

Performance of Data Structures for Small Sets of Strings

Steffen Heinz

Justin Zobel

School of Computer Science and Information Technology, RMIT University
GPO Box 2476V, Melbourne 3001, Australia
{sheinz,jz}@cs.rmit.edu.au

Abstract

Fundamental structures such as trees and hash tables are used for managing data in a huge variety of circumstances. Making the right choice of structure is essential to efficiency. In previous work we have explored the performance of a range of data structures—different forms of trees, tries, and hash tables—for the task of managing sets of millions of strings, and have developed new variants of each that are more efficient for this task than previous alternatives. In this paper we test the performance of the same data structures on small sets of strings, in the context of document processing for index construction. Our results show that the new structures, in particular our burst trie, are the most efficient choice for this task, thus demonstrating that they are suitable for managing sets of hundreds to millions of distinct strings, and for input of hundreds to billions of occurrences.

Keywords: data structures, binary search tree, splay tree, trie, burst trie, inverted index

1 Introduction

Data structures such as trees, tries, and hash tables are used in a vast variety of applications for managing sets of distinct strings. Which of these structures is chosen for a particular application depends on factors such as how well they scale, memory requirements, speed, and whether it is required that the strings be kept in sort order.

In previous work we have evaluated these structures on large sets of strings, comparing them on tasks associated with index construction for large text collections [11, 12, 20, 23]. Text collections used in practice range from news and law archives, business reports, and collections of email, to repositories of documents garnered from the world wide web. Such text collections can easily contain many millions of distinct words, and the number grows more or less linearly with collection size [19]. The collections themselves range from gigabytes to terabytes, consisting of millions of documents or more; at the upper extreme, the popular search engine **Google** indexes over a billion web pages. Practical management and querying of text collections relies on the use of efficient string data structures that offer fast insertion, search, and deletion of string keys.

Testing the structures on such large-scale data sets led to interesting discoveries. Surprisingly, on text collections from 100 Mb to 45 Gb, the relative speeds of the various structures were almost constant. Hash tables (assuming an efficient string hash-

ing function [15]) are the fastest of the structures, and we found that performance can be significantly improved through the simple strategy of using chains to manage collisions and move-to-front within chains. Trees are several times slower and, surprisingly, existing adaptive and balanced tree structures were no faster than standard binary trees. However, our novel variant of adaptive tree, the fab-tree, yielded significant improvements in performance. Tries require inordinate volumes of memory, but are fast. We developed a novel trie structure, the burst trie, that is as fast as a conventional trie and as compact as a tree—a dramatic increase in efficiency compared to previous trie structures. The underlying principle of the burst trie is that it stores, not individual strings, but small sets of strings that share a common prefix.

A question that naturally arose from this work is: which structures are best for *small* data sets? It might be argued that performance on a small data set is unimportant, but there are many applications in which the time taken to manage small sets of data is a major component of processing costs. One such example is, again, indexing of text collections. During the index construction process, it is necessary to identify the set of distinct words that occurs in each document [22], that is, undertake the task of *per-document vocabulary accumulation*. Most of the individual documents are small—a few hundred words or so—but nonetheless the task of collating these words in a dynamic structure is an important cost overall. More generally, in any application in which a small data structure is frequently consulted, inefficiency can significantly degrade performance of the application as a whole.

In this paper we experimentally explore performance of the data structures discussed above for the task of per-document vocabulary accumulation. Compared to the task of managing large sets of strings, different relative performance is to be expected: for example, over only a small number of words adaptive and balanced structures are unlikely to have the time to reach the equilibrium that in principle might give them an efficiency advantage. We present the results of experiments on a variety of sets of documents. Overall, burst tries are faster than trees, but by a smaller margin than was observed for larger data sets. Hash tables can be faster again, if sort order is not required, but—particularly in comparison to the results for large data sets—speed is critically dependent on hash table size. The slowest structure of all is the trie, a striking contrast to the efficiency of this structure on large data sets.

The focus of this work is on comparison of the speed of the structures and not their memory usage; none use significant volumes of memory for this application, and their memory requirements for large data sets are known from our previous work [12]. However, memory usage is measured indirectly: excessive use of memory leads to increased running time, since

high memory usage requires large memory management costs when processing of a document is complete. These costs, for traversing and freeing a data structure, are not relevant if a large text collection is processed as a whole. In per-document processing, however, these costs are significant.

2 Candidate data structures

There is a range of data structures that are suitable for tasks such as per-document vocabulary accumulation that involve management of small sets of strings. We discuss several choices including trees, tries, and hash tables. Those structures share one property: they store a record containing a string key and additional associated information depending on the task. For the task of per-document vocabulary accumulation, a record consists of a unique string representing a word, together with a counter for its frequency in the document.

