Burst Tries: A Fast, Efficient Data Structure for String Keys

Steffen Heinz Justin Zobel* Hugh E. Williams
School of Computer Science and Information Technology, RMIT University
GPO Box 2476V, Melbourne 3001, Australia
{sheinz,jz,hugh}@cs.rmit.edu.au

Abstract

Many applications depend on efficient management of large sets of distinct strings in memory. For example, during index construction for text databases a record is held for each distinct word in the text, containing the word itself and information such as counters. We propose a new data structure, the burst trie, that has significant advantages over existing options for such applications: it requires no more memory than a binary tree; it is as fast as a trie; and, while not as fast as a hash table, a burst trie maintains the strings in sorted or near-sorted order. In this paper we describe burst tries and explore the parameters that govern their performance. We experimentally determine good choices of parameters, and compare burst tries to other structures used for the same task, with a variety of data sets. These experiments show that the burst trie is particularly effective for the skewed frequency distributions common in text collections, and dramatically outperforms all other data structures for the task of managing strings while maintaining sort order.

Keywords Tries, binary trees, splay trees, string data structures, text databases.

1 Introduction

Many computing applications involve management of large sets of distinct strings. For example, vast quantities of information are held as text documents, ranging from news archives, law collections, and business reports to repositories of documents gathered from the web. These collections contain many millions of distinct words, the number growing more or less linearly with collection size [52]. Other applications also involve large numbers of distinct strings, such as bibliographic databases and indexes for genomic data [51].

Not only must these sets of strings be managed, but, in applications such as indexing, for efficiency reasons the complete set should be held in memory [55]. Construction of an inverted index involves the determination of the set of distinct terms—that is, the vocabulary of the collection—and, for each term, maintenance of information such as its total occurrence count and the set of locations at which it occurs. To gather these statistics the collection is parsed

*Justin Zobel is the contact author for all correspondence.

into terms and an in-memory data structure is used to store all distinct terms together with their statistics. It is this task, of *vocabulary accumulation*, that we focus on in this paper, as an example of an application of data structures for string management.

There are several existing data structures that can be used for the task of vocabulary accumulation or, more generally, to maintain in memory a set of records that have a string as a key. Of these, for large data sets the most efficient is hashing. Standard methods for hashing strings and maintaining a chained hash table are not particularly fast; however, in other work we have shown that, with use of a bitwise hash function [39], chaining, and move-to-front in chains [53], hash tables are much faster than the alternatives. The disadvantage of hashing is that the strings are randomly distributed among slots; while this problem is not significant for bulk insertion or creation of a static index, it means that hashing is not a candidate structure for other applications, such as index maintenance, in which it is convenient for strings to be available in sorted order. Having strings sorted allows them to be browsed, and allows the index information for strings with shared prefixes (which for efficient range query evaluation should be kept together on disk) to be updated with minimal disk activity.

The other candidate structures are varieties of search tree that maintain strings in sorted order. Tries [4, 16, 22, 29, 35] and ternary search trees [10, 12] are fast, but space-intensive. Binary search trees and the major variants—splay trees [8, 45], randomised search trees [30], and red-black trees—use much less space, but are slow.

In this paper we propose a new data structure, which we call the *burst trie*. This structure is a collection of small data structures, which we call *containers*, that are accessed via a conventional trie, which in this context we call an *access trie*. Searching involves using the first few characters of a query string to identify a particular container, then using the remainder of the query string to find a record in the container. A container can be any data structure that is reasonably efficient for small sets of strings, such as a list or a binary search tree. The general principle for maintaining a burst trie is that, initially, it consists of a single container. When a container is deemed to be inefficient, it is *burst*, that is, replaced by a trie node and a set of child containers which between them partition the original container's strings. Thus there are two major design decisions to be explored for burst tries: what data structure to use for a container; and how to determine when a container should be burst. We experiment with several data structures for containers, and propose and test heuristics for determining when a container should be burst. Some of these heuristics are based on the assumption that string data sets are typically skewed and—in a lesson learned from comparing hashing to trees [53]—that a major obstacle to efficiency is excessive numbers of string comparisons.

Our experiments, over several data sets, compare burst tries to existing data structures for vocabulary accumulation. As our experiments show, a burst trie is over twice as fast as a conventional tree, only 25% slower than a hash table, uses no more space than a conventional tree or hash table, and maintains strings in sort order. Compared to a splay tree, the fastest variant of a burst trie can accumulate the vocabulary of a 10 Gb collection of web data in less than 40% of the time, while using no more memory. Compared to a ternary search tree, the burst trie is around 25% faster and uses only 50% of the memory. Analysing the superior performance of the burst trie in comparison to other structures, we show that the ability of a burst trie to store frequently-accessed records in locations that are close to the root of containers far outperforms the adaptivity of a splay tree. These results are consistent with theoretical analysis of similar structures [27, 48], which shows that logarithmic height bounds, and therefore good search

performance, are expected for burst tries. Overall, the burst trie is significantly superior to all existing tree structures for tasks such as vocabulary accumulation.

2 Existing data structures

Binary search trees

In a *binary search tree* (BST), each tree node stores a string and two pointers to left and right child nodes. A search to find a query string involves, at each node, comparing the query string to the node's string to determine whether the string has been found or, otherwise, whether to branch left or right. At the root, the string comparison typically terminates after inspection of a single character; as the search progresses, the number of characters inspected at each string comparison gradually increases.

Although the allocation of strings to nodes is determined by the insertion order, for a skew distribution it is reasonable to expect that common words occur close to the beginning of the text collection and are therefore close to the root of the BST. Assuming the distribution is stable, accesses to a common term should be fast, since the first levels of the tree are usually kept in cache and only a few string comparisons are required. On the other hand, if strings are inserted in sorted order (thus creating "sticks") or the distribution changes (as can happen, for example, if the documents to be indexed are in a variety of languages), the behaviour of a BST can be extremely poor. In the absence of a rebalancing technique to prevent sticks from occurring or to move common nodes towards the root, a BST is of limited use in practice for vocabulary accumulation, despite, as in our earlier work, good performance on typical data [53]. However, we include BSTs in our discussion, since they are a common data structure and serve as a yardstick in our experiments.

There are several well-known variants of BSTs that maintain balance or approximate balance, in particular AVL trees and red-black trees. With these techniques, the tree is reorganised on insertion or deletion, thus ensuring that leaves are at approximately the same depth. On the one hand, use of rebalancing ensures that for n nodes there is an $O(\log n)$ upper limit to the length of a search path. On the other hand, the rebalancing does not consider the frequency of access to each node, so common words can be placed in leaves. We have experimented with red-black trees [53], and observed that the overall effect is detrimental, with red-black trees slightly faster only for data with low skew, and standard BSTs faster for typical vocabulary accumulation. We do not report on experiments with red-black trees in this paper.

In principle, the cost of searching a BST can be reduced by skipping string prefixes. As the search proceeds, the upper and lower bounds on the search narrow, and the characters shared by these bounds do not need to be checked. However, we have found that the cost of identifying how much of the prefix to skip outweighs any savings.

Splay trees

A *splay tree* [45] is a variant of a BST, in which, on each search, the node accessed is moved to the root by a series of node rotations, an operation known as *splaying*. Splaying has several effects that would be expected to be beneficial for vocabulary accumulation. Intuitively, commonly-accessed nodes should remain near the root (and in the CPU cache), thus allowing them to be

accessed rapidly; the tree quickly adapts to local changes in vocabulary; and the use of splaying guarantees that the amortised cost of accessing a tree of n nodes is at most $O(\log n)$.

In practice, however, splay trees have significant disadvantages [53]. In comparison to a BST, a splay tree requires more memory, since an efficient implementation of splaying requires that each node have a pointer to its parent. Also, even a common word such as "the" can be moved surprisingly far from the root between searches for it; even in data in which it is every seventeenth word or so, it is often found deep in the tree. Moreover, splaying is a rather expensive reorganisation technique, with around three term comparisons and six pointer assignments at each level. We investigated variations of splaying [53] and found that the most efficient is to only splay after every after k accesses, with say $k = 11$, thus moving common words close to the root but reducing the total cost of reorganisation. We report experiments with both splaying at all accesses and intermittent splaying with an interval of 4 accesses, as this value of k worked well on our data.

Another form of BST that reorganises on each access is the randomised search tree (RST) [30], in which each node uses an additional number r, initially zero. RSTs are treaps, in which an inorder traversal is in sort order for strings and the heap property is maintained by the numbers r, so the number in a node is at least as large as that of both its children. At each search, a positive random number r' is generated; if $r < r'$ for the node accessed, then r is replaced by r', otherwise r is left unchanged. The tree is then reorganised using rotations to restore the treap property. We have found that RSTs provide similar performance to splay trees [53], and do not report experiments with them in this paper.

Hash tables

The most efficient form of *hash table* for vocabulary accumulation is based on chaining [56]. In such hash tables, a large array is used to index a set of linked lists of nodes, each of which is said to be at a slot. On search, an array index is computed by hashing the query string. The string is then sought for in the linked list for that index.

Several factors are important for good performance. First, the hash function needs to achieve a reasonably uniform distribution of strings to slots. In practice, this means that the order of slots does not correspond to string sort order, so once the vocabulary has been accumulated the distinct strings must be sorted—a small overhead only. Second, the hashing needs to be fast. The functions commonly described in texts, based on modulo and repeated multiplication and division, are simply too slow; for this reason, we developed a fast string hashing function based on bit-wise operations [39]. Last, the hash table needs to be adequately large. For standard chaining, the number of slots needs to be a significant fraction of the number of distinct strings. However, in other work we have found that if, on each search, the accessed node is moved to the front of the list, the same efficiency can be obtained with much smaller tables [56]; and, most importantly for this application, with move-to-front chains efficiency declines much more slowly with increasing vocabulary size.

A lower bound on the cost of hashing is that, per search, a string inspection is required for the hash function and a comparison is required between the query string and the string in the first node in the slot. With move-to-front chains and vocabulary accumulation, over 99% of searches terminate at the first node in the slot. It is for this reason—the small number of total string inspections required—that, as can be seen in our experiments, hashing is the fastest of

all the methods we tested.

Tries and ternary search trees

A *trie* is an alternative to a BST for storing strings in sort order [4, 16, 22, 29, 35]. A node in a standard trie is an array of pointers, one for each letter in the alphabet, with an additional pointer for the empty string. A leaf is a record concerning a particular string. A string is found in a trie by using the letters of the string to determine which pointers to follow. For example, if the alphabet is the letters from 'a' to 'z', the first pointer in the root corresponds to the letter 'a'; the node T indicated by this pointer is for all strings beginning "a-". In node T, the pointer corresponding to the letter 'c' is followed for all strings beginning "ac-"; the pointer corresponding to the empty string is for the record concerning the single-letter string "a".

