

Efficient Trie-Based Sorting of Large Sets of Strings

Ranjan Sinha

Justin Zobel

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

Abstract

Sorting is a fundamental algorithmic task. Many general-purpose sorting algorithms have been developed, but efficiency gains can be achieved by designing algorithms for specific kinds of data, such as strings. In previous work we have shown that our *burstsrt*, a trie-based algorithm for sorting strings, is for large data sets more efficient than all previous algorithms for this task. In this paper we re-evaluate some of the implementation details of *burstsrt*, in particular the method for managing buckets held at leaves. We show that better choice of data structures further improves the efficiency, at a small additional cost in memory. For sets of around 30,000,000 strings, our improved *burstsrt* is nearly twice as fast as the previous best sorting algorithm.

Keywords: Sorting, string management, algorithms, experimental algorithmics.

1 Introduction

Algorithms for sorting are a key topic of computer science. They have been investigated since before the first computers were constructed, yet developments and improvements are ongoing, with for example considerable new research in the 1990s. Sorting also continues to be a key consumer of computing resources, due to its use in applications such as database systems. Indeed, with growth in stored data volumes well outstripping improvements in processor performance, the need for better sorting algorithms is continuing.

Most of the best-known sorting algorithms are general-purpose or are best suited to fixed-length items such as integers: quicksort, mergesort, insertionsort, shellsort, and so on (Sedgewick 1998). Over the last decade, however, new sorting algorithms have been developed for the specific task of sorting strings, and these significantly outperform existing general-purpose methods (Bentley & Sedgewick 1998). Many of these methods are adaptations of radixsort. In radixsorts, the items to be sorted are treated as sequences of symbols drawn from a finite alphabet, and each symbol is represented in a fixed number of bits. Traditional radixsort proceeds by assuming the items to consist of a fixed number of symbols and ordering the items on the rightmost or least significant symbol, then the next rightmost symbol, and so on. Thus the final stage is to sort on the most significant symbol. However, such right-to-left approaches are not well-suited to variable-length strings.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

The newer radixsort algorithms are left-to-right or MSD (most significant digit first), commencing at the left with the most significant symbol, and in outline all are similar. The set of strings is partitioned on the leftmost symbol, giving a set of buckets. Then each bucket is partitioned recursively, until the buckets are so small that a simple algorithm such as insertionsort can be efficiently used. These algorithms outperform previous methods, but do not make good use of cache due to the need to repeatedly re-reference each string.

In previous work we developed a new string sorting algorithm, *burstsrt* (Sinha 2002, Sinha & Zobel 2003). The principle of *burstsrt* is that a trie is dynamically constructed as the strings are sorted, and is used to allocate each string to a bucket (Heinz, Zobel & Williams 2002). The theoretical cost is the same as that of the MSD radixsorts, in that the leading characters of each string are inspected once only, but the pattern of memory accesses makes better use of cache. In MSD radixsorts, prior to the bucket-sorting phase each string is repeatedly accessed, once for each character; in contrast, in *burstsrt* each string is accessed once only, while the trie nodes are accessed randomly. As the set of trie nodes is much smaller than the set of strings, better use is made of cache. We showed that *burstsrt* is competitive with the best methods on smaller sets of strings, and is considerably faster once the volume of strings significantly exceeds cache size.

In this paper, we re-investigate our implementation of *burstsrt*. A particular issue is the data structure used to represent the buckets. The original implementation used linked lists, which are relatively compact, avoid dynamic memory management, and are only traversed during bucket sorting; that is, lists are suitable for this application because random access is not required. However, an alternative is to use dynamic arrays, which avoid pointers at some cost in memory management, and lead to stable sorting.

Using experiments on a variety of string data sets, ranging in size from 100,000 items to over 30,000,000 items, we have found that *burstsrt* with array-based buckets is much faster than the original implementation. In all cases tested, *burstsrt* is the fastest string sorting method; for the largest data set, it is almost twice as fast as the best of the previous methods—and almost four times as fast as an efficient implementation of quicksort (Bentley & McIlroy 1993). The results unequivocally show that *burstsrt* has better asymptotic behaviour, and is the best method to use for sorting strings.

2 Background

A great variety of general-purpose sorting methods have been proposed. However, many of the best-known methods are not particularly well-suited to sorting of strings. Consider for example the behaviour

of quicksort (Sedgewick 1998). An array of strings to be sorted is recursively partitioned, the size of each partition approximately halving at each stage, if the pivot is well-chosen. As the sorting proceeds, the strings in a given partition become increasingly similar; that is, they tend to share prefixes. Determining the length of this shared prefix is costly, so they must be fully compared at each stage; thus the lead characters are repeatedly examined, a cost that is in principle unnecessary. Similar problems arise with tree-based methods such as splay sort (Moffat, Eddy & Petersson 1996), which is efficient for special cases such as sorted data, but otherwise is not competitive. It is for this reason that methods designed specifically for strings can be substantially more efficient.

