Cache-Conscious Collision Resolution in String Hash Tables

Nikolas Askitis and Justin Zobel

School of Computer Science and Information Technology,
RMIT University, Melbourne, Australia 3000
{naskitis, jz}@cs.rmit.edu.au

Abstract. In-memory hash tables provide fast access to large numbers of strings, with less space overhead than sorted structures such as tries and binary trees. If chains are used for collision resolution, hash tables scale well, particularly if the pattern of access to the stored strings is skew. However, typical implementations of string hash tables, with lists of nodes, are not cache-efficient. In this paper we explore two alternatives to the standard representation: the simple expedient of including the string in its node, and the more drastic step of replacing each list of nodes by a contiguous array of characters. Our experiments show that, for large sets of strings, the improvement is dramatic. In all cases, the new structures give substantial savings in space at no cost in time. In the best case, the overhead space required for pointers is reduced by a factor of around 50, to less than two bits per string (with total space required, including 5.68 megabytes of strings, falling from 20.42 megabytes to 5.81 megabytes), while access times are also reduced.

1 Introduction

In-memory hash tables are a basic building block of programming, used to manage temporary data in scales ranging from a few items to gigabytes. For storage of strings, a standard representation for such a hash table is a *standard chain*, consisting of a fixed-size array of pointers (or *slots*), each the start of a linked list, where each node in the list contains a pointer to a string and a pointer to the next node.

For strings with a skew distribution, such as occurrences of words in text, it was found in earlier work [10] that a standard-chain hash table is faster and more compact than sorted data structures such as tries and binary trees. Using move-to-front in the individual chains [27], the load average can reach dozens of strings per slot without significant impact on access speed, as the likelihood of having to inspect more than the first string in each slot is low.

Thus a standard-chain hash table has clear advantages over open-addressing alternatives, whose performance rapidly degrades as the load average approaches 1 and which cannot be easily re-sized. However, the standard-chain hash table is not cache-conscious, as it does not make efficient use of CPU cache. There is little spatial locality, as the nodes in each chain are scattered across memory. While there is some temporal locality for skew distributions, due to the pattern of frequent re-access to the commonest strings, the benefits are limited by the overhead of requiring two pointers per string and by the fact that there is no opportunity for hardware prefetch. Yet the cost of cache inefficiency is serious: on typical current machines each cache miss incurs a delay of hundreds of clock cycles while data is fetched from memory.

A straightforward way of improving spatial locality is to store each string directly in the node instead of using a pointer, that is, to use *compact chaining*. This approach both saves space and eliminates a potential cache miss at each node access, at little cost. In experiments with large sets of strings drawn from real-world data, we show that, in comparison to standard chaining, compact-chain hash tables can yield both space savings and reductions in per-string access times.

We also propose a more drastic measure: to eliminate the chain altogether, and store the sequence of strings in a contiguous *array* that is dynamically re-sized as strings are inserted. With this arrangement, multiple strings are fetched simultaneously — caches are typically designed to use blocks of 32, 64, or 128 bytes — and subsequent fetches have high spatial locality. Compared to compact chaining, array hash tables can yield substantial further benefits. In the best case (a set of strings with a skew distribution) the space overhead can be reduced to less than two bits per string while access speed is consistently faster than under standard chaining. Contiguous storage has long been used for disk-based structures such as inverted lists in retrieval systems, due to the high cost of random accesses. Our results show that similar factors make it desirable to use contiguous, pointer-free storage in memory.

These results are an illustration of the importance of considering cache in algorithm design. Standard chaining was considered to be the most efficient structure for managing strings, but we have greatly reduced total space consumption while simultaneously reducing access time. These are dramatic results.

2 Background

To store a string in a hash table, a hash function is used to generate a slot number. The string is then placed in the slot; if multiple strings are hashed to the same location, some form of *collision* resolution is needed. (It is theoretically impossible to find find a hash function that can uniquely distinguish keys that are not known in advance [12].) In principle the cost of search of a hash table depends only on load average — though, as we show in our experiments, factors such as cache efficiency can be more important — and thus hashing is asymptotically expected to be more efficient than a tree.