There are several features of the input that affect the relative performance of different data structures for such applications. The first, and arguably the most important, is *skew*: the degree to which the commonest words are prevalent in the input. If a single word accounts for, say, half of all word occurrences, then a data structure that provides rapid access to this word—even at the cost of slower access to other words—will have an advantage. It is this characteristic that potentially benefits adaptive structures such as splay trees. On the other hand, if the distribution is more uniform, time spent modifying the structure may be wasted.

Skew in text is sometimes explained by reference to Zipf’s distribution, which, while highly inaccurate as a description of real text collections, does succinctly describe the phenomenon of common words dominating as a proportion of word occurrences [21]. For example, in the Wall Street Journal component of the first TREC disk [10], the word “the” accounts for around one word occurrence in 17, and the commonest five words (“the”, “of”, “to”, “a”, and “and” respectively) account for around one word occurrence in six.

A related characteristic is *locality*: the degree to which a recently-observed word is likely to occur again, independent of its overall frequency. For example, in a collection containing a chronological sequence of newspaper articles, a word such as “Lockerbie”, although rare overall, is common for a period of days or weeks. Within a document, locality of topic often means that a word is used several times within a paragraph, then never used again. Three other characteristics are also important: whether the input is sorted, how many distinct strings there are, and how long the strings tend to be (consider ordinary text versus legislation, for example). These characteristics are not directly explored in our experiments, but are responsible for some of the differences observed between data collections.

Theoretical analysis of a data structure can be used to suggest likely average and worst-case asymptotic costs, but, as these issues suggest, such analysis can only be indicative of performance in practice. Skew can be readily modelled, but assumptions must be made on the degree of skew. Locality is more difficult analytically. As the examples of common words show, there is a relationship between word length and word frequency—an important factor in performance given that string comparisons are usually the dominant computational cost for string data structures.

In our experiments, the theoretical behaviour of a structure was not generally a good indicator of relative performance. For example, structures such as randomised search trees [1] and splay trees [16]

achieve their logarithmic cost bounds by using node rotations to restructure the tree on each access. Rotations are computationally expensive, a cost that more than offsets the saving of lower average search lengths, and moreover have the disadvantage of occasionally introducing gross imbalance. While a simple binary search tree has a linear worst case, it is rarely observed in practice; Sleator and Tarjan [16] expected splay trees to be superior to binary search trees for skew data, but we have not observed this behaviour in practice, for any scale or data set. As another example, the relative speed of string comparisons and string hashing has a drastic effect on the relative speed of trees and hash tables; with the string hash functions described in textbooks, trees are vastly superior, whereas with our fast hash function the reverse is observed.

Search trees

A standard *binary search tree* (BST) stores in each of its nodes a key together with a pointer to each of the node’s two children. A query for a key q starts at the root; if q is not found in a node, a comparison between q and the node’s key determines which branch downwards the tree is to follow. An unsuccessful search ends at a leaf, where depending on the purpose of the query, a new node may be inserted.

The performance of a BST depends on the insertion order of keys. The worst case is of $O(n)$ for a BST with n nodes and occurs if the keys are inserted in sort order yielding to long sticks. In our large-scale practical experiments [12, 20, 23] on string data sets drawn from the Web with billions of string occurrences and millions of distinct strings, the performance of a BST was surprisingly good, although a small number of strings in sort order at the start of the data led to dramatic degeneration in performance. A BST should not be used in practice for this task.

Considering per-document processing of a text collection, we can reasonably assume that the vast majority of documents will not consist of strings in sort order, therefore average logarithmic search cost should dominate. The performance of a BST in this case depends much more on the characteristics of the input. If the collection is skew, as in text where a small percentage of the distinct strings account for a large proportion of the text [21], common keys are likely to be observed early in a document. They would then be stored in the upper tree levels, where only a few comparisons are needed to access them. The common strings tend to be short, further reducing total processing costs. Hence, a BST should be well-suited to the task of per-document vocabulary accumulation.

The cost of string comparisons during a tree traversal can be reduced as the search is narrowed: if the query string is known to be lexicographically greater than “international” but less than “internet” the first six characters can be ignored in subsequent comparisons. However, we have found in a range of experiments on large data sets that the additional tests and operations needed to identify how many characters to ignore cost significantly more than the savings, and we do not use this technique in any tree structure used in our experiments.

AVL trees [13] and *red-black trees* [6, 13] are variants of BSTs that reorganise the tree on insertion to maintain approximate balance. They achieve the logarithmic worst case bound by storing some additional information in each node, a single bit in a red-black tree and two bits in an AVL tree. AVL trees and red-black trees guarantee the absence of long sticks but due to the reorganisation commonly-accessed keys are not necessarily clustered at the top tree levels. On the

other hand, for a skew input—where most accesses are not insertions—the cost of balancing is kept low. The benefits of having a balanced tree structure in contrast to a BST could therefore offset the additional costs that incur at insertion time.

