Efficient Data Structures for Cache Architectures

Nikolas Askitis

Doctor of Philosophy
School of Computer Science and Information Technology,
RMIT University,
Melbourne, Australia.

www.naskitis.com
askitisn@gmail.com

August 1st, 2007

Acknowledgments

I owe a lot to my loving parents Zoi and Peter Askitis. My parents offered me unconditional support with the comfort, encouragement, and foundations I needed to move forward and excel in academia. I have been truly blessed with wonderful and caring parents. Thanks Mom and Dad!

I am also very grateful to my supervisor and mentor, Professor Justin Zobel. Justin encouraged me to pursue a research career and supported me immensely during my PhD candidature. His unwavering encouragement, enthusiasm, patience, and guidance helped me get through some of the toughest times of my life thus far, and helped me to discover a passion for research. I truly feel honored and privileged to have had the opportunity to know and work with such a marvelous and inspiring man. Thank you Justin.

I would also like to thank the anonymous reviewers of my journals and conference papers. Their feedback was invaluable at improving the quality of my thesis.

I would also like to thank my research colleagues at RMIT University. They provided an enjoyable and stimulating atmosphere for research. I feel honored to have had the opportunity to meet and work with such wonderful people. In particular, I would like to thank Dr. Ranjan Sinha for our interactions and his enthusiasm for developing the HAT-trie.

I especially want to thank my dear friend Dr. S. Reddy and his wife Krishna. Their love, teachings, and guidance helped me get through the crucial periods of my candidature. I also greatly appreciate their support during my travels, in particular my 2006 Industry Internship in India — an enlightening experience that taught me a way of life that I would have never experienced otherwise.

Sincerely Yours,
Dr. Nikolas Askits.

Contents

Abstract

A key decision when developing computing applications is the choice of a mechanism to store and retrieve strings. There are many data structures available for this task, including the different types of binary search tree, trie, and hash table. The most efficient current in-memory data structures are the hash table with move-to-front chains and the burst trie, both of which use linked lists as a substructure, and variants of binary search tree.

These pointer-intensive data structures are computationally efficient, but typical implementations use large numbers of nodes and pointers to manage strings, an approach that is not efficient in use of cache. Nodes are typically scattered across main memory, which can cause a cache-miss on access. As a consequence, the current choices available for string management, including the chaining hash table, burst trie, and variants of binary search tree, attract a significant performance bottleneck on current cache-oriented processors. Data structures therefore need to be designed to exploit cache, in order to minimize costly accesses to main memory. Current research has addressed this issue by attempting to cluster or relocate nodes based primarily on profiled information, where frequently accessed nodes are stored contiguously in blocks.

However, such techniques are of limited value for dynamic data structures for strings. Furthermore, they do not eliminate pointers, which is the key to improving the cache-efficiency of programs. In this thesis, we explore cache-conscious alternatives to many existing string data structures, which simultaneously reduce space consumption and processing time.

Our experiments show that, for large sets of strings and in comparison to well-known data structures, the improvement can be substantial. For hashing, in the best case the total space overhead is reduced from 200 bits per string to less than a single bit per string at little to no cost in speed. For the burst trie, over 300 megabytes of strings can be stored in a total of under 200 megabytes of memory with significantly improved search time. These results, on a variety of data sets and string distributions, show that cache-friendly variants of fundamental data structures can yield remarkable gains in performance.

We then extend our study to explore the application of the burst trie on disk, by developing a disk-resident B-trie. The B-trie is a practical and efficient approach to maintaining a trie-based disk-resident string data structure, while sustaining competitive performance against variants of

B-trees. Our B-trie can offer savings in space, while simultaneously exceeding the performance of a B-tree — by almost 50% under skew access. Overall, these results show that simple cache-oriented redesigns of fundamental data structures can lead to dramatic performance improvements.

Chapter 1

Introduction

Simple in-memory data structures are basic building blocks of programming, and are used to manage temporary data in scales ranging from a few items to gigabytes. For the storage and retrieval of strings, the main data structures are the varieties of hash table, trie, and binary search tree.

An efficient representation for the 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. The most efficient form of trie is the burst trie; a *standard* burst trie consists of trie nodes — each node being an array of pointers, one for each letter of the alphabet — that descend into containers represented as linked lists, where each node has a string pointer and a pointer to the next node. The use of a trie structure allows for rapid sort access to containers; the strings within containers however, remain unsorted. In the *standard* binary search tree (BST), each node has a string pointer and two child pointers. These string data structures are illustrated in Figure 1.1 and are currently the fastest and most compact tools available for managing large sets of strings in-memory [Heinz et al., 2002; Williams et al., 2001; Zobel et al., 2001; Bell and Gupta, 1993; Knuth, 1998; Crescenzi et al., 2003].

