries are somewhat more space-efficient than tries, but still require an average of around one pointer per character per distinct string [1], and thus are not suitable for this application.

B-trees are not ideally suited for searching of skew data. The number of comparisons amongst n strings is always close to $\log_2 n$—other balanced structures, such as AVL trees and skip lists, also have this drawback—and space management within B-tree nodes must be either array-based, giving costly insertion, or tree-based, thus ensuring that B-trees are less space efficient than the other strategies. Skip lists, moreover, require more key comparisons than the other schemes [4]; in our experiments we have found that string comparisons are the dominant cost.

T-trees have been specifically proposed as a suitable structure for in-memory management of large sets of distinct search terms [3]. T-trees are similar to BSTs, with the modification that each node contains an array of lexicographically adjacent search terms, up to a fixed

Table 1: *Statistics of each text collection.*

	Small AP	TREC1	Stopwords	Small Web	Large Web
Size (Mb)	23	1,206	292	2,146	31,745
Distinct words	74,439	618,443	589	1,335,011	9,211,024
Word occurrences	3,693,936	183,710,119	74,940,753	238,894,460	2,009,762,446

limit. With only two pointers per node they are thus more space-efficient than BSTs, or even hash tables, but require a more complex search procedure. In a series of past experiments we compared T-trees to BSTs and splayed T-trees to splay trees, and found T-trees to be consistently slower; problems included the search conditions and, on insert, the need to extend the node to accommodate the new string. We do not explore T-trees here, but note that they may have a role if space is limited.

Experiments

Test data. The test data we use to compare the structures is drawn from the TREC project. We use five data sets. First is a small file, "Small AP", drawn from the Associated Press subcollection. Second is "TREC1", the data on the first TREC CD. Third is "Stopwords", the data on the first TREC CD after all words other than 601 common and closed-class words have been removed; this data set shows the performance of the data structures when the vocabulary is very limited, an environment that, relatively, should favour trees. Fourth is "Small Web", a small collection of web pages extracted from the Internet Archive for use in TREC retrieval experiments. Last is "Large Web", a larger collection of such web pages. The statistics of these collections are shown in Table 1.

We used three different machines to compare schemes. Experiments with the Small AP, TREC1, and Stopword were run on a single-CPU Pentium II 300 MHz with 256 Mb of memory. Experiments with the Small Web data were on a dual-CPU Sparc 20, with 384 Mb of memory. Experiments with the Large Web data were on a dual-CPU Intel Pentium II 233 MHz with 256 Mb of memory. (We chose to use both a Sun and a Pentium to see any dependency on hardware, but none was obvious; two separate Pentium machines had to be used because they held different data collections.) In all experiments the times shown are CPU, not elapsed.

The hash functions, search routines, and insertion routines used in this paper are available on the web site `http://www.cs.rmit.edu.au/~hugh/zhw-ipl.html`. This web site also includes pointers to other related material, such as the home page of the TREC project.

Table 2: *Running time in seconds (in parentheses, peak memory usage in megabytes) for each data structure.*

	Small AP	TREC1	Stopwords	Small Web	Large Web
Binary tree	9.3 (2)	521.9 (13)	89.2 (1)	1126.4 (30)	6232.0 (208)
Splay, all searches	12.5 (2)	605.4 (15)	120.8 (1)	1877.1 (35)	7170.0 (243)
Splay, intermittent	10.1 (2)	494.5 (15)	88.2 (1)	1405.9 (35)	6107.0 (243)
Radix hashing	5.6 (6)	288.8 (15)	71.0 (5)	1408.7 (28)	3127.0 (177)
Bit-wise hashing	2.5 (6)	123.9 (15)	30.2 (5)	404.1 (28)	1361.1 (177)

Results. In our experiments we measured the time and space required by three kinds of data structure: BSTs, splay trees, and hash tables. For splay trees, we measured the time taken for a tree splayed at every access, and for a tree splayed at every eleventh access. For hash tables, we measured the time taken with two hash functions, radix and bit-wise. In each case we used a 4 Mb hash table, that is, of 1,048,576 slots. (Note that it is not necessary that hash tables be prime in size. In other work we have verified that these hash functions yield a uniform distribution of hash values on this kind of data [5], even when the hash table size is a power of 2. Use of powers of 2 allows economies throughout the code.)

Results are shown in Table 2, with times in seconds and peak memory use in parentheses. The reported times are for accesses to the data structure only, and do not include costs such as parsing the input data. Hash tables are much faster than trees, around 16 minutes instead of 94 minutes, for example, to process 30 gigabytes of words. As expected, radix hashing is slower than bit-wise hashing, but the magnitude of the difference is remarkable; we estimate that, with radix hashing, over 80% of the total time spent managing the data structure is consumed by hashing. Compared to integer arithmetic, floating point instructions are relatively slow on the SPARC; hence the greater difference between radix hashing and bit-wise hashing on the Small Web data.