Splay trees are another adaptive variant of BST. They have an amortised average cost of $O(m \log n)$ over a sequence of m searches and insertions in a n -node tree [2, 16]. The tree is modified at each access, by rotating the accessed node to the root through a series of double rotations. A double rotation replaces a node with its grandparent and is efficiently implemented by storing an additional parent pointer in each node. We use non-recursive bottom-up splaying instead of the top-down splaying used in other work [2, 18], since we found that bottom-up splaying is more efficient [20]. Splaying leads to clustering of commonly-accessed nodes towards the root, a beneficial feature for skew input. However, we have found in other work [11, 20] that the potential low expected search costs are offset by the costly rotations. We evaluated a variant of the splay tree, the intermittent splay tree, which is splayed only on every k -th tree access [20]. We achieved with $k = 11$ time savings of more than 15% compared to the original splay tree.

Randomised search trees, like splay trees, are reorganised at each access [1]. The main difference is that each node has in addition to the search key a randomly-generated integer heap key, the heap key has a random chance of being increased on access, and the accessed node is rotated upward to maintain the heap property on the heap key. Based on the poor results in our earlier experiments [11], we do not test randomised search trees here.

Fab-trees

In our earlier work, we identified limitations in all the tree structures we tested. Using this analysis we proposed a new form of tree, the *frequency-adaptive binary search tree* (fab-tree) [11]. A node in a fab-tree includes left and right child pointers, a parent pointer (for efficient rotation), and additionally a counter. The counter is initialized to one when the node is created. When a node is accessed, the counter is incremented, then single rotations are used to maintain the heap property on the counter. That is, the series of rotations ends once a parent node with a higher frequency count is encountered.

Such an approach—the strategy used in the a monotonic tree [4]—has two drawbacks. First, the counters will eventually overflow, and second, the tree will become increasingly static, with no adaptation to locality. To address these issues, we occasionally reset the counters, in a strategy we called “hot set”. Whenever the root-node exceeds a threshold of b accesses, a tree traversal is used to visit all nodes with a counter of at least t , terminating at nodes whose counter value is smaller than t , and resets counters whose value is at least t by a right-shift of s bits. We observed the best performance for a hot-set fab-tree with $b = 65,535$, $t = 16$, and $s = 1$. The additional storage space needed for a frequency counter is then only two bytes, independent of the size of the input.

Figure 1 shows an example of a fab-tree before and after a reset caused by an access to “pisces”. The reset has not preserved the heap property; “krebs” has a smaller counter than “gorgon”. With subsequent accesses, the heap property is gradually restored. If “krebs” is not searched for again, it will be gradually demoted.

On a broad range of medium to large data sets that are skew we observed that, compared to the other tree structures, fab-trees are more effective at clustering commonly-accessed keys in the upper tree levels

and have lower average search lengths. Compared to intermittent splay trees, we achieved time savings of about 11% and in comparison to randomised search trees we achieved time savings of up to 35% [11].

Tries

Tries [7, 8, 13] can be used for fast management of strings. A node in a standard trie can be represented by an array of pointers, where each pointer corresponds to a letter in the alphabet including an additional end of string character. A leaf of a trie is a record that corresponds to a particular string. A query for a string $c_1 \dots c_k$ of length k starts at the root of the trie. At each node on level l we follow the array pointer that corresponds to the l th character of the query string. Search in a trie is fast, since it requires only a single character comparison and pointer traversal for each letter in the query string. In a *compact trie*, nodes that lead to a single leaf are omitted, as suggested by Sussman [17].

Compact tries, and variants such as Patricia tries [9, 14], are fast but space-intensive. For this reason, alternatives have been developed, such as the ternary tree [3, 5], where trees rather than arrays are used to represent nodes. However, we have found that ternary trees are neither as fast nor as compact as the structure described below [12], and we do not report them in this paper.

Burst tries

In recent work [12] we have proposed a new and efficient trie variant, the *burst trie*, that dramatically outperforms other tree and trie structures for the task of accumulating the vocabulary of a large text collection. A burst trie consists of a standard trie, called the access trie, which is used to organise accesses to leaf nodes, and the leaf nodes themselves. In contrast to a compact trie, where a leaf node contains only one record, a leaf node in a burst trie is a container or data structure that is able to store up to c records.