Therefore, to implement a hash table, there are two decisions a programmer must make: choice of hash function and choice of collision-resolution method. A hash function should be from a universal class [22], so that the keys are distributed as well as possible, and should be efficient. The fastest hash function for strings that is thought to be universal is the bitwise method of Ramakrishna and Zobel [18]; we use this function in our experiments.

Since the origin of hashing — proposed by H.P. Luhn in 1953 [12] — many methods have been proposed for resolving collisions. The best known are separate or *standard* chaining and open addressing. In chaining, which was also proposed by Luhn, linear linked lists are used to resolve collisions, with one list per slot. Linked lists can grow indefinitely, so there is no limit on the *load average*, that is, the ratio of items to slots.

Open addressing, proposed by Peterson [17], is a class of methods where items are stored directly in the table and collisions are resolved by searching for another vacant slot. However, as the load average approaches 1, the performance of open addressing drastically declines. These open addressing schemes are surveyed and analyzed by Munro and Celis [16], and have recently been investigated in the context of cache [9]. Alternatives include coalesced chaining, which allows lists to coalesce to reduce memory wasted by unused slots [25], and to combine chaining and open-addressing [7]. It is not clear that the benefits of these approaches are justified by the difficulties they present.

In contrast, standard chaining is fast and easy to implement. Moreover, in principle there is no reason why a chained hash table could not be managed with methods designed for disk, such as linear hashing [13] and extensible hashing [19], which allow an on-disk hash table to grow and shrink gracefully. Zobel et al. [27] compared the performance of several data structures for in-memory accumulation of the vocabulary of a large text collection, and found that the standard-chain hash table, coupled with the bitwise hash function and a self-organizing list structure [12], move-to-front on access, is the fastest previous data structure available for maintaining fast access to variable-length strings. However, a standard-chain hash table is not particularly cache-efficient. With the cost of a memory access in a current computer being some hundreds of CPU cycles, each cache miss potentially imposes a significant performance penalty.

A cache-conscious algorithm has high locality of memory accesses, thereby exploiting system cache and making its behavior more predictable. There are two ways in which a program can be made more cache-conscious: by improving its *temporal* locality, where the program fetches the same pieces of memory multiple times; and by improving its *spatial* locality, where the memory accesses are to nearby addresses [14]. Chains, although simple to implement, are known for their inefficient use of cache. As nodes are stored in random locations in memory, and the input sequence of hash table accesses is unpredictable, neither temporal nor spatial locality are high. Similar problems apply to all linked structures and randomly-accessed structures, including binary trees, skiplists, and large arrays accessed by binary search.

Prefetch is ineffective in this context. Hardware prefetchers [6,2] work well for programs that exhibit regular access patterns. Array-based applications are prime candidates, as they exhibit stride access patterns that can be easily detected from an address history that is accumulated at run time. Hardware based prefetches however, are not effective with pointer-intensive applications, as they are hindered by the serial nature of pointer dereferences.

For such situations, techniques such as software prefetches [11,3] and different kinds of pointer cache [5,21,26] have been proposed. To our knowledge, there has been no practical examination of the impact of these techniques on the standard-chain hash table, nor is there support for these techniques on current platforms.

Moreover, such techniques concern the manifestation of cache misses, as opposed to the cause, being poor access locality. Chilimbi et al. [4] demonstrates that careful data layout can increase the access locality of some pointer-intensive programs. However, use of their techniques requires work to determine the suitability for tree-like data structures, which could prove limiting. Chilimbi et al. [4] note the applicability of their methods to chained hashing, but not with move-to-front on access, which is likely to be a limiting factor, as move-to-front is itself an effective cost-adaptive reordering