Search in a trie is fast, requiring only a single pointer traversal for each letter in the query string. That is, the search cost is bounded by the length of the query string. With a small increase in complexity of implementation, a trie can be substantially reduced in size by omitting nodes that lead to a single leaf [46]. In such a trie, chains that lead to a leaf without branching are eliminated; we refer to this variant as a *compact* trie. However, even so, compact tries are extremely large, in particular because nodes that are close to the leaves tend to be sparse, with only a few non-null pointers each.

There are several variant forms of tries with reduced space requirements compared to standard tries, such as *ternary search trees* (TSTs) [10, 12] and compact tries. In a TST, each node represents a single character c, and has three pointers. The left (respectively, right) pointer is for strings that start with a character that alphabetically precedes (respectively, follows) c. Thus a set of TST nodes connected by left and right pointers are a representation of a trie node. These can be rebalanced on access. The central pointer is for strings starting with c, thus corresponding to the 'c' pointer of a trie node. TSTs are slower than tries, but more compact. We report experiments with compact tries and TSTs below. Tries are discussed in detail in Section 6.

3 Burst tries

In earlier work comparing the tree data structures discussed above [56], we observed that, compared to hashing, these structures had three sources of inefficiency. First, the average search lengths were surprisingly high, typically exceeding ten pointer traversals and string comparisons even on moderate-sized data sets with highly skew distributions. In contrast, a search under hashing rarely requires more than a string traversal to compute a hash value and a single successful comparison. Second, for structures based on BSTs, the string comparisons involved redundant character inspections, and were thus unnecessarily expensive. For example, given the query string "middle" and given that, during search, "michael" and "midfield" have been encountered, it is clear that all subsequent strings inspected must begin with the prefix "mi". Third, in tries the set of strings in a subtrie tends to have a highly skew distribution: typically the vast majority of accesses to a subtrie are to find one particular string. Thus use of a highly time-efficient, space-intensive structure for the remaining strings is not a good use of resources.

These considerations led us to the *burst trie*. The primary design goal for the burst trie was to reduce the average number of string comparisons required during a search to less than two.

It was clear that achieving this goal requires an adaptive structure that stores more frequent terms such that they can be retrieved more rapidly than less frequent terms. As a secondary goal, any gains in performance should not be offset by impractical memory requirements, as observed in tries or TSTs.

A burst trie is an in-memory data structure, designed for sets of records that each have a unique string that identifies the record and acts as a key. Formally, a string s with length n consists of a series of symbols or characters c_i for $i = 0, \ldots, n$, chosen from an alphabet A of size $|A|$. We assume that $|A|$ is small, typically no greater than 256.

A burst trie consists of three distinct components, a set of records, a set of containers, and an access trie:

Records. A record contains a string; information as required by the application using the burst trie (that is, for information such as statistics or word locations); and pointers as required to maintain the container holding the record. Each string is unique.

Containers. A container is a small set of records, maintained as a simple data structure such as a list or a BST. For a container at depth k in a burst trie (depth is discussed below), all strings have length at least k, and the first k characters of all strings are identical. It is not necessary to store these first k characters. Each container also has a header, for storing the statistics used by heuristics for bursting.

Thus a particular container at depth 3 containing "author" and "automated" could also contain "autopsy" but not "auger". Choice of data structure for representing containers is considered later.

Access trie. An access trie is a trie whose leaves are containers. Each node consists of an array p, of length $|A|$, of pointers, each of which may point to either a trie node or a container, and a single *empty-string* pointer to a record. The $|A|$ array locations are indexed by the characters $c \in A$. The remaining pointer is indexed by the empty string.

The depth of the root is defined to be 1. Leaves are at varying depths.

As discussed in Section 6, there are more compact forms of trie, designed to alleviate the space wasted by sparse nodes. However, by design the nodes in an access trie should rarely be sparse, and compact nodes are more costly to traverse and maintain. As we show, for small alphabets the use of simple arrays does not lead to excessive space usage; for large alphabets, other strategies—such as dividing symbols into bytes—have to be explored, as is true for tries in general.

A burst trie can be viewed as a generalisation of other proposed variants of trie [49, 40, 46], as discussed later.

Figure 1 shows an example of a burst trie storing ten records whose keys are "came", "car", "cat", "cave", "cy", "cyan", "we", "went", "were", and "west" respectively. In this example, the alphabet A is the set of letters from A to Z, and in addition an empty string symbol $\perp$ is shown; the container structure used is a BST. In this figure, the access trie has four nodes, the deepest at depth 3. The leftmost container has four records, corresponding to the strings "came", "car", "cat", and "cave". One of the strings in the rightmost container is "$\perp$", corresponding to the string "we". The string "cy" is stored wholly within the access trie, as shown by the empty-string pointer to a record, indexed by the empty string.

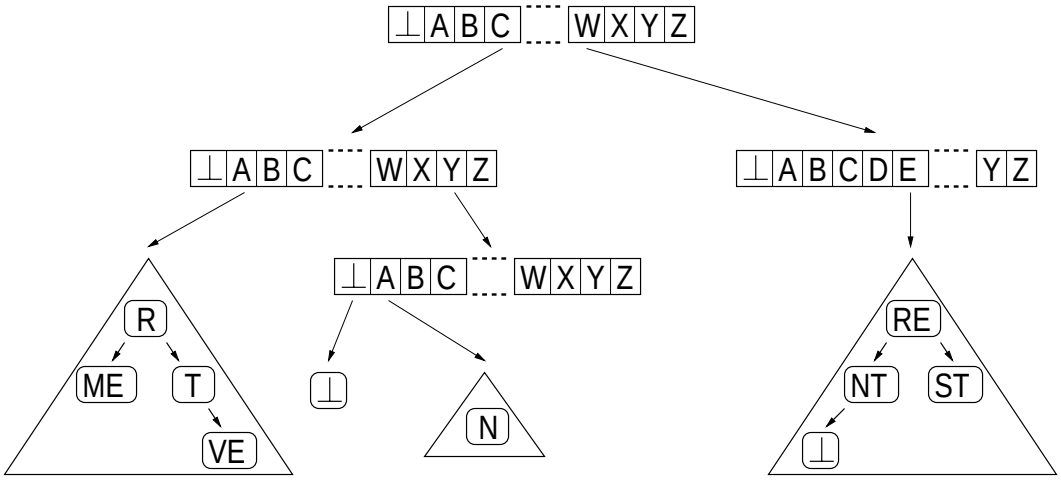

Figure 1: *Burst trie with BSTs used to represent containers.*

The use of a BST as a container enables the retrieval of records in sort order. An inorder traversal of the burst trie starts at the root. Although the records within the container store only suffixes of the original strings, they can be reconstructed by keeping track of the traversal path. However, even if an unordered structure such as a list was used to represent containers, the containers themselves would be in sort order, and their—by design—small size means that they can be sorted rapidly.

We now consider the three main operations applied to the tree: searching, insertion, and bursting.

Search

Searching a burst trie has two distinct stages, as follows. Input is a query string $c_1 \cdots c_n$ of n characters. Output is a pointer to a record, or null in the case of unsuccessful search.

Search algorithm

1. The access trie is traversed according to the leading characters in the query string. Initially the current object is the root of the access trie and the current depth i is 1.

   While the current object is a trie node t of depth $i \leq n$,

   (a) Update the current object to be the node or container pointed to by the c_ith element of t's array p, and

   (b) Increment i.

   If the current object is a trie node t of depth $i = n + 1$, the current object is the object pointed to by the empty-string pointer, which for simplicity can be regarded as a container of either zero or one records.

   Otherwise, if the current object is null the string is not in the burst trie, and search terminates.

2. The current object is a valid container. If $i \leq n$, use the remaining characters $c_i \cdots c_n$ to search the container, returning the record if found or otherwise returning null. Otherwise, the current object is a container of zero or one records that is pointed to by the empty-string pointer (since the entire string was consumed by traversal of the trie), which should be returned.

For example, to search for the record with key "came" in the burst trie in Figure 1, first follow the pointer from the 'C' slot of the rootnode to the next trie node. There, follow the pointer from the 'A' slot, corresponding to the second character. We reach the left-most container and there search for the string "me", the suffix obtained by removing the first two characters from "came". We then use the standard search algorithm for a BST.

Note that in many languages the most common words are typically short. Thus these terms are typically stored at an empty-string pointer, and are found after simply following a small number of access trie pointers with no search in a container at all. This ease of access to common terms is one of the major reasons the burst trie is so fast.

Insertion

A burst trie can grow in two distinct ways. First is insertion, when a record is added to a (possibly initially empty) container. This occurs when a new string is encountered. Second is bursting, when a container at depth k is replaced by an access trie node pointing to a set of containers, each of which is at depth $k + 1$. This occurs when the container is deemed, by heuristics as discussed later, to be inefficient. Bursting is considered below. We now consider insertion; input is a new string $c_1 \cdots c_n$ of length n; the outcome is addition of a record to a burst trie.

Insertion algorithm

1. Stage 1 of the search algorithm above is used to identify the container in which the record should be inserted. This container, which can be empty, is at depth k.

   For the special case of arriving at a trie node at depth $k = n + 1$, the container is under the empty-string pointer.

2. If $k \leq n$, the standard insertion algorithm for the container data structure is used to add the record to the container, using only the suffix characters $c_{k+1} \cdots c_n$. Otherwise, the record is added under the empty-string pointer.

Figure 2 shows the example burst trie after the insertion of the record with key "western".

Deletion is slightly more complex. To delete a record, it is first searched for, found, and deleted from its container by the usual method for the container data structure. However, deletion may empty a container, in which case the parent trie node needs to be updated. If a trie node no longer has any children, it can be removed from the burst trie, a process that can be applied recursively up to the root. Deletion does not arise in vocabulary accumulation, and we do not consider it further in this paper.

Bursting

Bursting is the process of replacing a container at depth k by a trie node and a set of new containers at depth $k+1$, which between them contain all the records in the original container.