Searches for a query key q involve trie traversal starting from the rootnode through the access trie until a container is reached. At that stage, the prefix characters of q are matched by the access trie nodes on the traversal path. Then, the inherent search routine of the container structure is used to search for the remaining characters of q . Hence, a container stores strings that share the same prefix; the prefix need not be stored, yielding space savings. If a container reaches the threshold of c records, a new trie node is allocated and replaces the container, which is said to be burst. The contents of the container are distributed among the empty containers of the new trie node. A burst trie with $c = 4$ is illustrated in Figure 2, before and after a burst triggered by the insertion of “bands”.

In our earlier work [12] we achieved the best results with $c = 35$ and BSTs as containers. A burst trie is often faster than a compact trie, but has memory requirements close to that of trees and hash tables.

Hash tables

In previous work, we found that a *chained hash table* using a bit-wise hash function is the most efficient data structure for the task of accumulating the vocabulary of large text collections, assuming that strings do not need to be maintained in sort order [12, 23]. In such a hash table, an array is used to index a set of linked lists of nodes, each of which is said to be a slot of the hash table. For a search for string q , a hash value is computed for it, to determine which slot

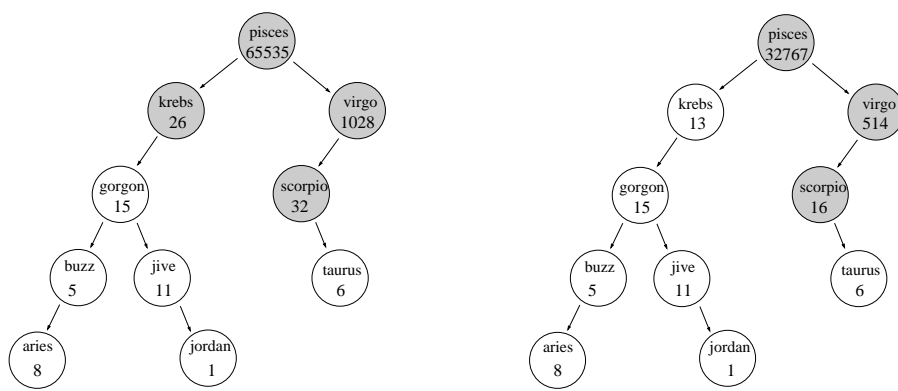


Figure 1: A *fab-tree* with $b = 65,535$, $t = 16$, and $s = 1$. Each node contains a counter. The gray nodes contain the current hot set—the nodes whose counts exceed t . On the left, the tree prior to reset. On the right, after reset caused by access to “*pisces*”.

is to be searched. A feature that influences the performance of our hash tables is the use of a fast string hashing function based on bit-wise operations [15]; this function is 10 to 30 times faster than the uniform string hashing functions in earlier literature.

To further improve the performance of hash tables we proposed use of a move-to-front policy in the linked lists [23], bringing the most recently accessed node in a list to the front. For a skew distribution of strings keys, commonly-accessed strings accumulate at the front of the linked lists, and the vast majority of searches require only one string comparison, even if there are on average dozens of records stored at each slot. The size of the hash table can be significantly reduced without affecting speed, a property that does not hold for chained hashing without a move-to-front policy [23]. We note that the bit-wise hash function was designed for strings drawn from ordinary text collections and not other data, for example fixed-length substrings of genome data. For such data, the set of possible strings is fixed and a perfect hash function can be used.

In contrast to the data structures discussed so far, strings in a hash table are usually not kept in sort order. This is potentially disadvantageous for per-document processing during index construction, where it is necessary to merge the per-document data with collection-wide data. With sorting, the access path is stable and cached, yielding lower access costs. To make a full comparison between the data structures, we report results on hash tables with a table size of 1024, 4096, and 16,384 slots respectively, showing times with and without sorting after accumulation of per-document data. The sort involves copying references to all list nodes into an array and sorting with a standard quicksort package.

3 Test data and methodology

Several text collections taken from the large Web track in the TREC project [10] are used in our experiments. The file *TREC1* contains the data on the first TREC CD-ROM; the three files *Web S*, *Web M*, and *Web L* contain collections of web pages extracted from the Internet Archive used in the TREC experiments. We also included the file *Gen* of n -grams of genome data, which, in contrast to the text data, shows little skew. The statistics of these collections are shown in Table 1.

A document in *Gen* contains a nucleotide string, typically thousands of nucleotides in length. Parsing involves extracting n -grams of length 9 and substituting wildcards with nucleotides, reduces the alphabet size of the collection to four. Parsing the other

collections involves extracting alphanumeric strings that contain no more than two digits and do not begin with a digit. Case-folding is not applied. XML and HTML markup, special characters, and punctuation are skipped. The fanout of the trie nodes in the compact trie and in the burst trie is 76 for the experiments on web data. However, not all of the slots are used due to our parsing restrictions. For the genome data, the fanout is 5, since there are four different nucleotides symbols and the end of string symbol to accommodate.