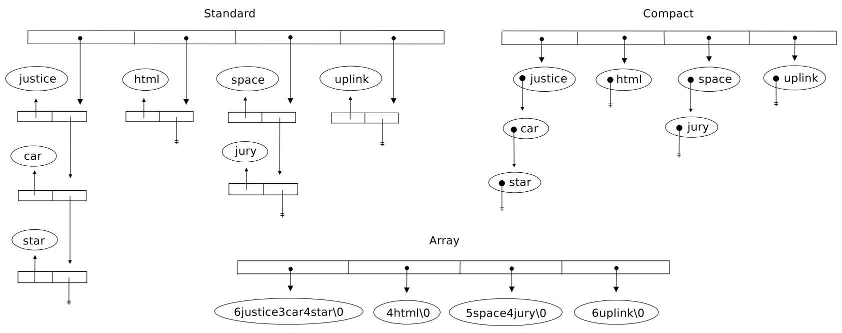

Fig. 1. The standard-chain (left), compact-chain (right) and array (below) hash tables

scheme. Nor is it clear that such methods will be of benefit in the kinds of environments we are concerned with, where the volume of data being managed may be many hundred times large than cache. However, significant gains should be available through cache-conscious algorithms, such as those we propose in this paper.

3 Cache-Conscious Hash Tables

Every node access in a standard-chain hash table incurs two pointer traversals, one to reach the node and one to reach the string. As these are likely to be randomly located in memory, each access is likely to incur a cache miss. In this section we explain our proposals for eliminating these accesses. We assume that strings are sequences of 8-bit bytes, that a character such as null is available as a terminator, and a 32-bit CPU and memory address architecture.

A straightforward step is to store each string in its node, rather than storing it in separate space. This halves the number of random accesses and saves 4 bytes per string, but requires that the nodes themselves be of variable length. Each node consists of 4 initial bytes, containing a pointer to the next node, followed by the string itself. We call this variant of chaining *compact*. The cache advantages of a compact-chain hash table are obvious, especially in the context of a skew distribution and move-to-front: each hash table lookup will involve only a single memory access, and the reduction in total size will improve the likelihood that the next node required is already cached.

We propose a novel alternative — to eliminate the chain altogether, and store the strings in a contiguous array. Prefetching schemes are highly effective with array-based structures, so this *array* hash table (shown, with the alternatives, in Figure 1) should maximize spatial access locality, providing a cache-conscious alternative to standard and compact chaining. Each array can be seen as a resizable *bucket*. The cost of access is a single pointer traversal, to fetch a bucket, which is then processed linearly.

Use of copying to eliminate string pointers for string sorting was proposed by Sinha et al. [23], who demonstrate that doing so can halve the cost of sorting a large set of strings [24]. It is plausible that similar techniques can lead to substantial gains for hash tables.

A potential disadvantage is that these arrays must be of variable size; whenever a new string is inserted in a slot, it must be resized to accommodate the additional bytes. (Note that this method does not change the size of the hash table; we are not proposing extendible arrays [20].) Another potential disadvantage is that move-to-front— which in the context of chaining requires only a few pointer assignments — involves copying of large parts of the array. On the other hand, the space overheads of an array hash table are reduced to the table itself and any memory fragmentation due to the presence of variable-length resizable objects. We explore the impact of these factors in our experiments.

A further potential disadvantage of array hash tables is that such contiguous storage appears to eliminate a key advantage of nodes — namely, that they can contain multiple additional fields. However, sequences of fixed numbers of bytes can easily be interleaved with the strings, and these sequences can be used to store the fields. The impact of these fields is likely to be much the same on all kinds of hash table. In our experiments, no fields were present.

We have explored two variants of array hash tables, *exact-fit* and *paging*. In exact-fit, when a string is inserted the bucket is resized by only as many bytes as required. This conserves memory but means that copying may be frequent. In paging, bucket sizes are multiples of 64 bytes, thus ensuring alignment with cache lines. As a special case, buckets are first created with 32 bytes, then grown to 64 bytes when they overflow, to reduce space wastage when the load average is low. (Note that empty slots have no bucket.) This approach should reduce the copying, but uses more memory. The value of 64 was chosen to match the cache architecture of our test machine, a Pentium IV.