As discussed by us in more detail elsewhere (Sinha 2002, Sinha & Zobel 2003), a range of new string sorting methods have been proposed in the last ten years. One is Bentley and Sedgewick’s multikey quicksort (Bentley & Sedgewick 1997, Bentley & Sedgewick 1998), in which the sorting proceeds one character at a time. When strings with the same first character are formed into a contiguous sequence, sorting of this sequence proceeds to the next character. We call this method *multikey quicksort*.

A family of methods that yields better performance is based on radix sort (Andersson & Nilsson 1993, McIlroy, Bostic & McIlroy 1993, Nilsson 1996, Rahman & Raman n.d.). All of these methods are based on the same principle: the first character (or more generally, the first symbol) of each string is used to allocate the string to a bucket. The strings in a bucket are then recursively distributed according to the next character (or symbol), continuing until the buckets are small. These final buckets can then be sorted with some simple method such as insertion sort.

Radix sorts are theoretically attractive because the leading characters in each string that are used to allocate the string to a bucket are inspected once only. As, clearly, these distinguishing characters must be inspected at least once—if they are not inspected, the value of the string cannot be determined—these algorithms approach the minimum theoretical cost. Note that the cost is still $O(n \log n)$ for a set of n distinct strings. While it is true that each character must be inspected at most once only, the length of the prefix that must be inspected grows as the log of the number of strings.

There are several distinct radix sort methods for string sorting. In 1993, McIlroy, Bostic, and McIlroy (McIlroy et al. 1993) reported several in-place versions that partition the set of strings character by character. This method was observed by Bentley and Sedgewick (Bentley & Sedgewick 1997) to be the fastest general string-sorting method, and in our earlier experiments (Sinha & Zobel 2003) we found that it was usually the fastest of the existing methods. We call this method *MBM radix sort*. Stack-based versions of radix sort were developed by Andersson and Nilsson (Andersson & Nilsson 1993, Nilsson 1996), which build and destroy tries branch by branch as the strings are processed. In our experiments, we test the most efficient of these methods, which we call *adaptive radix sort*.

These methods share the property that the sorting proceeds character by character. The first character of every string is inspected, to allocate each string to a bucket. The set of buckets can be managed with a simple structure such as an array of pointers, which is effectively a trie node. Then the first bucket is taken, and the second character of each string in the bucket is inspected to determine the next level of partitioning into buckets. Although in effect a trie is being used to allocate strings to small buckets, only

one branch of the trie need exist, corresponding to the recursively-constructed stack of subroutine calls. However, at each recursive step the accesses to the strings in a bucket are more or less random—unless the strings were originally sorted there is no locality. These random accesses are punitive in current computer architectures. In typical machines, instruction cycles are 20 to 200 times faster than memory access times; indeed, due to speed-of-light limitations, the processor can cycle once or twice before a signal can even reach memory.

3 Bursts sort

In work with string data structures by Heinz, Zobel, and Williams (Heinz et al. 2002), it was found that a *burst trie* that combined the properties of tries and trees could achieve the speed of the former in the space of the latter, while (in contrast to hash tables) maintaining the data in sort order. It was therefore attractive to investigate whether burst tries could form the basis of a fast sorting algorithm, by inserting strings into a burst trie then traversing. This led to the development of *bursts sort* (Sinha 2002, Sinha & Zobel 2003).

In outline, bursts sort is straightforward. A trie is created dynamically, initially consisting of a single trie node and a set of empty buckets. The input is an array of pointers to strings. First is an *insertion phase*. As strings are inserted, they are placed in buckets according to their leading characters. When the *capacity* of a bucket is reached, it is *burst*, that is, replaced by a trie node with a set of child buckets. Second is a *traversal phase*. Once all strings are inserted, the trie is traversed, and each bucket is sorted with a simple sort routine such as insertion sort. During this phase the pointers to the strings are copied back to the original array, in sort order.

Figure 1 shows an example of a burst trie storing ten keys: “able”, “aback”, “a”, “abet”, “acid”, “yawn”, “yard”, “yarn”, “year”, and “yoke” respectively. In this example, the alphabet is the set of letters from A to Z, and in addition an empty string symbol \perp is shown; the container used is a list. The access trie has just one trie node, at depth one. The leftmost container has five records, corresponding to the strings “able”, “aback”, “a”, “abet”, and “acid” and the rightmost container has four records corresponding to the strings “yawn”, “yard”, “yarn”, “year”, and “yoke”. The string “a” corresponds to the list node \perp in the leftmost container.