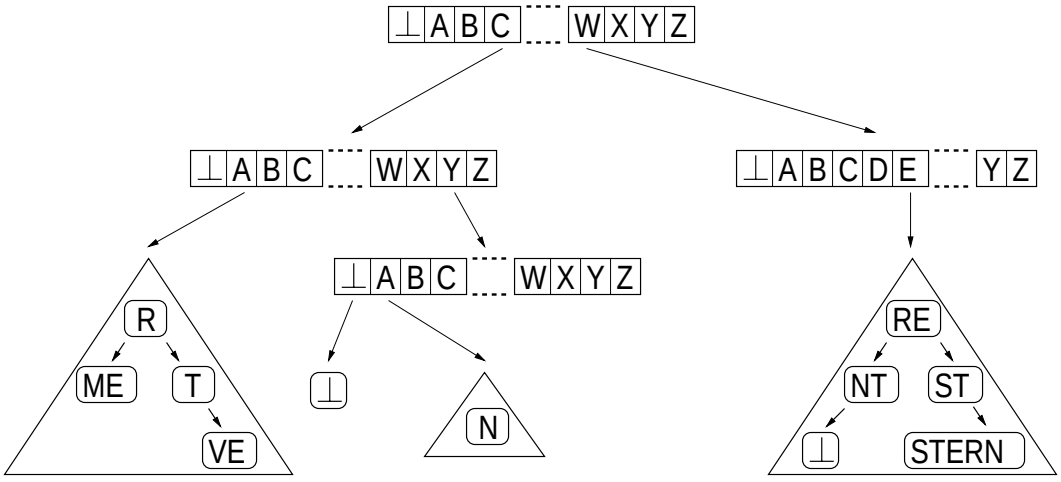

Figure 2: *Burst trie after insertion of a record with key "western".*

When to burst is considered later; here we consider how to burst. The process is as follows. Input is a burst trie; the outcome is replacement of a container with a new leaf node in the access trie and a set of child containers.

Bursting algorithm

1. A new access trie node b is created. Each pointer is set to an empty container.
2. For each record r in the original container, which has string $c^r_{k+1} \cdots c^r_n$,
   (a) The leading character c^r_{k+1} is removed from the string, and
   (b) The record is added to the container indicated by the c^r_{k+1}th pointer of b, using the string $c^r_{k+2} \cdots c^r_n$. If the record was the empty string, it is added under b's empty-string pointer.
3. The original container is discarded.

Thus bursting a container can actually save space, as the space occupied by the trie node and the new container headers can in principle be offset by the discarded characters. More significantly, for each string inserted in this trie node after the burst a character can be removed from the prefix; thus, for example, the single root node of the access trie saves one character in every stored string in the entire burst trie. It follows that judicious choice of containers to burst will reduce overall space consumption, although this was not a primary goal of our work.

In our experiments, we do not remove the leading character from each string at burst—it would be costly to do so—but do omit the first character from each string inserted in the node thereafter. It is for this reason that, as our experiments show, in some cases a burst trie is smaller than the corresponding tree.

Figure 3 shows the burst trie after bursting of the rightmost container. The records of the container are redistributed among the child nodes of a new trie node. The first character of each string is used to determine which child pointer is to follow. For example, given the term "stern" in the original container, the first character 's' determines that the suffix "tern" be

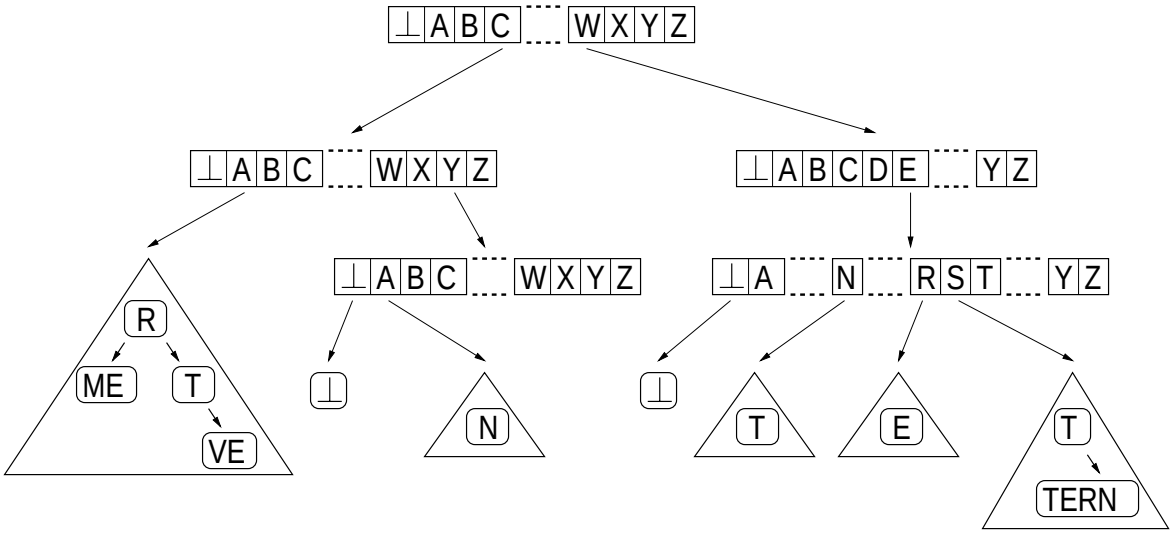

Figure 3: *Burst of a container into a new trienode.*

inserted into the container below the 'S' slot. The string "we" was represented by the empty string, in a container of several strings; it is now held under the empty-string pointer by itself.

The opposite of bursting is *consolidation*, the process of collecting a set of containers and a trie node into a single container. Consolidation could be governed by heuristics similar to those described below for bursting. We do not consider it in this paper, but use of consolidation is an interesting avenue to explore for further research.

Burst heuristics

Consider a burst trie in which the container data structures are BSTs. The properties of the burst trie depend on the rate of bursting. If all the containers have a single record each, as could arise if, for example, containers of two or more records were always burst, the burst trie is equivalent to a standard trie. As the burst rate is reduced, the burst trie behaves increasingly like a BST; if the condition for a burst is never reached, the burst trie is a single BST. The value of a burst trie is that it has the better properties of both structures, but only if the rate of bursting is appropriately managed.

A successful bursting strategy should, then, ensure that common strings are found quickly via the access trie, without excessive searching within containers. Appropriate management of containers (such as move-to-front in lists) should also be helpful in reducing search costs, by placing commonly-accessed items close to the root of the container and other items further away. A bursting strategy should also ensure that containers that are rarely accessed are not burst, since containers should be more space-efficient than trie nodes. In other words, a goal of bursting is to transform containers that have high average search costs, without bursting unnecessarily.

In the presence of a skew distribution of accesses, if a container includes a commonly-accessed record then, after bursting, the new container with this record will be more skew than the original container. Thus, assuming even a simple strategy such as containers represented by lists and move-to-front on access, bursting has the effect of increasing the likelihood that the

record is at the root of the new container. It follows that deciding whether to burst a container should depend on how often it is accessed and how costly each access is.

We propose and investigate several heuristics to trigger a burst of a container. A design goal for these heuristics was that they could be cheaply evaluated. Thus they have in common that they do not require a thorough inspection of a container to decide whether to burst. Instead, the heuristics use counters for each container. The heuristics are as follows.

Ratio. The first heuristic we propose requires two counters for each container. The counters keep track of two values: the number of times a container has been searched and the number of searches that have ended successfully at the root node of that container. We refer to a search that ends at the root as a *direct hit*. With these two values, the ratio R between the number of accesses and the number of direct hits can be calculated for each container.

This ratio can be used to determine whether to burst. We observed for a particular, fast hash table that its speed was largely due to the fact that the vast majority of searches terminated at the first node in each chain [53]. Requiring a high rate of direct hits for containers would seem, in principle, to make similar speed possible for burst tries. As a refinement, it is not worth bursting a container that is rarely accessed, so the ratio should only be tested once the total number of accesses exceeds some threshold.

For example, if less than, say, $R = 80\%$ of all accesses into a container are direct hits and there have been at least, say, 100 accesses to the container in total, it should be burst. (A container with less than 100 accesses is likely to contain only a small number of records or records with low frequency.) A drawback of *Ratio* is the additional memory required to maintain two counters per container, and the number of tests required at each access.

Limit. The second heuristic bursts a container whenever it contains more than a fixed number L of records. The rationale is to eliminate large containers, limiting total search costs. Compared to *Ratio*, only a single counter is required, and this need only be tested on insertion. This approach has the advantage that it should provide reasonable performance even if the distribution of strings is completely uniform.

There are potential disadvantages. The burst trie with *Limit* is likely to be slow if there is a container with less than K terms but all are very common, so that there are many accesses to the container but not many direct hits. However, this can only arise if these terms and no others share a common prefix, not a probable occurrence. Another disadvantage is that even containers with only rare strings are burst.

Trend. The last heuristic also uses only one counter per container. Whenever a container is created it is allocated a set amount of capital C. The current capital is modified on each access. On a direct hit, the capital is incremented by a bonus B. If a record is accessed that is already in the container, but is not a direct hit, the capital is decremented by a penalty M. When the capital is exhausted, the container is burst. Thus if the accesses are sufficiently skew, the bonuses will offset the penalties, and the container will not burst. The use of a start capital ensures that even containers with a uniform distribution will not burst until after a certain number of accesses. We explored many choices of the parameters for *Trend*. One good choice was start capital C of 5000, bonus B of 4, and penalty M of 32. We later discuss the effect of varying these parameters.

We considered many variants of this strategy. For example, on insertion of a new record, capital could be decreased, thus helping prevent creation of excessively large containers. Other variants are to change the start capital over time, to stabilise the trie for example; to reduce penalties in response to availability of memory; or to pay a penalty for every stage of a search, thus penalising long searches higher than short. We investigated a variety of such options, but did not observe behaviour superior to that in the heuristics described here.

Container data structures

Data structures to be used to represent containers must allow reasonably efficient insertion and search for records over small data sets. Candidates include linked lists, BSTs, and BST variants such as splay trees. (Hash tables are not a good candidate, since they are not efficient for small numbers of items.) We now discuss these options.

Linked lists. In a list, the only overhead in each record is a pointer to the next record, so lists are more space-efficient than the other candidate structures.

For a list in which the records are in no particular order, average access costs are high. In the context of using lists to represent containers, there are two solutions to this problem. One is to use move-to-front lists, in which the most recently accessed node is moved to the front position, a process that requires only a single test and—in the rare case that the record is moved—three assignments. With move-to-front, frequently accessed records are clustered towards the start of a list. The average search cost within a move-to-front managed list is less than twice the search cost of an optimal list where records are sorted by their access frequencies [44]. Move-to-front lists provide adaptation to local changes in vocabulary, such as the topic words of a particular document.

Another solution is to, on burst, distribute records to the new containers in decreasing order of frequency. This requires that each record stores a frequency count; such counts are kept for vocabulary accumulation, and thus are available for list management, but would otherwise be an additional cost. In our experiments this option was not effective when tested on *Web M* (described below) and we did not pursue it.

Binary search trees. A record in a BST requires two pointers to child nodes, so a BST uses more space than a list, but search paths are on average shorter. With BSTs, the position of a record in the tree depends not on its frequency, but on the order in which it was inserted.