The simplest way to traverse a bucket is to inspect it a character at a time, from beginning to end, until a match is found. Each string in a bucket must be null terminated and a null character must follow the last string in a bucket, to serve as the end-of-bucket flag. However, this approach can cause unnecessary cache misses when long strings are encountered; note that, in the great majority of cases, the string comparison in the matching process will fail on the first character.

Instead, we have used a skipping approach that allows the search process to jump ahead to the start of the next string. With skipping, each string is preceded by its length; that is, they are length-encoded [1]. The length of each string is stored in either one or two bytes, with the lead bit used to indicate whether a 7-bit or 15-bit value is present. It is not sensible to store strings of more than 32,768 characters in a hash table, as the cost of hashing will utterly dominate search costs.

4 Experimental Design

To evaluate the efficiency of the array, compact-chain, and standard-chain hash tables, we measured the elapsed time required for construction and search over a variety of string sets, as well as the memory requirements and the number of cache misses. The datasets used for our experiments are shown in Table 1. They consist of null-terminated variable length strings, acquired from real-world data repositories. The strings appear in order of first occurrence in the data; they are, therefore, unsorted. The trec datasets `trec1` and `trec2` are a subset of the complete set of word occurrences, with duplicates, in the first two TREC CDs [8]. These datasets are highly skew, containing a relatively

Table 1. Characteristics of the datasets used in the experiments

Dataset	Distinct strings	String occs	Average length	Volume (MB) of distinct	Volume (MB) total
trec1	612,219	177,999,203	5.06	5.68	1,079.46
trec2	411,077	155,989,276	5.00	3.60	937.27
urls	1,289,854	9,987,316	30.92	46.64	318.89
distinct	20,000,000	20,000,000	9.26	205.38	205.38

small set of distinct strings. Dataset `distinct` contains twenty million distinct words (that is, without duplicates) extracted from documents acquired in a web crawl and distributed as the "large web track" data in TREC. The `url` dataset, extracted from the TREC web data, is composed of non-distinct complete URLs. Some of our experiments with these sets are omitted, for space reasons, but those omitted showed much the same characteristics as those that are included.

To measure the impact of load factor, we vary the number of slots made available, using the sequence 10,000, 31,622, 100,000, 316,227 and so forth up until a minimal execution time is observed. Both the compact-chain and standard-chain hash tables are most efficient when coupled with move-to-front on access, as suggested by Zobel et al. [27]. We therefore enabled move-to-front for the chaining methods but disable it for the array, a decision that is justified in the next section.

We used a Pentium IV with 512 KB of L2 cache with 64-byte lines, 2 GB of RAM, and a Linux operating system under light load using kernel 2.6.8. We are confident — after extensive profiling — that our implementation is of high quality. We found that the hash function was a near-insignificant component of total costs.

5 Results

Skewed data. A typical use for a hash table of strings is to accumulate the vocabulary of a collection of documents. In this process, in the great majority of attempted insertions the string is already present, and there is a strong skew: some strings are much more common than others. To evaluate the performance of our hash tables under skewed access, we first construct then search using `trec1`. When the same dataset is used for both construction and search, the search process is called a self-search.

Figure 2 shows the relationship between time and memory for the hash tables during construction for `trec1`; the times in some of these results are also shown in Table 3 and the space in Table 5. Array hashing was the most efficient in both memory and speed, requiring in the fastest case under 24 seconds and around six megabytes of memory — an overhead of 0.41 MB or about 5 bits per string. This efficiency is achieved despite a load average of 20. Remarkably, increasing the number of slots (reducing the load average) has no impact on speed. Having a high number of strings per slot is efficient so long as the number of cache misses is low; indeed, having more slots can reduce speed, as the cache efficiency is reduced because each slot is accessed less often. Thus the usual assumption, that load average is a primary determinant of speed, does not always hold.