Figure 2 shows the burst trie after bursting both the left and the right containers in Figure 1. The records of the container are redistributed among the containers of the new trie node. For example, given the term “ble” in the original container, the first character ‘b’ determines that the suffix “le” be inserted into the container below the ‘B’ slot. (In our implementation the string is not actually truncated, but doing so saves considerable space, allowing much larger sets of strings to be managed (Heinz et al. 2002).)

Note how the string “a” is represented: as a single node under an empty-string pointer, because the characters of the string, including the string terminator, are represented in the trie path to this node.

In detail, bursts sort proceeds as follows.

1. Each string is inserted in turn into a burst trie, which is grown as necessary to maintain the limit L on container size.
2. When all strings have been inserted, the burst trie is traversed depth first and from left-to-right in each node, observing the following conditions.

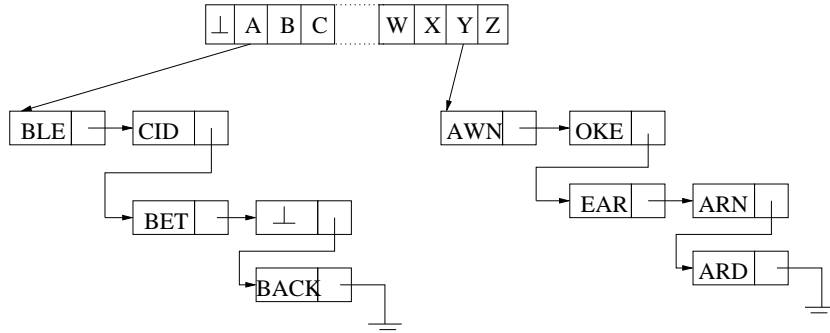


Figure 1: *Burst trie using lists to represent containers.*

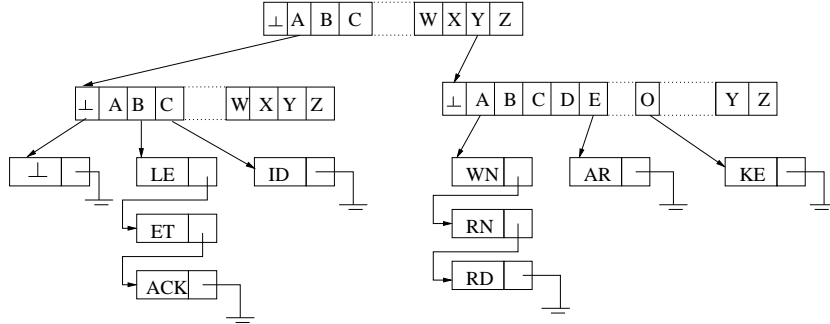


Figure 2: *Burst of containers into new trie nodes.*

- If the size of a container is one, the string can be output immediately.
- If the container is under the empty-string pointer, it can be traversed and output immediately.
- Other containers with more than one string must be sorted, starting at character position d for a container at depth d , and can then be output. We have used multi-key quicksort in our experiments; it is not tightly integrated with burstsort and there is scope for further improvement.

The cost of sorting a set of n identical strings is $O(n)$. A property of burstsort and of radixsorts in general is that sets of identical strings are handled efficiently; as the whole of each string must be inspected prior to placing it in a bucket (in an attempt to distinguish it from identical strings), it is known at insertion time that all the strings in that bucket are identical, and they do not subsequently need to be sorted.

Our previous experiments showed that burstsort made better use of cache than did other string sorting methods. The main reason appears to be the fact that each string is handled once only during the insertion phase and some strings are accessed again during bursting. Handling a string requires traversing a branch of trie nodes to find which bucket to place the string in. For reasonable choices of bucket capacity, the total number of trie nodes is small; for even the largest of our data sets, the trie size was less than a megabyte, and thus comfortably resided in cache. (The overhead space required for the trie was less than one bit per string.) In contrast, the radixsorts re-fetch the string for each trie node in the path, after a delay that leads to a reasonable likelihood that—for a large set of strings—the string is no longer in cache.

For sets of 100,000 strings, burstsort was as good as the various radixsorts, but no better. With larger

sets, however—for which even the array of pointers to the strings cannot be held in cache on our hardware—burstsot required only 60%–80% of the time used by the other methods. For example, on “Set 5” of 10,000,000 distinct strings (described below), burstsort required 11.3 seconds, while the best of the other methods required 14.6 seconds.

These experiments showed that, of the earlier sorting methods, MBM radixsort and adaptive radixsort were the fastest. The performance of these methods is explored below.