This property can be used to reduce likely search costs: when a BST is burst, the records can first be sorted by decreasing frequency, then distributed to the new containers. Doing so means that the most frequent terms are likely to be the root nodes in their containers. As for linked lists, this requires that each record stores a frequency count.

Splay trees. For efficient splaying, a record in a splay tree requires three pointers, two to its children and one to its parent [53]. Thus they use the most space of any of the container structures we consider. However, since frequently-accessed records should be kept near the root, the structure provides adaptation to changing vocabulary and, in common with move-to-front lists, may lead to less bursting.

Table 1: *Statistics of text collections drawn from TREC.*

	Web S	Web M	Web L	Web XL	Web XXL	TREC1
Size (Mb)	100	2,048	10,240	15,360	45,229	1,206
Distinct words	114,109	1,212,885	3,604,125	4,617,076	8,634,056	503,242
Word occurrences ($\times 10^6$)	6	131	631	963	2,704	179
Documents	18,726	347,075	1,780,983	2,680,922	7,904,237	510,634
Parsing time (sec)	2.4	49.4	238.0	363.9	1,014.0	69.3

4 Experiments

Test data

The principal test data we used is six data sets drawn from the large Web track in the TREC project [24]. The five files *Web S*, *Web M*, *Web L*, *Web XL*, and *Web XXL* contain collections of web pages extracted from the Internet Archive for use in TREC retrieval experiments. The file *TREC1* is the data on the first TREC CD-ROM. The statistics of these collections are shown in Table 1. A word, in these experiments, is an alphanumeric string containing no more than two integers and not starting with an integer. We convert all words to lower case. XML and HTML markup, special characters, and punctuation are skipped.

These collections show the skew distribution that is typical of text. For example, *TREC1* has around 180 million word occurrences, of which just over half a million are distinct words. The most frequent words in the collection are "the", "of", and "and" respectively; the word "the" occurs as about one word in seventeen, almost twice as often as the second most frequent term. (The behaviour described, rather inaccurately, by Zipf's distribution [52, 54].) On the other hand, more than 200,000 words (40% of the total) occur only once.

We also used two collections with rather different properties: strings extracted from genomic data and music data. The genomic data is a collection of nucleotide strings, each typically thousands of nucleotides in length, and has an alphabet of 4 characters (wildcards were replaced by nucleotides). It is parsed into shorter strings by extracting n-grams of length 9. Such 9-grams are commonly used by genomic search utilities to locate regions where a longer inexact match may be found [51]. The music data consists of pieces in MIDI format stored in a textual representation, using an alphabet of 27 characters. We extract n-grams of length 7 from the music data, an approach used in indexing MIDI files [50]. In both cases the n-grams are overlapping, so that there is approximately one n-gram per symbol; that is, these n-grams are extracted by sliding a window over the data and taking every consecutive block of n characters.

Table 2 shows the statistics of these collections. They do not show the skew distribution that is typical of text. In the music collection, apart from one extremely common n-gram (representing the same note played eight times in sequence), the distribution was fairly uniform. The genomic data was even more uniform [33], with even the rarest n-grams occurring hundreds of times, and does not even show much locality. We expected that these data sets would lead to poor behaviour for burst tries, and indeed, are not aware of any practical data set that would be likely to be worse.

Table 2: *Statistics of non-text collections.*

	Genomic 9-grams	Music 7-grams
Size (Mb)	1,993	20
Distinct n-grams	262,146	3,147,961
N-gram occurrences ($\times 10^6$)	1,877	20
Documents	2,509,087	67,528
Parsing time (sec)	1,305.0	11.7

Methodology

The aims of our experiments are to explore the different burst heuristics and choices of container structure to identify which yield good performance, and to compare burst tries to the other principal data structures for vocabulary accumulation. The comparison is in terms of running time and memory usage. The burst trie has been tested and refined over a long period, and the other algorithms have been widely used by us for experiments; we are confident that these implementations are of high quality.[1]

In our experiments we gather the vocabulary and the statistics for the collections described above. That process involves the parsing of all data into strings. Each parsed term is inserted into a data structure that keeps the set of distinct terms, that is, the vocabulary of the collection. In addition, the data structure stores for each term its total occurrence frequency and the number of documents it occurs in.

For the purpose of comparing data structures for vocabulary accumulation, it is not necessary to include the parsing time or the time used to retrieve data from disk, since it is the same for all data structures. We therefore report CPU times, not elapsed times, and exclude the time taken to parse the collections into strings. (However, with load and parsing times excluded, elapsed times were generally no more than 1% greater than CPU times.) The internal buffers of our machines are flushed prior to each run in order to have the same starting conditions in each experiment. Little variation was observed between repeated runs. Two computers are used for this work: a Pentium III 750 MHz with 512 Mb of memory for the text collections and a Pentium III 500 MHz with 256 Mb of memory for the genomic and music collections. Both run standard versions of the Linux operating system.

The data structures tested keep the terms in alphabetical order, with the exception of hash tables and of burst tries with lists representing containers. To have a fair comparison of all data structures we use an in-memory quicksort at the end of each run with hash tables and with burst tries with list containers, and include the time of sorting in the reported times.

As reported elsewhere [53], we noticed that the standard `strcmp` string comparison function in C under Linux and Solaris is highly inefficient. By replacing the standard strcmp function with our own code we achieved speed gains of 20% or more in our experiments. We therefore used our own version of `strcmp` in all experiments.

[1] The code used for hashing and splay trees is at `http://www.cs.rmit.edu.au/~hugh/zhw-ipl.html`

Table 3: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of each text collection, for each reference data structure.*

	Web S	Web M	Web L	Web XL	Web XXL	TREC1
Trie	2.2 (23)	54.4 (246)	—	—	—	68.7 (100)
TST	3.1 (9)	80.0 (84)	386.2 (246)	—	—	94.8 (35)
BST	5.9 (4)	148.0 (39)	740.2 (118)	1,129.0 (151)	3,230.3 (285)	171.6 (16)
Splay, all acc.	6.7 (4)	176.0 (44)	881.5 (131)	1,336.8 (169)	4,055.1 (318)	215.7 (18)
Splay, int.	5.7 (4)	151.0 (44)	753.9 (131)	1,145.4 (169)	3,331.1 (318)	174.9 (18)
Hashing	1.7 (6)	44.1 (38)	218.0 (108)	323.9 (138)	921.4 (256)	48.4 (18)

5 Results

Reference data structures

In the first set of experiments we measure the time and space requirements of five reference data structures: compact tries (which we simply refer to as tries), TSTs, BSTs, splay trees, and hash tables. We report measurements with two variants of splaying: where the tree is splayed at every access, and where it is splayed only at every fourth access, that is, intermittent splaying; in other work [53], we have found that the latter is much more efficient. For hash tables, we use a hash table with 2^{20} slots and a bit-wise hash function, as discussed earlier.

Results with the text collections are shown in Table 3. Times are in seconds and the peak memory usage in Mb is given in parentheses. Runs that were impractical due to space requirements are marked with a hyphen. Several trends are obvious from these results. One is that tries (even the compact variant used in our experiments) use much more space than the other structures, and are not practical for accumulating the vocabularies of large collections. In a separate experiment we determined that a trie required around 500 Mb to accumulate the vocabulary of 5 Gb of web data. Similar problems apply to TSTs. Although less memory-demanding than a trie and faster than the other tree data structures, we were not able to use a TST for text collections larger than 10 Gb. Even so, both tries and TSTs were slower than hash tables.

Another trend is that, confirming our results in other work [53], for these data sets BSTs slightly outperform the variants of splay trees in both running time and memory usage. However, on other data BSTs could well be slower, due to the likelihood of sorted data yielding poor behaviour. Also as reported earlier [53], the intermittent variant of the splay tree is considerably faster than splaying after each access. The fastest data structure we tested is bit-wise hashing, which also requires the least memory.

Table 4 shows results for the genomic and music collections. Due to the limited size of the genomic vocabulary, where we have at most 4^9 distinct 9-grams, and due to the lack of locality, tries and TSTs are efficient. Tree variants, in particular the intermittent splay tree, are relatively much worse for genomic data than for web data. This is because genomic data is effectively random, so the past pattern of accesses is not a predictor of future accesses.

For the music data, the trie cannot be used because it requires too much memory. The TST is fast, but space-hungry, while the splay tree has worked well compared to BSTs due to the high locality of music data—individual pieces often have the same n-grams repeated many

Table 4: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of the non-text collections, for each reference data structure.*

	Genomic 9-grams	Music 7-grams
Trie	2,553.4 (7.7)	—
TST	4,200.6 (12.3)	35.9 (179.9)
BST	9,940.7 (8.5)	95.6 (96.1)
Splay, all acc.	15,385.2 (9.5)	78.0 (108.1)
Splay, int.	13,316.4 (9.5)	88.5 (108.1)
Hashing	1,940.3 (11.5)	43.3 (88.1)

Table 5: *Character comparisons and string comparisons in millions, and expected search lengths (standard deviation in parentheses), for each reference data structure, on* TREC1 *and* Web L*.*

	TREC1			Web L		
	Chars	Strings	Search len	Chars	Strings	Search len
Trie	892.4	21.5	5.83 (0.14)	—	—	—
Splay, int.	2,783.6	2,044.1	12.84 (0.59)	10,374.8	7,466.8	14.42 (0.91)
BST	3,690.9	2,863.2	14.76 (0.33)	14,144.2	9,799.2	14.46 (0.10)
Hashing	1,822.2	187.6	1.01 (0.01)	6,347.5	715.5	1.04 (0.01)

times—and the absence of an equivalent to the common words in text. In both cases hashing has again shown good performance. (Given the nature of the genomic data, a purpose-built hash function would be likely to perform even better.)

Another measure of a data structure is the number of character and string comparisons required, and expected search length. Table 5 shows these statistics for the reference data structures (other than splaying at all accesses, which is consistently worse in all respects than intermittent splaying, and TSTs). We show results for two collections, but similar relative behaviour was observed on all the collections. In this table, the number of character comparisons is the total count of characters inspected, in access trie nodes and in strings during string comparisons. The number of string comparisons is the number of invocations of the string-comparison utility, which occurs at every node in hashing and trees, and occasionally in the trie when a unique suffix is encountered. (For hash tables, the cost of inspecting the string to compute a hash value is not included.) The expected search length was computed by, for each string, storing the number of times it had been searched for; the expected search length can then be computed by, for each string, multiplying the cost of accessing the string (its depth in the tree) by its frequency of occurrence. In these adaptive structures, the expected search length changes from access to access, so we averaged the expected search length over a sequence of snapshots, one at every 10,000,000 searches.