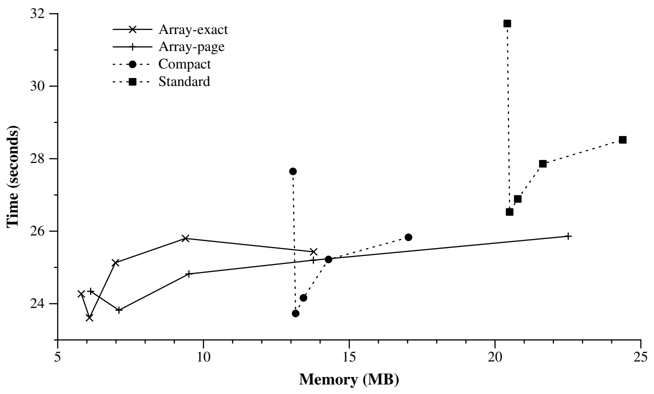

Fig. 2. Time versus memory when `trec1` is used for construction. Each point represents a different number of slots: 10,000, 31,622, 100,000, 316,228, and 1,000,000 respectively.

Table 2. The number of L2 cache misses during the construction and self-search with `trec1`

Slots	**Array**	**Compact**	**Standard**
10,000	68,506,659	146,375,946	205,795,945
100,000	89,568,430	80,612,383	105,583,144
1,000,000	102,474,094	94,079,042	114,275,513

The exact-fit and paging methods are compared in Figure 2. The exact-fit model is much more space-efficient. The relationship with speed is more complex, with paging faster in some cases, but degrading relative to exact-fit due to a more rapid increase in space consumption. We found that there is little relationship between speed and number of instructions executed.

The chaining hash tables use much more space than the array hash tables, and are no faster. The best case for chaining was with the compact chains, which in the best case required over 23.5 seconds and 13 MB of memory — an overhead of about 12 bytes per string. The standard-chain hash table was markedly inferior to both the array and compact approaches for both time and space. Given that the standard hash table was the fastest-known data structure for this task, we have strong evidence that our new structures are a significant advance.

The extent to which the speed is due to cache misses is shown in Table 2, where the `valgrind` tool (available online) is used to measure the number of L2 cache misses. (L1 misses have only a small performance penalty.) There is a reasonable correlation between the number of misses and the evaluation time. As can be seen, increasing the number of slots to 100,000 can reduce the number of misses for the chained methods, but as the table grows — and exceed cache size — performance again falls. When the load average is below 1, chaining becomes slightly more cache-efficient than the array, due to the fact that the array search function is slightly more complex.

In another experiment, the hash tables were constructed using dataset *trec1* and then searched using *trec2*. The array offered both the best time and space at 19.6 seconds with 31,662 slots. The compact chain was also able to offer a search time of about 20 seconds, but required 100,000 slots and more than double the memory required by the array. The standard chain was considerably slower.

Despite its high memory usage, the compact chain performed well under skewed access, partly due to the use of move-to-front on access. With buckets, move-to-front is computationally expensive as strings must be copied. Table 3 compares the construction and self-search costs of *trec1* with and without move-to-front on access, and includes the comparable figures for compact and standard chaining (both with move-to-front). The use of move-to-front results in slower construction and search times for the array hash; even though the vast majority of searches terminate with the first string (so there is no string movement), the cases that do require a movement are costly. Performing a move-to-front after every *k*th successful search might be more appropriate. Alternatively, the matching string can be interchanged with the preceding string, a technique proposed by McCabe [15]. However, we believe that move-to-front is unnecessary for the array, as the potential gains seem likely to be low.

URLs. Our next experiment was a repeat of the experiment above, using `urls`, a data set with some skew but in which the strings were much longer, of over thirty characters on average. As in the skewed search experiments discussed previously, our aim was to find the best balance between execution time and memory consumption. Construction results are shown in Figures 3; results for search were similar.