In Table 2 we show results for the task of overall vocabulary accumulation on these data sets, where for each distinct word its frequency and the number of documents it occurs in is determined. These experiments essentially repeat those reported in our earlier work [11, 12, 20, 23], but consolidate our findings and are on the same hardware as the new results discussed below. As this table shows, for text data burst tries use only a little more memory than trees, while running more than twice as fast, and are faster and much smaller than than a compact trie. Hash tables are more efficient, but do not maintain strings in sort order. Of the tree structures, the *fab-tree* is the fastest. In contrast to our earlier experiments, in which we used case-folding, in these runs case-folding was not applied during parsing, leading to slightly different relativities in speed. Genomic data is in many respects a worst case for an adaptive or unbalanced structure, and thus reveals very different relative behaviour. In particular, the compact trie is very efficient, due to the occupancy of the space of the possible strings—every single possible 9-gram occurs in our data, so every node is fully utilised.

In all our experiments we report CPU times only, not elapsed times. We exclude the parsing time, since it is the same for all structures. Prior to each experimental run the internal buffers of our machine, an AMD Athlon 1.2 GHz with 512 Mb of main memory, were flushed to ensure a non-biased test environment. The results are over single runs; we have observed almost no variance between times over multiple runs, so reporting the average times is not necessary.

The task in each experiment is per-document vocabulary accumulation and involves determining for each document its unique set of strings and for each string the frequency. Prior to processing a new document, the data structure is traversed, emptied and freed. In the case of hash tables, the hash table itself is not freed but reset, that is, the slots are marked as empty.

To compare the performance of the structure on larger sets, we made additional runs where every 100 documents the structures are emptied. Hence, we treat every sequence of 100 documents in our collec-

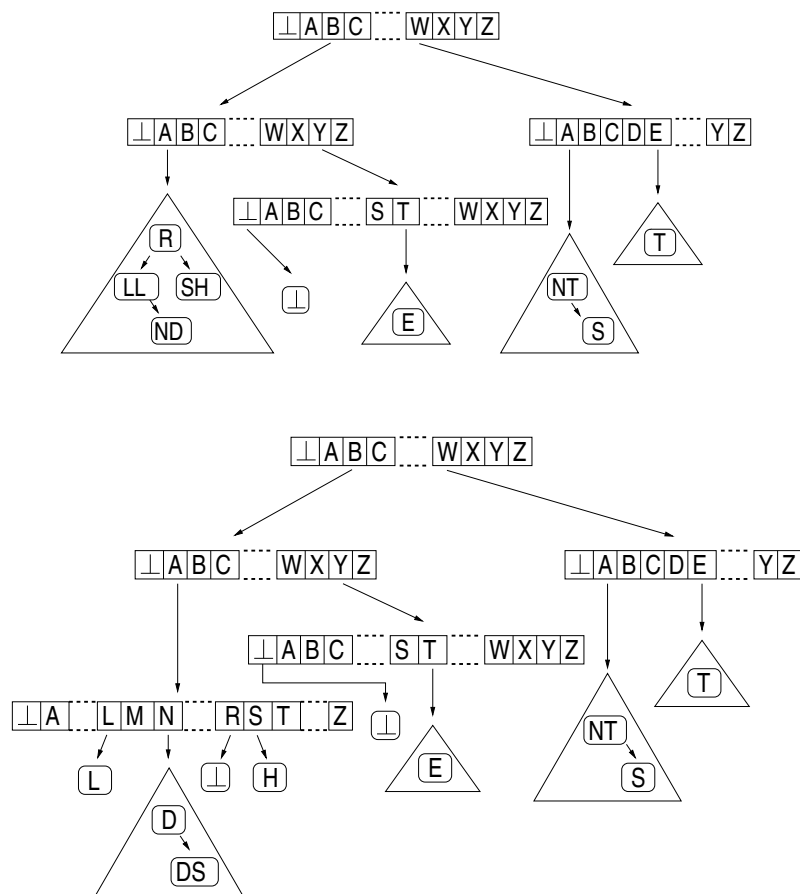


Figure 2: Above, a burst-trie storing nine strings in five containers: “ball”, “band”, “bar”, “bash”, “by”, “byte”, “want”, “was”, and “wet”. Below, inserting “bands” led to a burst of the left-most container. Its strings are distributed among the containers of a new access trie node.

tions as one logical document.