Surprisingly, hash tables have performed relatively well on the Stopwords data set, which has a very small vocabulary; the reason may be that this data set has short strings. Less surprisingly, making the wrong choice of size of the hash table has an impact on either time or space. In TREC1, the hash table had about 1.4 slots per word, and gave good performance. In Small AP and Stopwords, the hash table was unnecessarily large, which, as discussed below, not only consumes space but slightly slows processing.

Bit-wise hashing is several times faster than all kinds of tree for all data sets, and uses less space than the kinds of tree tested. In this application, where maintaining search order amongst terms is not useful, hashing is clearly the method of choice.

Table 3: *Running time (seconds) for each collection, for hashing over a range of table sizes, with and without move-to-front, and for table-doubling.*

	Small AP	TREC1	Stopwords	Small Web	Large Web
Table size 8 Kb:					
Hashing	4.2	565.9	28.9	2420.4	47,676.0
MTF hashing	3.8	246.9	30.4	1352.7	21,267.0
Table size 64 Kb:					
Hashing	2.5	175.2	29.0	619.8	7164.0
MTF hashing	2.6	136.4	30.5	433.7	3908.0
Table size 512 Kb:					
Hashing	2.4	126.9	29.3	397.3	1971.6
MTF hashing	2.5	124.1	30.4	369.8	1559.6
Table size 4 Mb					
Hashing	2.5	123.9	30.2	404.1	1361.1
MTF hashing	2.6	126.1	31.1	399.7	1251.7
Table size 32 Mb					
Hashing	2.6	129.9	30.3	405.6	1310.1
MTF hashing	2.7	133.9	31.8	405.2	1255.9
Table doubling	3.5	168.4	43.0	424.3	1339.4
(Final table size, Mb)	(0.26)	(2.00)	(0.01)	(4.00)	(32.00)

Improving hashing

A potential drawback to hashing is the problem of choice of hash table size: too small, and search lengths are excessive; too great, and memory is wasted and the CPU cache becomes less effective.

Following the observation that the distribution of words is highly skew, the hash table size can be kept small with little impact on efficiency, as follows. Since, in the great majority of cases, each chain will contain at most one common word, if this word is kept at the front of the chain average access costs will be low. One way of detecting which words are common is to maintain a counter in each node and test at each access, but this would be costly and, indeed, is unnecessary. A much simpler approach is to simply move the accessed node to the front of the chain after each successful search. In most cases the node will not need to be moved, so only a single test is required, and a node can be moved with three assignments.

Experiments with move-to-front hashing are reported in Tables 3 and 4. Table 3 shows the CPU time for different table sizes, and, in the last block, time and size for dynamically-sized

Table 4: *Percentage of matches found at the start of the list and average number of string comparisons per search for each collection, for hashing over a range of table sizes, with and without move-to-front, and for table-doubling. Note that the average number of comparisons can be less than 1 if a significant proportion of searches are with new strings to an empty slot.*

	Small AP	TREC1	Stopwords	Small Web	Large Web
Table size 8 Kb:					
Hashing	57.7%, 3.39	50.9%, 10.87	96.9%, 1.03	47.9%, 14.33	43.1%, 66.57
MTF hashing	68.8%, 2.52	73.3%, 3.06	96.6%, 1.04	74.0%, 5.58	68.6%, 24.23
Table size 64 Kb:					
Hashing	82.3%, 1.28	75.3%, 2.27	99.7%, 1.00	71.6%, 2.69	67.8%, 9.19
MTF hashing	88.2%, 1.17	90.5%, 1.26	99.9%, 1.00	90.3%, 1.57	86.2%, 3.90
Table size 512 Kb:					
Hashing	95.0%, 1.02	91.1%, 1.17	100.0%, 1.00	89.0%, 1.22	84.9%, 2.03
MTF hashing	96.2%, 1.01	97.7%, 1.03	100.0%, 1.00	96.9%, 1.07	94.9%, 1.36
Table size 4 Mb					
Hashing	97.6%, 0.99	97.4%, 1.02	100.0%, 1.00	95.7%, 1.04	94.1%, 1.13
MTF hashing	97.7%, 0.98	99.3%, 1.02	100.0%, 1.00	98.9%, 1.01	98.1%, 1.04
Table size 32 Mb					
Hashing	97.9%, 0.98	99.3%, 1.00	100.0%, 1.00	98.9%, 1.00	97.7%, 1.02
MTF hashing	97.9%, 0.98	99.6%, 1.00	100.0%, 1.00	99.3%, 1.00	99.1%, 1.00
Table doubling	88.1%, 1.14	93.9%, 1.09	96.9%, 1.03	93.6%, 1.08	96.4%, 1.05