As for `trec1`, array hashing greatly outperformed the other methods. However, the optimum number of slots was much larger and the best load average was less than 1. Exact-fit achieved its fastest time of 4.54 seconds, using 3,162,228 slots while consuming 60 MB; paging was slightly faster, at 4.38 seconds, but used much more space. The best speed offered by the standard chain was 4.84 seconds with 3,162,228 slots, consuming over 90 MB. Compact chaining had similar speed with 74 MB. Again, array hashing is clearly the superior method.

Table 3. Elapsed time (seconds) when *trec1* is used for construction and search, showing the impact of move-to-front in the array method, and comparing to compact and standard chaining. Self-search times for paged array hash tables are omitted as they are indistinguishable from the times for exact array hash tables.

	Slots	**Array**		**Array-MTF**		**Compact**	**Standard**
		page	exact	page	exact		
Construction	10,000	24.34	24.27	23.94	25.29	27.65	31.73
	31,622	23.82	23.61	23.70	23.68	23.73	26.53
	100,000	24.82	25.13	24.96	25.08	24.16	26.89
	316,228	25.20	25.80	25.23	26.07	25.22	27.86
	1,000,000	25.88	25.43	25.71	25.45	25.83	28.52
Self-search	10,000	—	23.25	—	24.58	27.10	31.71
	31,622	—	23.17	—	24.23	22.62	25.51
	100,000	—	24.93	—	25.94	22.68	25.60
	316,228	—	25.26	—	25.98	23.75	26.56
	1,000,000	—	25.27	—	26.07	24.26	27.02

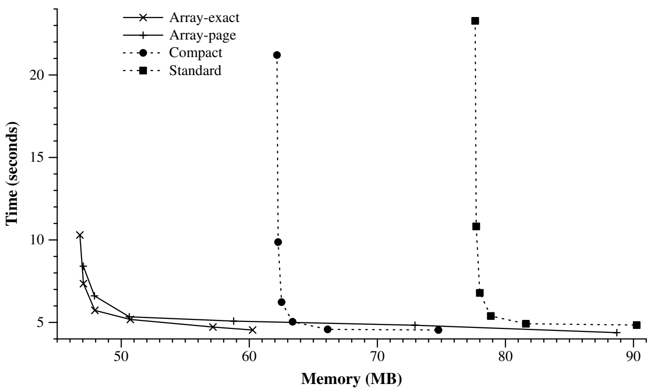

Fig. 3. Time versus memory when `urls` is used for construction. Each point represents a different number of slots: 10,000, 31,622, 100,000, 316,228, 1,000,000, and 3,162,287 respectively.

Table 4. Elapsed time (in seconds) when `distinct` is used for construction and self-search

	Slots	**Array**		**Compact**	**Standard**
		page	exact		
Construction	10,000	133.26	275.69	3524.32	3936.41
	100,000	50.70	59.45	370.30	419.45
	1,000,000	13.54	18.70	44.71	51.05
	10,000,000	9.79	10.80	11.57	13.92
	100,000,000	9.20	8.65	8.60	10.97
Self-search	10,000	—	109.11	3516.36	3915.26
	100,000	—	21.99	366.20	413.59
	1,000,000	—	11.14	42.47	47.08
	10,000,000	—	8.90	9.73	10.34
	100,000,000	—	6.96	6.67	6.94

Distinct Data. We then used the `distinct` dataset for construction and search. This dataset contains no repeated strings, and thus every insertion requires that the slot be fully traversed. Results for construction and self-search are shown in Table 4.

The difference in performance between the array and chaining methods is startling. This is an artificial case, but highlights the fact that random memory accesses are highly inefficient. With only 10,000 slots, the exact-fit array hash tables is constructed in about 275 seconds, whereas the compact and standard chains required about 3524 and 3936 seconds respectively. Paging requires only 133 seconds, a saving due to the lack of copying. This speed is despite the fact that the average load factor is 2000.