Burst tries

Having established benchmarks with the reference data structures, we explored the performance of burst tries. We ran experiment with the burst heuristics and different container data

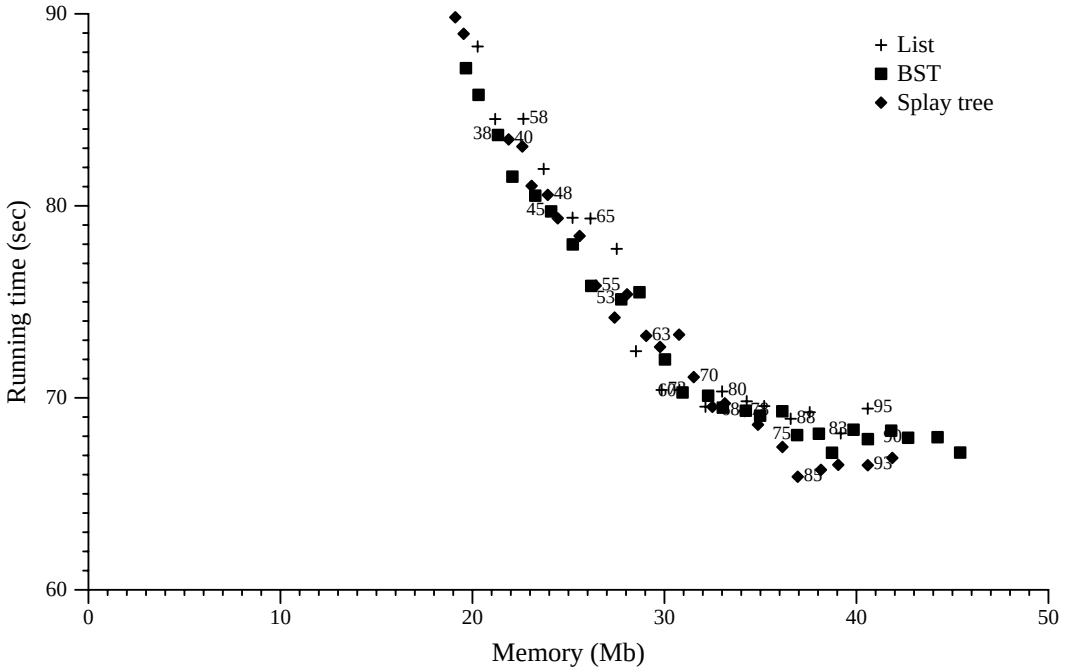

Figure 4: *Space and time for burst trie with* Ratio *burst heuristic and a variety of parameters, on* TREC1. *The parameter value is shown to the side of every third point.*

structures, on the same data sets as used above.

Ratio

Results with the *Ratio* burst heuristic on *TREC1* are shown in Figure 4. This graph shows the behaviour that might be expected: as the ratio is increased, the burst trie becomes faster—the containers get smaller—but larger, due to increased numbers of trie nodes. For example, for a ratio of 30, a burst trie with splay tree containers uses 19 Mb and 88 seconds, thus (in comparison to Table 3) using slightly more space than trees but running in half the time. At the other end of the curve, a burst trie with splay tree containers and a ratio of 85 requires 37 Mb and 66 seconds; that is, it is faster than any other tree-based or trie-based approach, while using slightly more memory than a TST.

As container structures, lists, BSTs and splay trees all yield a similar curve. However, the ratios that achieve particular points on the curve vary. For example, a BST with ratio 45 is about as efficient as a list with ratio 65. This is because a list of n nodes uses less space than a BST of n nodes, but is less efficient; lists require a higher ratio for the same efficiency. Overall these results are extremely promising; burst trees can outperform all other tree structures for vocabulary accumulation. Very similar behaviour for *Ratio* was observed on the *Web M* collection, as shown in Figure 5. (Note that different axis scales are used for the two different data sets.) We observed similar behaviour again when the minimum number of accesses is increased from 100 to 1000, except that varying ratio has less impact on space consumption.

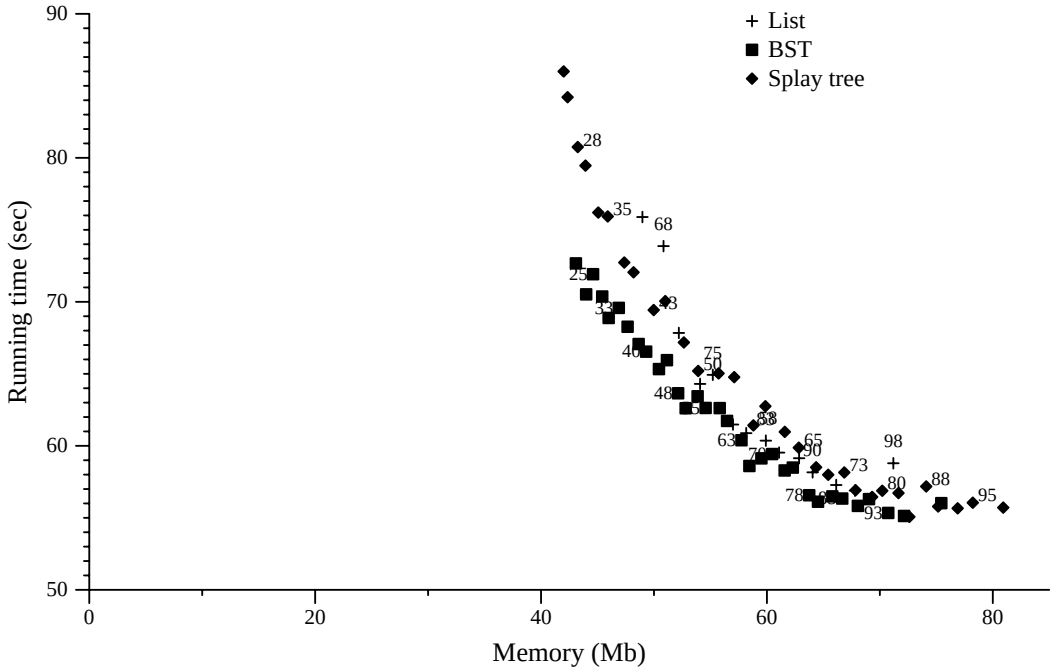

Figure 5: *Space and time for burst trie with* Ratio *burst heuristic and a variety of parameters, on* Web M. *The parameter value is shown to the immediate right of every third point.*

Limit

Results with the *Limit* burst heuristic on *TREC1* are shown in Figure 6. These results show a very different picture. For all but extremely small containers, *Limit* greatly outperforms *Ratio* for all container structures. Also, BST containers are faster for given memory than either list or splay tree containers.

The best results with *Limit* show that burst tries can be much more efficient than any other tree or trie structure. For example, the slowest run with BSTs as containers—with limit $L = 100$—uses only 15 Mb and requires 66 seconds. This is less memory than any existing tree structure tested, even standard BSTs, and is faster than tries and almost three times faster than BSTs or splay trees. Only hashing is faster, by about 27%.

Similar results for *Web M* are shown in Figure 7. In contrast to *TREC1*—which is controlled, has relatively few typing errors, and is limited in topic and source—*Web M* is uncontrolled. The relative behaviour of the different container structures is, a consequence, different to that in Figure 6, but it is still clear that BSTs are much the best container structure, and similar values of L have done well.

Results for BST containers with the *Limit* heuristic and all text data sets are shown in Table 6. These show that the gains achieved by burst tries, in comparison to the reference data structures, are consistent for all these data sets. Results for the other data sets are shown in Table 7. On the one hand, with the genomic data the burst trie did not do well: it is slower than the TST and half the speed of hashing. Nonetheless, it is still much faster than the other tree structures, and is the most compact of all data structures tested. On the other hand, with the music data the burst trie showed the best performance we observed, faster than any other data structure tested—including hash tables—with good space utilisation.

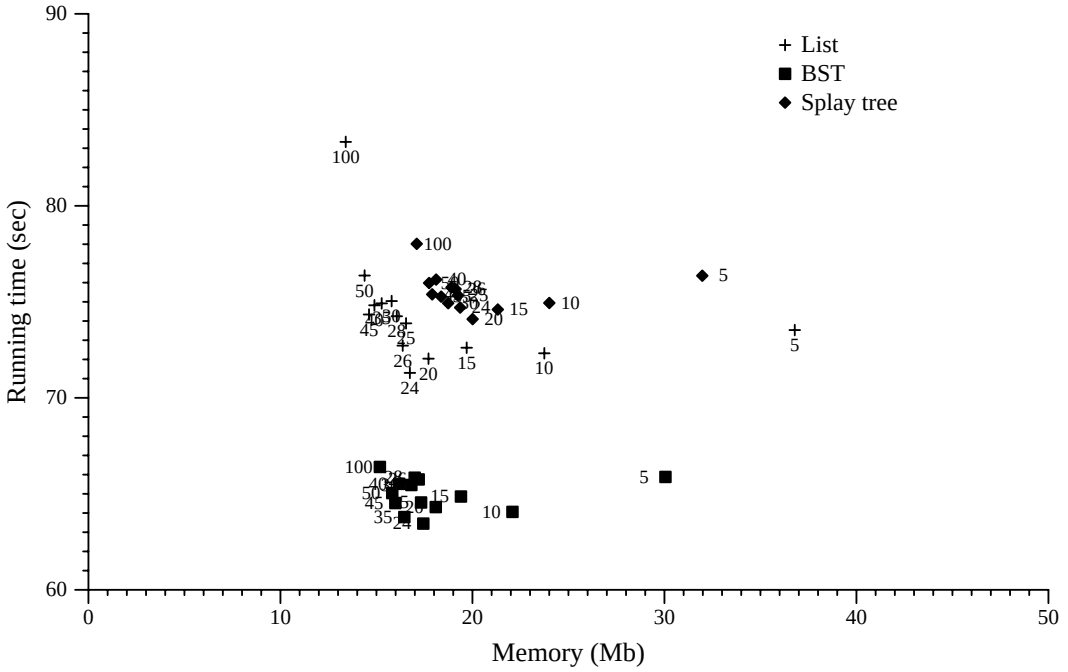

Figure 6: *Space and time for burst trie with* Limit *burst heuristic and a variety of parameters, on* TREC1. *The parameter value is shown immediately below every point.*

Table 6: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of each text collection, for burst tries with BST containers and the* Limit *heuristic.*

	Web S	Web M	Web L	Web XL	Web XXL	TREC1
$L = 35$	2.4 (4)	57.3 (40)	276.7 (120)	420.3 (152)	1,218.9 (284)	65.3 (16)
$L = 100$	2.5 (3)	59.7 (37)	288.1 (109)	431.1 (140)	1,221.3 (262)	68.2 (15)

(With other heuristics, the burst trie gave rather better performance on genomic data, at the cost of greater space utilisation. We do not report these results because these heuristics were not as effective on the other data sets.)