We allocate space for the strings block-wise and maintain the blocks with a size of 5 Kb in a linked list, to reduce the cost of invoking the system call used for memory allocation. Hence, the freeing of strings between documents is simplified; only one call is needed to free each allocated memory block, an improvement that assists all structures. We noticed that the standard `strcmp` function in C is highly inefficient. In our earlier work, by replacing the standard `strcmp` function with our own code we achieved speed gains of 20% or more in total processing time including all data structure maintenance and traversal.

4 Experimental results

Table 3 shows results for the string data structures earlier on the standard text collections. We discuss the results on the Web and TREC collections first. Among the tree structures that are variants of the BST, the intermittent splay tree outperforms the other structures. The intermittent splay tree improves on the original splay tree by up to 30%, indicating that splaying on each access is too expensive. Performing no rotations at all yields a similar performance: a BST is as fast as an intermittent splay tree, despite the fact that some of the individual documents present a poor case for the BST. Balancing a tree on insertion, as is done by the red-black tree, does not give significant gains compared to a BST.

The fab-tree outperforms the original splay tree, but is around 6% slower than the intermittent splay tree or the BST. The overhead of rotations and maintaining the counters is not an advantage for small sets of strings, in contrast to the results for large sets of

strings shown above and in our previous work [11]. The strengths of this structure are the ordering of the nodes by frequency and the adaptivity to changes in distribution; in a single document, insufficient data is observed to allow accumulation of good statistics, and adaptivity gives no advantage.

Compared to the performance on large sets of strings, the trie is surprisingly inefficient, at more than 40% slower than a BST on the text collections. The reasons are the more complex memory management and the nature of the trie itself. Even a small number of strings share common prefixes and require the introduction of trie levels that are only sparsely populated. Initialisation of a trie node requires looping over all slots and, at the end of processing a document, it is necessary to check every one of the 76 array slots in each trie node. That makes the traversal of a compact trie more expensive than a traversal of a tree and greatly influences overall running time.

The burst trie was clearly the best of the tree and trie structures. Compared to a BST, we observed a decrease in running time of around 20% on all text collections. The burst trie does not suffer from the introduction of a large number of access trie nodes, in contrast to the compact trie, but provides fast accesses to strings stored in containers; the BST containers, each storing at most 35 strings, can be rapidly emptied or traversed. The gain in performance for the burst trie is not nearly as great as for large sets of strings, but is still significant, and shows that on all scales it is the fastest structure for this task.

The performance of hash tables is highly dependent on table size and on whether at the end of processing a document the strings should be in sort order. We showed in earlier work that a move-to-front policy makes the performance of a hash table more resilient

Table 1: *Statistics of text collections and a collection of genome data.*

	Web S	Web M	Web L	TREC1	Gen
Size (Mb)	100	2,048	10,240	1,206	300
Distinct words	147,278	1,516,782	4,484,223	627,669	262,144
Word occurrences ($\times 10^6$)	6	131	631	179	283
Documents	18,726	347,075	1,780,983	510,634	447,780
Avg. word occs. per doc	332	377	354	350	631
Avg. distinct words per doc	151	160	153	167	519
Longest doc (word occs.)	31,933	71,781	253,426	384,627	378,977
Shortest doc (word occs.)	16	10	9	1	6
Longest doc (distinct words)	4,858	18,129	18,129	14,881	164,839
Shortest doc (distinct words)	14	8	7	1	1
Parsing time (sec)	1.48	28.05	132.55	36.91	58.94

Table 2: *Running time in seconds for each data structure on the data sets, for the task of overall vocabulary accumulation. In parentheses, memory usage in Mb. One experiment could not be run on our hardware due to large memory requirements and is marked with “—”.*

	Web S	Web M	Web L	TREC1	Gen
BST	5.7 (4.5)	162.2 (48.5)	838.7 (145.0)	159.5 (19.9)	1,123.4 (8.5)
Red-black tree	7.0 (5.7)	229.1 (60.1)	1,279.5 (179.2)	246.8 (24.7)	1,066.7 (10.5)
Splay-tree	6.9 (5.1)	222.5 (54.3)	1,159.4 (162.1)	236.6 (22.3)	1,926.5 (9.5)
Splay-tree, intermittent	6.1 (5.1)	188.3 (54.3)	977.7 (162.1)	182.5 (22.3)	1,657.2 (9.5)
Fab-tree	4.9 (5.7)	148.7 (60.1)	769.7 (179.2)	144.4 (24.7)	1,484.5 (10.5)
Compact trie	3.2 (57.1)	89.7 (586.0)	—	105.8 (255.0)	338.8 (7.7)
Burst trie	3.1 (5.7)	88.2 (62.8)	477.2 (185.0)	93.2 (25.2)	651.2 (7.3)
Hashing 2^{20} slots, sorted	2.6 (8.0)	59.2 (46.7)	290.8 (131.9)	61.5 (21.5)	246.1 (11.5)

when table size is too small [23]. However, the results for hashing small set of strings show that only the smaller table sizes (1,024 or 4,096 slots) yield running times that are faster than a burst trie, and this only under the condition that no sorting of the vocabulary is required. With sorting, running times are considerably slower than most of the other structures. Even for applications where sorting is not required, selecting the right table size is a matter of guesswork.