The results for self-search are similar to those for construction, with the array being up to 97% faster than the chaining methods. Once again, increasing the number of slots allows the chaining methods to be much faster, but the array is competitive at all table sizes. The chaining methods approach the efficiency of the array only when given surplus slots. For instance, with 100,000,000 slots, the compact chain is by a small margin the fastest method. However, the compact chain required 845 MB and the standard chain 1085 MB. The array used only 322 MB, a dramatic saving.

Memory Consumption. Hash tables consume memory in several ways: space allocated for the strings and for pointers; space allocated for slots; and overhead due to compiler-generated structures and space fragmentation. The memory required by the hash tables was measured by accumulating the total number of bytes requested with an estimated 8-byte overhead per memory allocation call. (With a special-purpose allocator, the 8-byte overhead for fixed-size nodes could be largely eliminated, thus saving space in standard chaining. On the other hand, in many architectures 8 rather than 4 bytes are required for a pointer.) We compared our measure with the total memory reported by the operating system under the `/proc/stat/` table and found it to be consistent.

For a standard chain, each string requires two pointers and (in a typical implementation) two `malloc` system calls. A further four bytes are required per slot. The space overhead is therefore $4n+24s$ bytes, where n is the number of slots and s is the number of strings inserted. In a compact chain, the overhead is $4n+12s$ bytes.

The memory consumed by the array hash table is slightly more complicated to model. First consider exact-fit. Apart from the head pointers leading from slots, there are no further pointers allocated by the array. The space overhead is then notionally $4n$ bytes plus 8 bytes per allocated array — that is, up to $12n$ bytes in total — but the use of copying means that there is unknown amount of space fragmentation; fortunately, inspection of the actual process size shows that this overhead is small. The array uses length-encoding, so once a string exceeds 128 characters in length an additional byte is required. For likely data the impact of these additional bytes is insignificant. Even in `urls` only 5,214 strings required this extra byte.

With paging, assume that the block size is B bytes. When the load average is high, on average each slot has one block that is half empty, and the remainder are fully used; thus the overhead is $12n+B/2$ bytes. When the load average is low — that is, $s<n$ — most slots are either empty, at a cost of $4n$ bytes, or contain a single block, at a cost of $B-l+8$, where l is the average string length. For short arrays, we allow creation of blocks of length $B/2$. Thus the wastage is around $4n+s(B-l+8)$ bytes.

The memory consumed is shown in Table 5. As can be seen by comparison with Table 1, the overhead for the exact-fit is in the best case less than two bits per string. This is a dramatic result — the space overhead is around one-hundredth of that required for even compact chaining, with, on `trec1`, only minimal impact on access times.

Exact-fit is considerably more space-efficient than paging, in particular when the table is sparse; for large tables, compact chaining is preferable. Again, there are no cases where standard chaining is competitive. These results are a conclusive demonstration that our methods are a consistent, substantial improvement.

Table 5. Comparison of the memory (in megabytes) consumed by the hash tables

Slots	`trec1`		`urls`		`distinct`		
	10,000	1,000,000	10,000	1,000,000	10,000	1,000,000	10,000,000
array-exact	5.81	13.78	46.77	57.16	205.52	218.39	322.64
array-page	6.13	22.51	47.04	72.94	205.83	249.66	463.29
compact	13.07	17.03	62.16	66.12	445.43	449.39	485.39
standard	20.42	24.38	77.64	81.60	685.43	689.39	725.39

6 Conclusions

We have proposed new representations for managing collisions in in-memory hash tables used to store strings. Such hash tables, which are a basic data structure used in programs managing small and large volumes of data, have previously been shown to be faster and more compact than sorted structures such as trees and tries. Yet in-memory hash tables for strings have attracted virtually no attention in the research literature.

Our results show that the standard representation, a linked list of fixed-size nodes consisting of a string pointer and a node pointer, is not cache-efficent or space-efficient. In every case, the simple alternative of replacing the fixed-length string pointer with the variable-length string, yielding a compact chain, proved faster and smaller.