For a limit $L = 35$, we tested the effect of not sorting terms by frequency prior to bursting. Omission of sorting reduces the proportion of direct hits, but the total impact is an increase of only 3% in running time. Thus even if frequencies are not stored good performance is obtained.

Overall, these results are extremely promising, as illustrated in Figure 8: burst trees are faster than a trie and smaller than a BST. One interesting feature is that they are—with the exception of genomic data—consistent between data sets. Another interesting feature, which we found highly surprising, is that burst tries showed the same relative performance at all scales. *Web XXL* is 300 times larger than *Web S*, but, in comparison to the other tree structures, the speed and memory consumption is almost unchanged. Burst tries require around as much space as a BST, but are around as fast as a conventional trie.

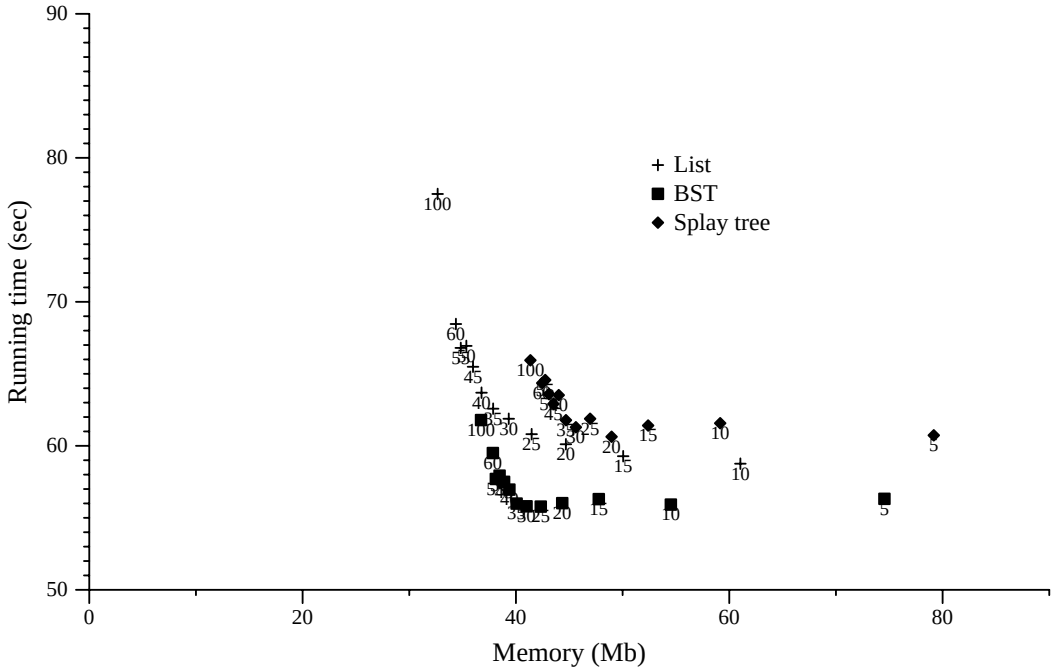

Figure 7: *Space and time for burst trie with* Limit *burst heuristic and a variety of parameters, on* Web M. *The parameter value is shown immediately below every point.*

Table 7: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of the non-text collections, for burst tries with BST containers and the* Limit *heuristic.*

	Genomic 9-grams	Music 7-grams
$L = 35$	4333.2 (7.3)	29.2 (92.7)
$L = 100$	5475.9 (7.3)	36.2 (87.6)

Trend

We explored many parameter combinations for the *Trend* burst heuristic. A typical set of results is shown in Figure 9, where M is varied and the other parameters are held constant. We had expected that this heuristic would be the best of all, as it was likely to lead to containers being burst more rapidly if they held common terms, and more slowly otherwise, but the results do not approach those of *Limit*. Despite wide investigation we did not identify combinations of parameters with significantly better performance than that shown in these graphs.

Further analysis

In the experiments reported above, we case-folded all strings and limited the total alphabet to 36 symbols. It is possible that doing so would favour tries, and burst tries in particular, because the node size is kept small. We tested the effect of a larger alphabet on *Web M*, by removing the case-folding and increasing node size to 76 symbols (of which only 62 occur in our strings). For trees and burst tries, the increase in space is almost the same: both are larger by about

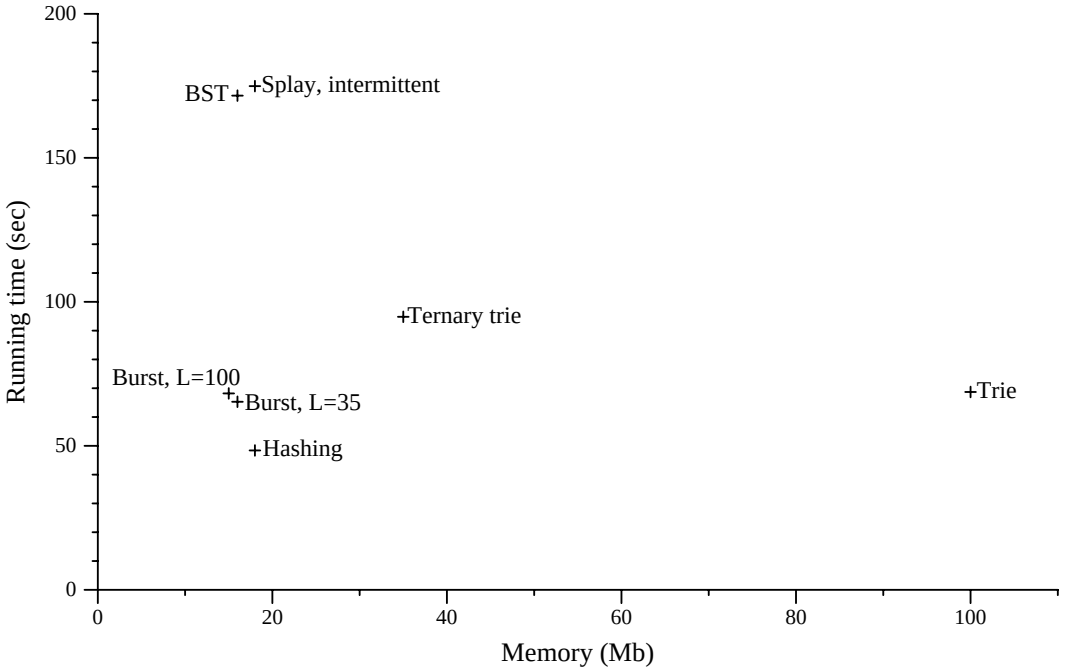

Figure 8: *Space and time for reference data structures and for the burst trie with the* Limit *burst heuristic on TREC1.*

Table 8: *Character comparisons and string comparisons in millions, and expected search lengths (standard deviation in parentheses), for burst tries with BST containers and* Limit *burst heuristic, on* TREC1 *and* Web L.

	TREC1			Web L		
	Chars	Strings	Search length	Chars	Strings	Search length
$L = 35$	996.5	184.4	3.51 (0.16)	3,270.6	441.6	3.99 (0.20)
$L = 100$	1,052.2	248.6	3.31 (0.15)	3,418.0	642.8	3.74 (0.17)

40%. Both trees and tries are slightly slower. Thus the use of case-folding has had little impact on the relative performance of the data structures.

In addition to space and speed, for the reference data structures we measured numbers of character and string comparisons, and expected search lengths. Similar results for burst tries are reported in Table 8.

Compared to the other tree data structures, the burst trie requires around one tenth to one fifth of the string comparisons and around one third of the character comparisons. Indeed, the burst trie also requires fewer comparisons than hashing. This shows that the only reason that the burst trie is slower than hashing overall is that, with hashing, a smaller number of pointers need be followed, and, we believe, hashing is able to make better use of the CPU cache.

In another set of experiments we contrasted the search performance of the burst trie to the search performance of a splay tree and hashing. In these experiments, processing of one data set was used to build a structure, which was then searched with the string occurrences in another data set. During the second, search-only phase, insertions were not allowed, so with the *Limit*

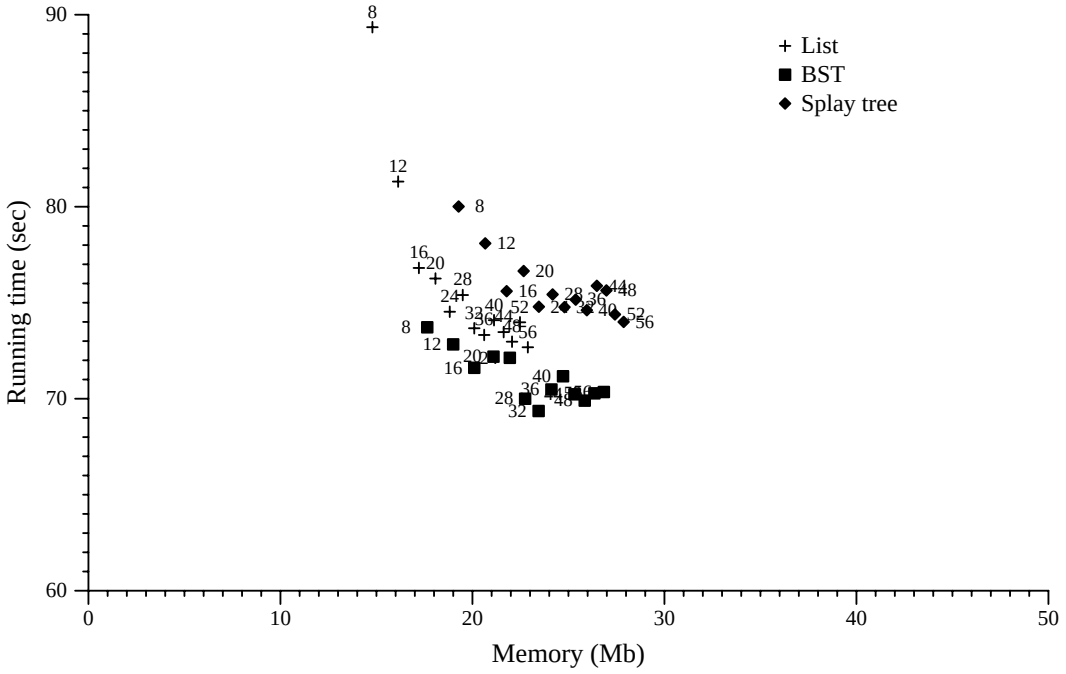

Figure 9: *Space and time for burst trie with* Trend *burst heuristic and a variety of container structures, on* TREC1. *The parameter value M is shown immediately adjacent every point; we fixed $C = 5000$ and $B = 4$.*

Table 9: *Running time in seconds for searching data structures for term occurrences in* Web XXL*, after either* TREC1 *or* Web XXL *has been used to prime the data structure.*

	TREC1	Web XXL
Splay, int.	3,015	3,070
Hashing	727	786
Burst trie, $L = 35$	917	1,008
Burst trie, $L = 100$	924	988

heuristic no bursting occurs. The results of these experiments are shown in Table 9, for two cases. In the first case, *TREC* was used to build the data structure, which was then searched with the string occurrences in *Web XXL*. The collection *Web XXL* contains many terms that are not present in the smaller *TREC1*, and thus many of the searches are not successful. In the second case, *Web XXL* was used to build the data structure, which was again searched with the string occurrences in *Web XXL*. Thus all searches for a query term are guaranteed successful.