For collection *Gen*, where the distribution of n-grams is not as skew as in web text documents, the alphabet contains only four characters plus an additional end-of-string character, and the strings are of fixed length, the compact trie shows the best performance of the sorted structures. For genomic data the trie nodes are small and access to strings is fast. Compared to a BST, the best tree structure on the genomic data, the running time of a compact trie is around 24% faster. The burst trie shows similar performance to the BST, while the fab-tree is particularly disappointing. Hashing with the larger table size of 16,384 slots gave the fastest running time, but as before sorting the hash table greatly degrades performance.

The results discussed above are potentially sensitive to average document length. To experiment with the impact of document length we re-ran the experiments after coalescing each sequence of 100 documents into a single document, thus creating collections in which the number of documents is reduced by a factor of 100 but the documents are 100 times as long. Results are shown in Table 4. There are marked changes in relative performance. Fab-trees are in this case the fastest of the tree structures, the burst trie is now over 30% faster than a BST, while the compact trie has gotten considerably worse. The average time per document has increased for all tree and trie structures, presumably due to the increased depths. The average time for unsorted hashing has decreased, due to reduced memory management costs, and is much

faster than the alternatives; however, the larger table gave better performance than the smaller. Sorting at end-of-document still degrades performance, but in this case only the burst trie is faster than hashing with sorting. On the genomic data, only the compact trie has outperformed the burst trie.

The relationship between the results in Tables 3 and 4 is illustrated in Table 5, which shows for each data structure and data set the ratio between the times in the two tables. These ratios indicate how average costs change for the different data structures as the data size increases. For the tree structures, the averages rise because the trees become deeper; the rise in cost for the burst trie is markedly less than for the other structures. For the hash tables, the averages fall because the costs of post-document traversal of the table and of memory management are amortised, while access costs do not change.

5 Conclusions

We have investigated the efficiency of a wide range of simple data structures for the task of managing small sets of strings. Our focus has been on text data, which is typically skew, but we have also reported results for genomic data, which has very different characteristics: it is not skew, there is little locality, and the strings are of fixed length.

Our current work complements our earlier exploration of data structure performance for management of large sets of strings, in which we identified inadequacies in all three of the major approaches. For hashing, the previous string hash functions were either too slow, non-uniform, or could only be used for a very limited range of table sizes, and table size must be chosen carefully; we therefore developed new hashing functions and a chaining strategy that addressed these issues [15, 23]. For trees, BSTs have an unacceptable worst case, but adaptive structures are hampered by the cost of rotations and occasional

Table 3: *Running time in $ms \times 10^{-1}$ per document for each data structure on the data sets, for the task of per-document vocabulary accumulation.*

	Web S	Web M	Web L	TREC1	Gen
BST	1.330	1.528	1.447	1.421	6.116
Red-black tree	1.282	1.578	1.469	1.437	6.372
Splay-tree	1.677	2.022	1.880	1.919	7.996
Splay-tree, intermittent	1.287	1.535	1.437	1.418	6.748
Fab-tree	1.298	1.584	1.504	1.506	8.014
Compact Trie	2.088	2.603	2.480	2.328	4.697
Burst Trie	1.004	1.199	1.163	1.132	6.162
Hashing 1,024 slots	0.748	0.902	0.857	0.914	9.950
Hashing 4,096 slots	0.908	1.044	1.007	1.034	4.768
Hashing 16,384 slots	1.538	1.755	1.737	1.760	4.128
Hashing 1,024 slots, sorted	1.586	1.754	1.695	1.766	15.693
Hashing 4,096 slots, sorted	1.938	2.183	2.107	2.154	10.746
Hashing 16,384 slots, sorted	3.861	4.072	3.977	4.065	11.332

Table 4: *Running times in $ms \times 10^{-1}$ per original document for each data structure on the data sets, for the task of per-document vocabulary accumulation, with artificial documents consisting of 100 consecutive documents each.*