We proposed the novel alternative of replacing the linked list altogether by storing the strings in a contiguous array. Despite what appears to be an obvious disadvantage — whenever a string is inserted, the array must be dynamically resized — the resulting cache efficiency means that the array method can be dramatically faster. In most cases, the difference in speed compared to the compact chain is small, but the space savings are large; in the best case the total space overhead was less than two bits per string, a reduction from around 100 bits for compact chaining, while speed was similar. We explored cache-aligned storage, but even in the best case the further gains were small. Our results also show that, in an architecture with cache, and in the presence of a skew data distribution, the load average can be very high with no impact on speed of access.

Our new cache-conscious representations dramatically improve the performance of this fundamental data structure, reducing both time and space and making a hash table by far the most efficient way of managing a large volume of strings.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, January 1974.
2. J. Baer and T. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.
3. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc. ASPLOS Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM Press, New York, 1991.
4. T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation*, pages 1–12. ACM Press, New York, 1999.
5. J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proc. Annual ACM/IEEE MICRO Int. Symp. on Microarchitecture*, pages 62–73. IEEE Computer Society Press, 2002.
6. J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsletter*, 23(1-2):102–110, 1992.
7. C. Halatsis and G. Philokyprou. Pseudochaining in hash tables. *Communications of the ACM*, 21(7):554–557, 1978.
8. D. Harman. Overview of the second text retrieval conference (TREC-2). In *Information Processing & Management*, pages 271–289. Pergamon Press, Inc., 1995.

9. G. L. Heileman and W. Luo. How caching affects hashing. In *Proc. ALENEX Workshop on Algorithm Engineering and Experiments*, January 2005.
10. S. Heinz, J. Zobel, and H. E. Williams. Self-adjusting trees in practice for large text collections. *Software—Practice and Experience*, 31(10):925–939, 2001.
11. M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proc. Symp. on High-Performance Computer Architecture*, pages 206–217, January 2000.
12. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Longman, second edition, 1998.
13. P. Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–587, 1982.
14. A. R. Lebeck. Cache conscious programming in undergraduate computer science. In *Proc. SIGCSE Technical Symp. on Computer Science Education*, pages 247–251. ACM Press, New York, 1999.
15. J. McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965.
16. J. I. Munro and P. Celis. Techniques for collision resolution in hash tables with open addressing. In *Proc. ACM Fall Joint Computer Conf.*, pages 601–610. IEEE Computer Society Press, 1986.
17. W. W. Peterson. Open addressing. *IBM J. Research & Development*, 1:130–146, 1957.
18. M. V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In *Proc. DASFAA Symp. on Databases Systems for Advanced Applications*, volume 6, pages 215–224. World Scientific, April 1997.
19. A. Rathi, H. Lu, and G. E. Hedrick. Performance comparison of extendible hashing and linear hashing techniques. In *Proc. ACM SIGSMALL/PC Symp. on Small Systems*, pages 178–185. ACM Press, New York, 1990.
20. A. L. Rosenberg and L. J. Stockmeyer. Hashing schemes for extendible arrays. *Jour. of the ACM*, 24(2):199–221, 1977.
21. A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proc. Int. Symp. on Computer Architecture*, pages 111–121. IEEE Computer Society Press, 1999.
22. D. V. Sarwate. A note on universal classes of hash functions. *Information Processing Letters*, 10(1):41–45, 1980.
23. R. Sinha, D. Ring, and J. Zobel. Cache-efficient string sorting using copying. *In submission*.
24. R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Jour. of Exp. Algorithmics*, 9, 2005.
25. J. S. Vitter. Analysis of the search performance of coalesced hashing. *Jour. of the ACM*, 30(2):231–258, 1983.
26. C. Yang, A. R. Lebeck, H. Tseng, and C. Lee. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Architecture Code Optimisation*, 1(4):445–475, 2004.
27. J. Zobel, H. E. Williams, and S. Heinz. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, December 2001.