4 Implementation options

The implementation of burstsort used for our original work was strongly influenced by design choices that had proven effective for burst tries. However, these are not necessarily ideal for sorting, where for example random access to stored strings is not required. We therefore identified and evaluated a range of implementation options. These were:

- Data structure used to represent buckets.
- Size of the root trie node.
- Bucket capacity.
- Bucket sorting method.

In detail, these options are as follows.

Bucket representation

In our original work, we used linked lists to represent buckets. During the insertion phase, linked lists are highly efficient. First, the list nodes for all the strings can be allocated as a single array, and the array of pointers can be copied to the array of nodes in one pass. The strings themselves are not accessed during this process, which requires only a tiny fraction of the total sorting time. Second, during the insertion phase

a linked list need only be accessed when a bucket is burst, which is a relatively rare occurrence; the great majority of strings do not participate in a burst operation. As each inserted string can be placed at the start of a linked list representing a bucket, the existing nodes in the list are not accessed. That is, there are few random accesses, and a linked list allows extremely cheap insertion.

However, in subsequent experiments it became apparent that linked lists lead to inefficiencies in the traversal phase. In particular, sorting a bucket requires that the list be traversed and the string pointers copied to an array (a fragment of the original array of pointers can be used). With a bucket capacity of 10,000—a figure that gave the greatest overall efficiency in preliminary experiments—around 75% of total time was spent in bucket sorting. Also, the “insert at start of list” strategy means that burstsort is not stable.

Alternatives to linked lists were considered in the context of burst tries; for string management, it was found that a binary tree is the most efficient option. However, these options are not of value for sorting, as searching is not a factor. (Burst tries are used for management of distinct strings; for sorting, copies must be kept, as additional data may be associated with each string.)

Another alternative is to use arrays. In the simplest implementation of this approach, when the first string is to be placed in a bucket, an array of pointers is dynamically created. Additional strings are placed sequentially in the array. When it is full, it is burst as before. However, such an approach has serious drawbacks. If buckets are small, the size of the trie becomes unacceptable. If they are large, vast quantities of space are consumed: most buckets never approach the fixed capacity. A variation of this approach is to grow the arrays dynamically, up to the capacity, before bursting. These issues are discussed further below.

Managing buckets in this way is more costly than with linked lists: insertion into a bucket requires that both the start and the last-used position in the array must be accessed, or that counters be maintained within trie nodes; and, as the bucket grows, reallocation of memory and copying of pointers is required, leading to possible memory fragmentation. However, bucket sorting during the traversal phase is likely to be significantly more efficient, and the sort is stable.

An issue with the array representation is how to manage sets of identical strings, as there is no bound on the number of such strings and reallocating arrays is potentially an $O(n^2)$ costs. We chose to use linked lists of arrays, with a tail pointer to avoid traversal. The strings are only added at the end in each of the arrays, so stability is maintained.

Size of root node

In adaptive radixsort, the size of nodes dynamically switches between 2^8 and 2^{16} pointers, depending on the number of strings to be managed. With the larger node, pairs of letters are consumed at once, saving some operations, and the cost of inspecting null pointers can be avoided; the additional pointer at each level is required when end-of-string is observed. In our experiments we observed that this strategy was only occasionally successful, as it could lead to costly stack operations that had little benefit if the number of observed pairs of letters was small.

However, the simple heuristic of allowing the root node to be 2^{16} pointers has the potential to yield some benefit: this node can be maintained statically in the sort routine, and at run time the number of nodes allocated dynamically is somewhat reduced.

Bucket capacity

The bucket capacity is a parameter that balances the size of the trie against the cost of bucket sorting. A large trie—the consequence of small buckets—incurs memory management costs and poor cache behaviour; large buckets are expensive to sort. We test a range of bucket capacities (from 16 to 8192 strings) in our experiments.

The impact of the bucket capacity depends on the data structure used to represent buckets. Varying the capacity for a linked-list representation is straightforward. For an array representation, how the array grows also needs to be considered. There are several possibilities. One is to allocate all-at-once: all non-empty buckets are the size of the threshold. This results in dramatic memory wastage for a large threshold, though it may reduce dynamic memory management. We found that this approach is not effective.

Another possibility is to grow buckets linearly: the bucket size is increased by one, or a small constant size, for each element placed in that bucket. This scheme in principle minimises memory use, but in practice leads to fragmentation and $O(m^2)$ reallocation costs, due to copying, for a bucket of m slots.

A compromise option is to grow buckets exponentially: initially the buckets are small, then are multiplied in size until the threshold size is reached. The overhead per string is capped by the size multiplier, and in practice should be much less than this theoretical limit. Only a small number of distinct bucket sizes are created, reducing fragmentation, and dynamic memory management costs should not be excessive. Compared to the all-at-once approach, however, an extra check is required at each insertion.