These results show that even a static burst trie gives good performance, but show that hashing is able to achieve slightly better results than previously. The two cases yield very similar results, perhaps due to the fact that even in the first case—where the vocabularies of *TREC* and *Web XXL* are very different—the vast majority of searches are successful. Interestingly, the splay tree has shown the least degradation with increase in vocabulary size.

In other experiments we have observed that expected search lengths in splay trees, although high, do not markedly degrade with increasing collection size, while they do slowly grow for the

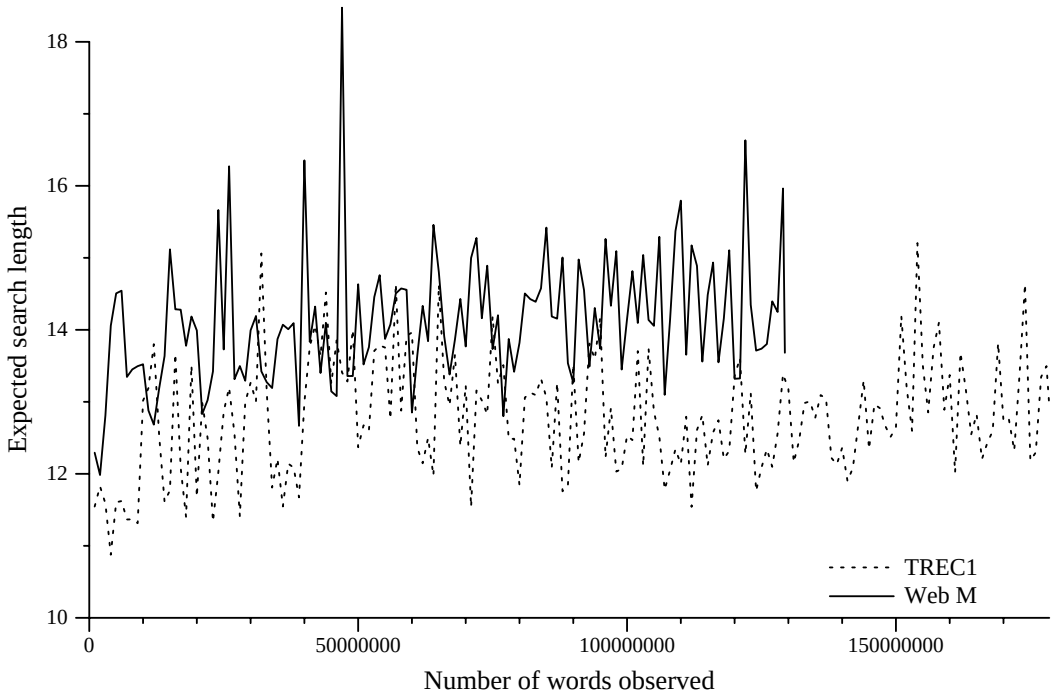

Figure 10: *Expected search length for intermittent splay tree, on TREC1 and Web M collections.*

other techniques. The trends in search lengths are illustrated in detail in Figures 10 and 11, which show expected search lengths over a series of snapshots for splay trees and burst tries, on *TREC1* and *Web M*. However, we estimate that—even should these trends continue indefinitely, a dubious supposition given the scale of extrapolation required—splay trees would still be the slowest data structure for collections of tens of terabytes or more.

6 Related work

Standard trie structures, as outlined in Section 2, are well understood. Tries were first proposed by de la Briandais [16], and the term "trie" was coined by Fredkin as an derivative from "information retrieval" [22]. Tries are widely used to represent a set of strings [1, 5, 29], that is, for dictionary management. The range of applications encompasses natural language processing [7, 36], pattern matching [20, 42], searching for reserved words for a compiler [2], for IP routing tables [34], and text compression [9]. Balanced tries (represented in a single array) are used as a data structure for dynamic hashing [18]. This broad range of applications justifies the view of tries as a general-purpose data structure, whose properties are well-understood [17, 19, 29, 25, 41, 47].

As our experiments have confirmed, tries are fast but space-intensive. The high memory usage of tries has long been recognized as a serious problem [13, 31], and many techniques have been proposed to reduce their size. Proposals for modifications to tries that address the issue of high memory usage can be broadly placed in two groups: reduction in trie node size and reduction in the number of trie nodes.

Trie node size depends on the data structure used to represent a trie node. A trie node can be implemented in many ways, in particular as an array of the size of the alphabet (an array

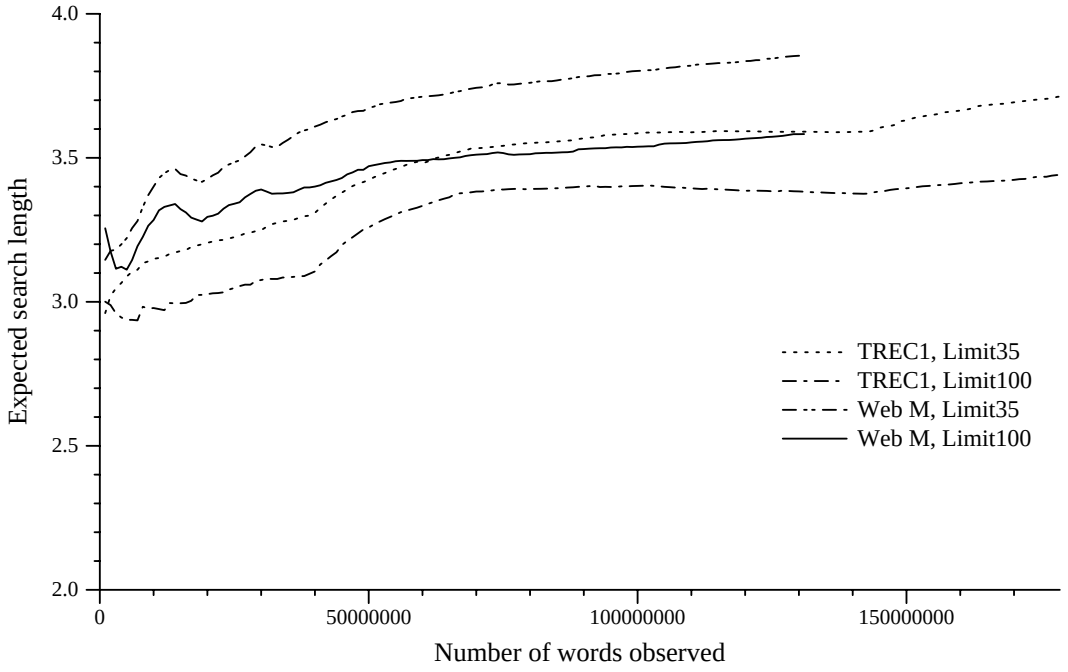

Figure 11: *Expected search length for burst trie with BST containers and* Limit *heuristic, on TREC1 and Web M collections.*

trie) [22], which is the method used in our experiments, or as a linked list that stores non-empty children as subtrees (a list trie) [16, 28]. A list implementation offers space savings if a trie node has only a few children, when an array of fixed size would consist largely of null pointers. However, the space savings come at the expense of a linked list traversal in each node instead of a simple array lookup [43].

Sussenguth [46] proposed a modified form of the linked list implementation of a trie, called a double-chained tree, in which all children of a node are placed in a list as in the linked-list implementation explained above. In addition, each list node contains a pointer to a list of its children. Based on theoretical analysis of a list trie, Sussenguth has suggested that the expected search time can be minimised when list trie nodes stop branching when there are less than six keys. Instead of a further branching the keys should be kept directly in the node in a linked list where they can be sequentially accessed [46]. However, the analysis concerns a slow variant of tries and cannot be applied to array tries.

Another option is to implement a trie node as a binary search tree. Bentley and Sedgewick [10] propose the TST, where the trie nodes are binary search trees. Clement et al. [12] analyse and empiricially test these hybrid trie structures. Array tries, list tries, and BST tries are shown to have logarithmic access costs but with different constants. The array trie implementation, which was used in our experiments, has the lowest access cost but the highest memory cost. List tries are reported to need about three times as many comparisons as a TST, which is reported to have about one-ninth the storage cost of an array trie. Our results confirm that TSTs have lower memory requirements than an array trie, but not to this extent. TSTs are described by Clement et al. as "the method of choice for managing textual data" [11], since, of the three hybrid trie structures, they offer the best time-space trade-off. It was this result

that led us to adopt a TST as a reference data structure.

Aoe et al. describe an algorithm for the compaction of a trie that is applicable to a dynamic set of keys [6]. The principle is that separate tries are constructed for the front and the rear of keys based on the observation that strings share common prefixes and suffixes. The space savings of such a two-trie structure compared to a compact trie used in our experiments is reported to be about 20%, whereas the insertion and retrieval time of a two-trie is reported to be similar to a (non-compacted) standard trie [6]. Hence, as for compact tries, a two-trie structure is not practical for a large set of keys.

The second group of proposals to reduce the size of tries address the number of trie nodes. In a standard trie, all characters of all strings are represented by pointers between trie nodes. However, in natural-language applications the trie nodes near the leaf levels tend to be sparse. (This sparsity does not arise, for example, with genomic data.) A simple way to reduce the number of trie nodes is to omit chains of nodes that have only a single descendant and lead to a leaf. This variant of a trie, in which tailing "sticks" caused by single-descendant nodes are eliminated [46], we refer to as a compact trie; we use the array implementation of a compact trie in our experiments.

Patricia tries [23, 32], which generalize the idea of collapsing sticks to a single node, are an important variant of a trie that has been widely analysed [21, 26, 38, 47]. All single-descendant nodes are omitted in a Patricia trie, not just those that lead directly to a leaf node. Single-descendant nodes are eliminated by keeping at each branch a counter to indicate which element of the key (for example which character position in a string) determines the next branching. During a search, some character positions can be skipped, since they are do not discriminate between strings stored in the next levels of the trie. To avoid false matches a final comparison has to be performed between the query string and the string found at a leaf node where a search terminates. Alternatively, the substring that is representing the single branch can be stored instead of a counter as described above, an alternative implementation known as a compact Patricia trie. Both implementations lead to a decrease in the number of trie nodes, a saving that is partly offset by a more complex structure and more complex traversals.