hash tables where the table is doubled when the load average reaches 2.0 strings per node. (This threshold seemed a good compromise; lower thresholds increased memory usage without significantly improving performance, while higher thresholds markedly reduced speed.) In this table, pointers require 4 bytes, so for example a 8 Kb table contains 2048 pointers; the first "4 Mb" row is drawn from Table 2. Table 4 shows the proportion of accesses that find the string at the front of the list, and the length of an average search.

As can be seen, the simple move-to-front heuristic can have a dramatic impact. For the Small Web data, for example, and a table of 16,384 pointers, average chain length is 81 nodes by the end of processing; yet total time is less than 10% worse than for a table of 1,048,576 pointers, where average chain length is about 1.3. For both table sizes, the correct word is found at the head of the chain in more than 90% of searches. Without move-to-front, processing is considerably slower. Common words—which account for the vast majority of word occurrences—tend to have their first occurrence close to the start of the file, and thus tend to occur towards the front of static chains, but even so, move-to-front yields significant, and sometimes dramatic, gains. For all data files, move-to-front with a table size of 16,384 slots gave much better performance than for trees, despite, in the case of the Large Web data, average chain length of around 500 nodes.

Moreover, move-to-front is usually faster than the dynamic approach, in which table size is doubled and all strings are rehashed after the load factor exceeds a threshold of 2.0. The reason for this is, perhaps, surprising: it appears that a significant additional cost is that the compiler cannot optimise the code as well with a variable table size. In these experiments we restricted table sizes to be powers of 2 so that bitmasks could be used in place of modulo (when reducing hash values to the table size); this optimisation yielded significant gains—the times with modulo were 30% greater—but, as can be seen, was not sufficient. The doubling approach also has the disadvantage that the hash table eventually consumes a significant portion of physical memory and rehashing is costly.

For small vocabularies and large tables, the extra test and occasional move-to-front (which disrupts the order in the chain) incurs a slight performance penalty. There is also a slight loss of performance due to poorer use of the CPU cache. However, even so, a fixed-size table without move-to-front tends to be faster than a dynamic table. As our results show, table size is not crucial at all; a rough guess at table size within a factor of 10 provides better performance than allowing table size to vary.

Several small optimisations contributed to the speed of all our programs. For example, we used table sizes of powers of 2 to allow simplification of arithmetic, and the standard `strcmp` function under both Solaris and Linux proved highly inefficient; replacing it with our own code yielded overall speed improvements of 20% or more.

Conclusions

We have investigated candidate data structures for managing large lexicons of English words. In our experiments with BSTs, splay trees, a heuristic splay tree, and two variants of hashing, we have found that hashing using an efficient bit-wise hash scheme is three to four times faster than other schemes.

We have proposed a new move-to-front hashing heuristic, in which an accessed word is relocated to the front of the hash table chain. This simple scheme improves speed remarkably. For small-to-medium size hash tables, move-to-front bit-wise hashing is almost twice as fast as bit-wise hashing. For larger tables—where hash table chains are on average much shorter—the speed improvement is less, but is still significant, and, for reasonable vocabulary sizes, move-to-front with a static table is also faster than dynamically doubling table size to maintain a constant load average, even if the table is only one-tenth of the size of the vocabulary.

Index construction schemes—which require efficient management of large lexicons—often use tree-based schemes. Our findings show that move-to-front bit-wise hashing is the method of choice for efficient index construction.

Acknowledgements

We thank Sam Kimberley. This work was supported by the Australian Research Council and the Multimedia Database Systems group at RMIT University.

References

[1] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, 5–7 January 1997.

[2] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.

[3] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In Y. Kambayashi, editor, *Proc. International Conference on Very Large Databases*, pages 294–303, Kyoto, Japan, August 1986.

[4] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[5] M.V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In *Proc. International Conf. on Database Systems for Advanced Applications*, pages 215–223, Melbourne, Australia, April 1997.

[6] R. Sedgewick. *Algorithms in C: Parts 1–4: Fundamentals, data structures, sorting, searching*. Addison-Wesley, Reading, MA, USA, 1998.

[7] H.E. Williams, J. Zobel, and S. Heinz. Splay trees in practice for large text collections. Technical report, RMIT University, 2000. (in submission).

[8] I.H. Witten and T.C. Bell. Source models for natural language text. *International Journal on Man Machine Studies*, 32:545–579, 1990.