In our experiments, we use the exponential approach. The memory requirements were no more than 10% greater than the memory requirements needed for the list version, and, as can be seen in the experimental results reported below, the method is extremely efficient. In our implementation, a level counter was added to each pointer in the trie structure to keep track of the bucket size. A static array was maintained which had the amount of elements to allocate at each level. Two exponential schemes were tested: starting at 16 pointers and growing by a factor of 8; and starting at 8 and growing by a factor of 4.

Bucket sorting mechanism

Once all the strings have been placed in the buckets in the trie nodes, the buckets must be sorted. In this phase, the strings in each bucket are copied back to the original array and sorted using an algorithm that is suited to small numbers of strings. We have tested a range of sorting routines, including insertion-sort, shellsort, MBM radixsort, and multikey quicksort. We found that multikey quicksort and MBM radixsort were the most efficient. Multikey quicksort is used in all our experiments; the comparison to MBM radixsort is reported below.

Other issues

Some implementation details have unpredictable impact on performance. Consider the fact that a bucket is burst when it reaches a certain size: this means that it is necessary to know bucket size. If the size is not stored, the bucket must be fully traversed at each string, an unacceptable cost; thus a counter must be held. If space is created for an array of counters in each trie node, the nodes occupy more space, but if they are stored in a header node in each bucket, as

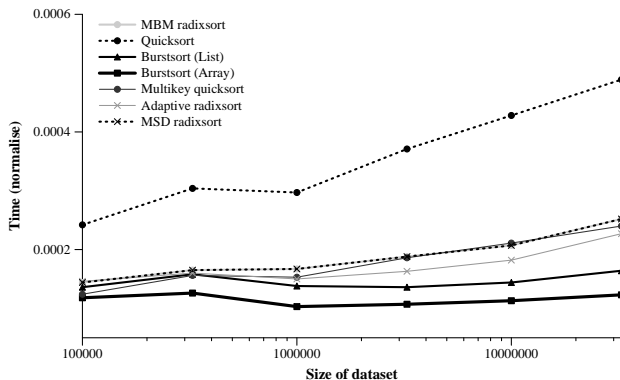


Figure 3: *Duplicates*, relative sorting time for each method. The vertical scale is time in milliseconds divided by $n \log n$.

would be required with the list representation of buckets, an extra pointer access is required. It is not clear which approach is likely to be more efficient, and in practice it is likely to depend on the data set size and the relative cost of a memory access. We do not believe that experiments on a single machine or even single architecture can identify which approach is superior.

5 Experiments

We have used two kinds of data in our experiments: words and web URLs. The words are drawn from the large web track in the TREC project (Harman 1995, Hawking, Craswell, Thistlewaite & Harman 1999), and are alphabetic strings delimited by non-alphabetic characters in web pages (after removal of tags, images, and other non-text information). The web URLs have been drawn from the same collection. For the word data, we created six subsets, of approximately 10^5 , 3.1623×10^5 , 10^6 , 3.1623×10^6 , 10^7 , and 3.1623×10^7 strings each. We call these SET 1, SET 2, SET 3, SET 4, SET 5, and SET 6 respectively. For the URL data, we created SET 1 to SET 5. In each case, only Set 1 fits in cache. In detail, the data sets are as follows.

Duplicates. Words in order of first occurrence, including duplicates. The distribution of words show similar characteristics to most natural language documents, in that some words are much more frequent than others. For example, SET 6 has just over thirty million word occurrences, of which just over seven million are distinct words.

No duplicates. Unique strings in order of first occurrence in the web data.

URL. These were extracted from the TREC documents in order of occurrence. We have stripped “http://” from the start of each URL, as it occurs in all of them. There are large numbers of duplicates, and strings are long, with an average length of 32 characters.

The methods tested are as discussed above: variations of burtsort, as implemented by us; and, for comparison other sorting methods, with high-quality implementations from the web, in most cases by the inventors of the algorithms. All of the programs are written in C. The aim of the experiments is to evaluate the impact of the implementation options on burtsort, using the other methods as benchmarks. In our earlier work we evaluated a wide range of sorting

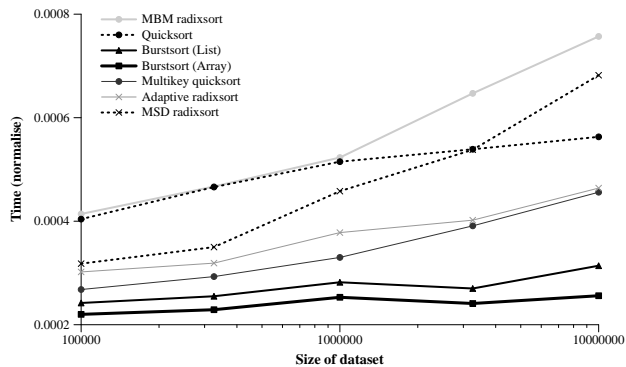


Figure 4: *URL*, relative sorting time for each method. The vertical scale is time in milliseconds divided by $n \log n$.

algorithms, but we found that they were not competitive and do not report on them here.