	Web S	Web M	Web L	TREC1	Gen
BST	1.602	2.099	1.990	1.818	17.161
Red-black tree	1.415	2.607	2.033	1.802	17.893
Splay-tree	1.880	2.584	2.428	2.328	23.371
Splay-tree, intermittent	1.463	2.018	1.937	1.791	21.693
Fab-tree	1.474	2.089	1.942	1.721	22.941
Compact trie	2.318	3.505	3.325	3.163	9.028
Burst trie	0.977	1.377	1.308	1.176	13.424
Hashing 1,024 slots	0.635	1.074	1.020	0.881	75.900
Hashing 4,096 slots	0.603	0.920	0.863	0.767	23.587
Hashing 16,384 slots	0.507	0.891	0.843	0.763	10.147
Hashing 1,024 slots, sorted	0.988	1.815	1.701	1.383	89.549
Hashing 4,096 slots, sorted	0.988	1.635	1.558	1.284	37.084
Hashing 16,384 slots, sorted	1.089	1.700	1.602	1.315	23.618

excessive search lengths; in our fab-tree, these shortcomings are mitigated [11, 20]. For tries, which are fast but space-intensive, the memory requirements are impractical; our burst trie provides greater speed at a small fraction of the memory cost [12]

The results in this paper are further evidence of the efficiency of our new data structures. Of the sorted structures, burst tries are the most efficient. They require a little more memory than do trees, but are faster at all scales. Of the trees, fab-trees are faster for all but the smallest data sets, where they are still competitive. Across all the structures, hash tables with move-to-front chaining and efficient hash functions are the fastest and most compact.

Another result is the poor performance of structures that have been specifically proposed for such applications. Splay trees are less efficient than the more primitive alternative of a simple BST. A compact trie is not useful at any scale: for large sets of strings it is too space-intensive and for small sets of strings it is much the slowest structure; only for the genomic data does the trie show acceptable performance. These observations are in contrast to common wisdom as to the relative merits of such data structures. In contrast, our new data structures are the most efficient for many aspects of text processing, for which they should now be regarded as the preferred method for managing string data.

Acknowledgements

This work was supported by the Australian Research Council. We thank Hugh Williams.

References

- [1] C. R. Aragon and A. Seidel. Randomized search trees. *Algorithmica*, 16:464–497, 1996.
- [2] J. Bell and G.K. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software—Practice and Experience*, 23(4):369–382, 1993.
- [3] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, 1997. ACM/SIAM.
- [4] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM Jour. of Computing*, 8(1):82–110, 1979.
- [5] 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.
- [6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Massachusetts, 1990.
- [7] R. de la Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, volume 15, Montvale, NJ, USA, 1959. AFIPS Press.

Table 5: *Ratio of the corresponding values given in Table 4 and Table 3 showing the change in average running time per document. Values greater than one indicate slower average processing time per document.*

	Web S	Web M	Web L	TREC1	Gen
BST	1.20	1.37	1.38	1.28	2.81
Red-black tree	1.10	1.65	1.38	1.25	2.81
Splay-tree	1.12	1.28	1.29	1.21	2.92
Splay-tree, intermittent	1.14	1.31	1.35	1.26	3.21
Fab-tree	1.14	1.32	1.29	1.14	2.86
Compact trie	1.11	1.35	1.34	1.36	1.92
Burst trie	0.97	1.15	1.12	1.04	2.18
Hashing 1,024 slots	0.85	1.19	1.19	0.96	7.63
Hashing 4,096 slots	0.66	0.88	0.86	0.74	4.95
Hashing 16,384 slots	0.33	0.51	0.49	0.43	2.46
Hashing 1,024 slots, sorted	0.62	1.03	1.00	0.78	5.71
Hashing 4,096 slots, sorted	0.51	0.75	0.74	0.60	3.45
Hashing 16,384 slots, sorted	0.28	0.42	0.40	0.32	2.08

- [8] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.
- [9] G. Gonnet. *Handbook of Algorithms and Data structures*. Addison-Wesley, Reading, Massachusetts, 1984.
- [10] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.
- [11] S. Heinz and J. Zobel. Frequency-adaptive binary search trees. In submission, 2001.
- [12] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. In submission, 2001.
- [13] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, Massachusetts, 1973.
- [14] D. R. Morrison. Patricia: a practical algorithm to retrieve information coded in alphanumeric. *Jour. of the ACM*, 15(4):514–534, 1968.
- [15] 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.
- [16] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Jour. of the ACM*, 32:652–686, 1985.
- [17] E. Sussenguth. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963.
- [18] M.A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley, New York, 1997.
- [19] H. E. Williams and J. Zobel. Searchable words on the web. In submission, 2001.
- [20] H. E. Williams, J. Zobel, and S. Heinz. Splay trees in practice for large text collections. *Software—Practice and Experience*. To appear.
- [21] I. H. Witten and T. C. Bell. Source models for natural language text. *Int. Jour. on Man Machine Studies*, 32:545–579, 1990.
- [22] 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.
- [23] J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*. To appear.