The value of the Patricia optimisation depends on how many nodes are single-descendant. In *TREC1*, we found that 31% of all nodes in a compact trie are single-descendant, and a similar percentage in *Web M*. However, even with a reduction in number of nodes of 30% compared to the compact trie, a Patricia trie would be much larger than a TST for a given collection. The Patricia trie is more complex than a compact trie and is thus likely to yield higher access costs on average, especially since the single-descendant nodes in a compact trie are rarely visited: although over 30% of nodes are single-descendent, we found that they are less than 4% of the nodes actually visited during processing of *TREC1* or *Web M*.

Interestingly, the Patricia optimisation is of little value for burst tries. Using the *Limit* heuristic with $L = 35$, the number of single-descendant nodes is small; for *TREC1*, only 4 of 88,016 nodes are single-descendant nodes, while for *Web M* around 2% of access trie nodes are single-descendant nodes.

In many descriptions of Patricia tries, each node has only two pointers, and the next bit rather than next byte is used to determine which pointer to follow. Pointer traversals are a major contribution to running time. With such a digital trie, typically 8 to 32 traversals are required for even a short string, an unacceptable cost. However, a digital trie is potentially more compact than a character-based trie when the set of strings is sparse, and if space was

highly restricted might be an option for an access trie.

A related structure is the LC-trie [4] and a dynamic variant, the LPC-trie [35]. These are digital tries that use level compression in addition to the path compression of Patricia tries. It is not clear how such compression can be cheaply maintained, and—as for digital tries in general—for large data sets the traversal costs are still significant. They are a candidate alternative access trie mechanism if space is restricted, but are neither fast or easy to maintain.

Another way to reduce the size of a trie is orthogonal to the ideas mentioned above. The idea is to change the order in which the letters of a string are tested. A string of length n is then viewed as a tuple of n attributes [15]. By changing the order of attributes we can influence the size of the result trie. Unfortunately, the problem of determining which order of attributes leads to the smallest trie is NP-complete [15] and is only useful for a static set of keys. However, heuristics have been proposed to minimize a static trie by locally optimizing the order of attribute testing. Comer [14] proposed the order-containing trie (or O-trie) as a space-efficient implementation of a trie. An O-trie keeps in each node information about which attribute to test. A static trie is built and then the attribute testing in subtrees is reordered to reduce the size of the trie.

Other approaches to trie compaction are, likewise, only applicable to static tries [3, 37, 40]. Al-Suwaiyel and Horowitz propose conserving memory by overlaying nodes of a trie so that empty pointers in one node occupy the same location as full pointers in another [3]. Purdin [37] proposed reduction of the size of a trie node by using a compressed bitmap in each node to indicate which children slots are used. Because of the overhead of compression, the scheme is only applicable to static tries, and is not as fast as an array trie.

Ramesh et al. [40] investigated the problem of finding a good compromise between memory usage and low access costs in a trie. The idea is to collapse subtrees of trie nodes completely and store the (suffix) strings represented by the subtree in a BST that is used as a leaf node of the trie. They propose several heuristics that transform a trie for a given set of strings into a optimized trie. One heuristic is designed to meet the objective that the resulting trie has only a minimal number of trie nodes but the worst-case binary search time does not exceed a fixed quantity. Another minimizes the worst-case binary search tree for a given maximal number of trie nodes. An additional heuristic tries to solve both objectives. Although those heuristics cannot be applied to dynamic sets of strings, the work of Ramesh et al. has similarities to ours. The most important difference is that our approach is intended for a dynamic set of strings. Our innovation is to grow an approximation to such a trie smoothly starting with a container, whereas Ramesh et al. start with a static trie and transform subtrees of the trie into BSTs. Since it is not possible to build a trie for a large text collection, because of memory requirements, the heuristics described by Ramesh et al. cannot be used.

Some of the proposals discussed above can be re-interpreted in the light of our proposals, as our work and these proposals share the idea of—in our terminology—storing suffixes of strings in containers [40, 46]. Another related proposal is the b-trie [49], where b is a block size (such as a disk block), allowing the trie to be used as a disk access structure. A contrast with our work is that we have undertaken a large-scale practical investigation of a wide range of alternative container designs and burst heuristics; another is that we have identified that the relationship between string frequency and string length is an important contributor to efficiency. These largely theoretical earlier papers do not explore how bursting should proceed; do not investigate the relationship between space and speed, but focus on space alone; have not identified the scale

of the space savings that are available in practice; and the proposed structures are less general.

However, the analytical work of Knessl and Szpankowski is of relevance [27, 48]. These papers theoretically explore the height of tries for strings under different distributions, and include the case where multiple strings are held at a single trie leaf. They establish bounds on the height of (in our terminology) burst tries. These results show that under a Markovian model burst tries are expected to be relatively flat, with no branches exceeding $C \cdot \log n$ in length, where n is the number of strings and C is determined by the distribution.

7 Conclusions

We have described the burst trie, a variant form of trie that is highly efficient for managing large sets of strings in memory. In the burst trie, each leaf holds a set of strings in a container, allowing dramatic space reductions with no impact on efficiency. We have explored a wide range of options for managing a burst trie, including different container structures, different heuristics for deciding when to burst, and different techniques for bursting. In large-scale experiments with a variety of real data sets, we have shown that burst tries are faster than compact tries and use one-sixth of the space; are more compact than binary trees or splay trees and are over two times faster; and are close to hash tables in efficiency, yet keep the data in sort order.

These results demonstrate that the burst trie is dramatically more efficient than any previous structure for the task of managing sorted strings. The improvements are the same on data sets ranging from 100 Mb to 45 Gb in size. For text data, these gains are because the more common strings are typically also the shorter strings, and are stored wholly within the access trie. Rare strings are held in containers, but because they are typically long there is no loss of efficiency: the cost of searching the container is offset by the saving of not having to traverse a large number of trie nodes. Together, these savings allow in-memory processing of larger sets of strings than was previously possible, in much less time.

The performance of the burst trie depends on the distribution of strings. The worst case we observed was genomic data, where the strings are of equal length and have a relatively flat probability distribution. Even in this case, performance was comparable to the other tree structures. Analytical results have shown that the height of a burst trie is expected to be logarithmic [27, 48]. We therefore believe that the burst trie is the data structure of choice for practical string management.

Acknowledgements

This work was supported by the Australian Research Council. We thank Adam Cannane and Sam Kimberley, and the anonymous referees for their detailed comments.

References

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

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principle Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

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

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

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

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

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

[8] J. Bell and G.K. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software—Practice and Experience*, 23(4):369–382, 1993.

[9] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[10] 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.

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

[12] J. Clement, P. Flajolet, and B. Vallée. Dynamic sources in information theory: A general analysis of trie structures. *Algorithmica*, 29(1/2):307–369, 2001.

[13] D. Comer. Heuristics for trie minimization. *ACM Transactions on Database Systems*, 4(3):383–395, September 1979.

[14] D. Comer. Analysis of a heuristic for trie minimization. *ACM Transactions on Database Systems*, 6(3):513–537, 1981.

[15] D. Comer and R. Sethi. The complexity of trie index construction. *Jour. of the ACM*, 24(3):428–440, July 1977.

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

[17] L. Devroye. A study of trie-like structures under the density model. *Annals of Applied Probability*, 2(2):402–434, 1992.

[18] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

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

[20] P. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *Jour. of the ACM*, 33(2):371–407, 1986.

[21] P. Flajolet and R. Sedgewick. Digital search trees revisited. *SIAM Jour. of Computing*, 15(3):748–767, 1986.

[22] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[23] G. Gonnet. *Handbook of Algorithms and Data structures*. Addison-Wesley, Reading, Massachusetts, 1984.

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

[25] P. Jacquet and W. Szpankowski. Analysis of digital tries with Markovian dependency. *IEEE Transactions on Information Theory*, 37(5):1470–1475, September 1991.

[26] P. Kirschenhofer and H. Prodinger. Some further results on digital search trees. In L. Kott, editor, *Automata, Languages, and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 177–185, Rennes, France, 1986. Springer-Verlag.

[27] C. Knessl and W. Szpankowski. A note on the asymptotic behavior of the height in b-tries for b large. *Electronic Jour. of Combinatorics*, 7:R39, 2000.

[28] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental algorithms*. Addison-Wesley, Reading, Massachusetts, 1968.

[29] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, Massachusetts, 1973.

[30] C. Martínez and S. Roura. Randomized binary search trees. *Jour. of the ACM*, 45(2):288–323, March 1998.

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

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

[33] C. G. Nevill-Manning and I.H. Witten. Protein is incompressible. In J. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conf.*, pages 257–266, Snowbird, Utah, 1999.

[34] S. Nilsson and G. Karlsson. Ip-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communication*, 17(6):1083–1092, June 1999.

[35] S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In K. Mehlhorn, editor, *Proc. Second Workshop on Algorithm Engineering (WAE '98)*, pages 25–36, Max-Planck-Institut für Informatik, Saarbrücken, Germany, August 1998.

[36] J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–686, 1980.

[37] T. D. M. Purdin. Compressing tries for storing dictionaries. In H. Berghel, J. Talburt, and D. Roach, editors, *Proc. IEEE Symposium on Applied Computing*, pages 336–340, Fayettville, Arkansas, April 1990. IEEE.

[38] B. Rais, P. Jacquet, and W. Szpankowski. Limiting distribution for the depth in Patricia tries. *SIAM Jour. of Discrete Mathematics*, 6:197–213, 1993.

[39] 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.

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

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

[42] R. L. Rivest. Partial match retrieval algorithms. *SIAM Jour. of Computing*, 5(1):19–50, 1976.

[43] D. G. Severance. Identifier search mechanisms: A survey and generalized model. *Computing Surveys*, 6(3):175–194, 1974.

[44] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

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

[46] E. Sussenguth. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963.

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

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

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

[50] A. L. Uitdenbogerd and J. Zobel. Melodic matching techniques for large music databases. In D. Bulterman, K. Jeffay, and H. J. Zhang, editors, *Proc. ACM Int. Multimedia Conf.*, pages 57–66, Orlando, Florida, November 1999.

[51] H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*. To appear.

[52] H. E. Williams and J. Zobel. Searchable words on the web. In submission, 2001.

[53] H. E. Williams, J. Zobel, and S. Heinz. Splay trees in practice for large text collections. *Software—Practice and Experience*. To appear.

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

[55] 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.

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