The reported times are to sort an array of pointers to strings. Time used for parsing and retrieval of data from disk are not included, as these are the same for all algorithms. We have used the GNU gcc compiler and the Linux operating system on a 700 MHz Pentium computer with 2 Gb of fast internal memory. The total number of milliseconds of CPU time consumed by the kernel on behalf of the program has been measured, but for sorting only; I/O times are not included. The machine was under light load, that is, no other significant I/O or CPU tasks were running. For small datasets, times are averaged over a large number of runs, to give sufficient precision.

We have also compared burtsort to *Judy*, a suite of string management tools recently released under a public license.¹ The principles of Judy are similar to those of burst tries; probably the most crucial difference is that in Judy a compact representation is used for sparse nodes. The Judy source is a high-quality production implementation of an efficient data structure, and includes a sorting tool, which for sets of distinct strings is about half the speed of burtsort. However, it is not formally comparable, because for duplicate strings only one copy is maintained, with a frequency count; while this is acceptable for the task of sorting strings, it cannot be applied if the strings are associated with other data, such as records.

6 Results

Bucket representation

The time required to sort the duplicates, from Set 1 to Set 6, is shown in Table 2. Times are shown in milliseconds. In these results, the size of the root node is 2^8 slots, buckets grow exponentially by a factor of 8, and capacity is 8192 strings. As these results show, for the larger sets, both the array and list versions of burtsort are faster than any other sorting method. The array version is much faster than the list version, and, compared to all the other sorting methods, the gain in performance grows with data set size.

The second line is a conventional quicksort, optimised for strings; as can be seen, it is around half the speed of multikey quicksort at all data sizes. The radixsorts and multikey quicksort are of similar efficiency, but the relative performance varies with data set size: of these methods, MBM radixsort is the fastest for Set 1, while adaptive radixsort is the fastest for Set 6.

¹Judy is available at www.sourcejudy.com and sourceforge.net/projects/judy/.

Table 1: *Statistics of the data collections used in the experiments.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
<i>Duplicates</i>						
Size <i>Mb</i>	1.013	3.136	7.954	27.951	93.087	304.279
Distinct Words ($\times 10^5$)	0.599	1.549	3.281	9.315	25.456	70.246
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
<i>No duplicates</i>						
Size <i>Mb</i>	1.1	3.212	10.796	35.640	117.068	381.967
Distinct Words ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
<i>URL</i>						
Size <i>Mb</i>	3.03	9.607	30.386	96.156	304.118	—
Distinct Words ($\times 10^5$)	0.361	0.92354	2.355	5.769	12.898	—
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	—

Table 2: *Duplicates. Running time (milliseconds) to sort with each method.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Splaysort	234	1,135	3,660	15,530	65,670	256,420
Quicksort	121	527	1,770	7,620	29,890	113,190
Multikey quicksort	62	272	920	3,820	14,800	55,980
MBM radixsort	58	238	820	3,630	15,370	60,460
Adaptive radixsort	73	287	900	3,230	14,460	50,950
Burstersort-List	68	275	830	2,910	10,190	36,860
Burstersort-Array	64	221	630	2,250	8,000	29,710

An alternative view of these results is shown in Figure 3, where the time to sort the data is normalised for data set size by dividing by $n \log n$. The burstersorts show much the best asymptotic behaviour, with relative time barely growing with data set size.

Similar results are shown in Tables 3 and 4, for no duplicates and URLs respectively. (Splaysort is omitted from these tables because of its poor performance on the duplicates.) For the no-duplicates data, relative performance is almost identical to the duplicates. However, the results on the URLs are startlingly different. Conventional and multikey quicksort have both outperformed MBM radixsort, which was in many cases the most efficient of the existing methods on the other data. The burstersorts have shown even better performance for the small data sets than previously, and again have good asymptotic behaviour, as illustrated in Figure 4.

Most significantly, the results clearly separate the two versions of burstersort. Using arrays for buckets is more efficient than using lists, despite the space wastage implicit in the latter.

Size of root node

We tested the effect of using either one byte or two bytes to index the root node, that is, the node could consist of either 2^8 or 2^{16} pointers. The latter was consistently the most efficient, with improvements observed across all the collections of about 5%–10%. These experiments were for burstersort with lists; experiments with array-based burstersort are continuing, but we expect to see further improvements.

Bucket capacity

In our first experiment with capacity, we tested the efficiency of four bucket sizes: 16, 128, 1024, and 8192 pointers; for the array implementation, we used factor-of-8 bucket growth. Results are shown in Tables 5 and 6. Note that Set 6 is omitted in some cases, and Set 5 in one case, because for small capacities the total space requirements (due to growth in the

trie) exceed physical memory. Note also that these results are based on smaller numbers of runs than in the other tables, hence the reduced precision. As the results show, the largest of the capacities tested gives the greatest efficiency, with the gain increasing with data set size. These results suggest that there may be further improvement available by adapting the capacity to the size of the set of strings; doing so in a principled way is a subject for further research.

In our second experiment with capacity, we varied the way in which the buckets grew in the array version. In one case, the buckets grew over 4 levels and the size of each successive level differed by powers of 8, whereas in the other case, the buckets grew over 6 levels and the size of each successive level differed by powers of 4. As can be seen in Table 7 we did not observe any significant difference between the two approaches.

Bucket sorting mechanism

Using the array implementation, we tested a range of methods for sorting bucket, including insertion-sort, quicksort, MBM radixsort, multikey quicksort, and adaptive radixsort. MBM radixsort and multikey quicksort were significantly more efficient than the others for all data sets. The timings are reported in Table 8. In contrast to MBM radixsort, multikey quicksort is not a stable sorting algorithm. As can be seen, however, multikey quicksort is by a significant margin the more efficient approach—despite the fact that it is slightly less efficient, for small data sets, in our other experiments.

7 Conclusions

We have developed a new trie-based algorithm for sorting strings, burstersort, which proceeds by inserting each string into a dynamic trie, then traversing the trie in sort order. To contain the volume of dynamic memory used, the strings are held in large buckets, which are indexed by their leading characters; in the

Table 3: *No duplicates. Running time (milliseconds) to sort with each method.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Quicksort	141	561	2,380	9,600	35,890	138,790
Multikey quicksort	67	267	1,090	4,240	15,790	60,260
MBM radixsort	61	230	940	4,080	15,680	61,340
Adaptive radixsort	77	275	1,060	3,850	14,590	59,630
Burtsort-List	71	272	1,000	3,360	11,340	43,080
Burtsort-Array	61	221	790	2,670	9,280	35,210

Table 4: *URL. Running time (milliseconds) to sort with each method.*

	Data set				
	Set 1	Set 2	Set 3	Set 4	Set 5
Quicksort	202	802	3,090	11,160	39,760
Multikey quicksort	134	504	1,970	8,100	32,540
MBM radixsort	206	808	3,140	13,450	53,650
Adaptive radixsort	151	544	2,280	8,290	33,580
Burtsort-List	121	452	1,730	5,580	21,190
Burtsort-Array	110	395	1,530	5,070	17,950

traversal phase, these buckets are sorted with an algorithm that is suitable for small sets of strings. The costs of burtsort are in theory identical to those of MSD radixsort, but in previous work (Sinha 2002) we showed that burtsort is faster than all existing algorithms once the volume of data significantly exceeds cache size, and that it has excellent asymptotic characteristics.

In this paper we have shown that further substantial improvements in performance are available, through careful investigation of a range of implementation details. Most important of these is the data structure used for the buckets. In our experiments with using arrays to represent buckets, where the arrays grow exponentially in size up to a fixed capacity, large improvements in performance were observed. We also tested a range of other implementation options. Of these, the most significant was to change the size of the root node from 2^8 to 2^{16} pointers. Other options, such as alternative bucket sorting mechanisms, had little effect.

Overall, we have shown that for large sets of strings burtsort is nearly twice as fast as any previous sorting method. This is a dramatic improvement over our original list-based implementation. While our algorithm is not in-place, the overheads are a linear increase over the size of the data to be sorted. Where such memory is available, our array-based burtsort is by far the most efficient algorithm for sorting strings.

Acknowledgements

This work was supported by the Australian Research Council and VPAC. We thank Alistair Moffat for valuable discussions and feedback at the early stages of this project.

References

- Andersson, A. & Nilsson, S. (1993), ‘Improved behaviour of tries by adaptive branching’, *Information Processing Letters* **46**(6), 295–300.
- Bentley, J. L. & McIlroy, M. D. (1993), ‘Engineering a sort function’, *Software—Practice and Experience* **23**(11), 1249–1265.
- Bentley, J. & Sedgewick, R. (1997), Fast algorithms for sorting and searching strings, in ‘Proc.

Annual ACM-SIAM Symp. on Discrete Algorithms’, ACM/SIAM, New Orleans, Louisiana, pp. 360–369.

- Bentley, J. & Sedgewick, R. (1998), ‘Sorting strings with three-way radix quicksort’, *Dr. Dobbs Journal*.
- Harman, D. (1995), ‘Overview of the second text retrieval conference (TREC-2)’, *Information Processing & Management* **31**(3), 271–289.
- Hawking, D., Craswell, N., Thistlewaite, P. & Harman, D. (1999), Results and challenges in web search evaluation, in ‘Proc. World-Wide Web Conference’.
- Heinz, S., Zobel, J. & Williams, H. E. (2002), ‘Burst tries: A fast, efficient data structure for string keys’, *ACM Transactions on Information Systems* **20**(2), 192–223.
- McIlroy, P. M., Bostic, K. & McIlroy, M. D. (1993), ‘Engineering radix sort’, *Computing Systems* **6**(1), 5–27.
- Moffat, A., Eddy, G. & Petersson, O. (1996), ‘Splaysort: Fast, versatile, practical’, *Software—Practice and Experience* **26**(7), 781–797.
- Nilsson, S. (1996), Radix Sorting & Searching, PhD thesis, Department of Computer Science, Lund, Sweden.
- Rahman, N. & Raman, R. (n.d.), ‘Adapting radix sort to the memory hierarchy’, *ACM Jour. of Experimental Algorithmics*. To appear.
- Sedgewick, R. (1998), *Algorithms in C, third edition*, Addison-Wesley Longman, Reading, Massachusetts.
- Sinha, R. (2002), ‘Fast sorting of strings with dynamic tries’. Minor thesis, School of Computer Science and Information Technology, RMIT University.
- Sinha, R. & Zobel, J. (2003), Cache-conscious sorting of large sets of strings with dynamic tries, in ‘Proc. ACM SIGACT Algorithm Engineering and Experiments (ALENEX).’, Baltimore, Maryland. To appear.

Table 5: *Bucket capacities: impact on running time (milliseconds), list implementation.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
<i>Duplicates</i>						
16	190	660	1,560	5,970	23,050	—
128	70	300	860	3,440	12,560	51,340
1024	70	260	770	2,900	10,540	40,660
8192	68	275	830	2,910	10,190	36,860
<i>No duplicates</i>						
16	180	570	1,920	6,680	24,930	—
128	70	280	1,120	4,100	14,410	53,950
1024	70	260	950	3,410	12,280	46,440
8192	71	272	1,000	3,360	11,340	43,080
<i>URL</i>						
16	610	1,970	5,890	19,410	—	—
128	170	780	3,600	12,770	43,820	—
1024	140	480	1,710	7,720	35,590	—
8192	121	452	1,730	5,580	21,190	—

Table 6: *Bucket capacities: impact on running time (milliseconds), array implementation.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
<i>Duplicates</i>						
16	248	882	2,290	13,560	167,880	—
128	78	292	820	3,690	20,970	214,800
1024	63	223	630	2,430	9,840	45,840
8192	64	221	630	2,250	8,000	29,710
<i>No duplicates</i>						
16	243	818	3,610	33,520	507,960	—
128	79	284	1,160	5,280	37,430	476,550
1024	66	225	800	2,970	12,190	64,110
8192	61	221	790	2,670	9,280	35,210
<i>URL</i>						
16	779	2,219	6,880	30,900	—	—
128	173	705	2,960	11,160	41,710	—
1024	139	456	1,580	6,430	27,580	—
8192	110	395	1,530	5,070	17,950	—

Table 7: *Bucket growth strategies: impact on running time (milliseconds). For “powers of 4”, the stages are 8, 32, 128, 512, 2048, and 8192. For “powers of 8”, the stages are 16, 128, 1024, and 8192.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
<i>No duplicates</i>						
powers of 8	61	221	790	2,670	9,280	35,210
powers of 4	61	222	790	2,680	9,300	35,320
<i>Duplicates</i>						
powers of 8	64	221	630	2,250	8,000	29,710
powers of 4	59	219	630	2,250	8,030	29,780
<i>URL</i>						
powers of 8	110	395	1,530	5,070	17,950	—
powers of 4	110	396	1,530	5,030	17,760	—

Table 8: *Bucket sorting methods: impact on running time (milliseconds) of different algorithms for sorting buckets, in array-based burstsort.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
<i>No duplicates</i>						
MBM radixsort	64	240	930	3,290	11,590	43,360
Multikey quicksort	61	221	790	2,670	9,280	35,210
<i>Duplicates</i>						
MBM radixsort	63	249	760	2,860	10,420	38,620
Multikey quicksort	64	221	630	2,250	8,000	29,710