We use these data structures for common computing applications such as text compression, pattern matching, dictionary management, vocabulary accumulation, and document processing. Typical software systems such as databases and search engines are dependent on data structures to manage their data efficiently. Most existing data structures in computer science, such as the standard hash table, the trie, the BST and its variants including the splay tree, red-black tree, AVL tree, and ternary search trie (TST), assume a non-hierarchical random access memory (RAM) model, which states:

- All memory addresses supported by the underlying architecture are accessible.
- All memory accesses are of equal cost.

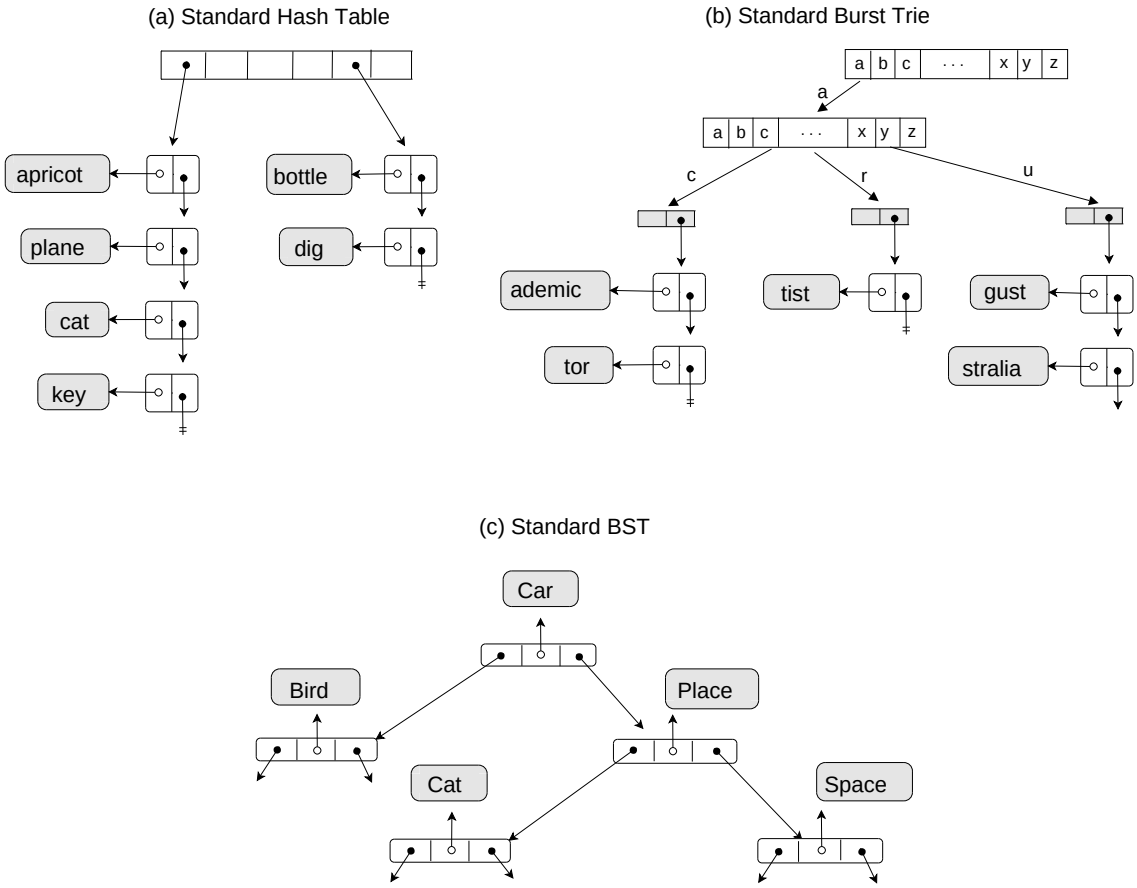


Figure 1.1: The hash table (a), burst trie (b), and binary search tree (c). These standard chained data structures are currently the fastest and most compact data structures for storage and retrieval of variable-length strings in memory. The hash table is the fastest and most compact, but it cannot maintain strings in sort order. The burst trie, however, can maintain efficient sorted access to containers while approaching the speed of hashing. Sorted access is necessary for tasks such as range search. The BST is a fully-sorted data structure that offers good performance on average.

These assumptions allow data to be placed anywhere in memory, with the expectation of uniform cost. As a result, existing data structures can be fast to access, as they can minimize the number of instructions executed. However, the speed of processors have increased more rapidly than has the response time of main memory devices, leading to a performance gap that has reached two orders of magnitude on current machines. The implications are serious. A random access to memory can force the processor to wait for many hundreds of clock cycles. To alleviate these high access costs, current processors have a system of caches that sit between the processor and memory. A cache is a small high-speed memory device that operates on the assumption that recently accessed data is likely to be referenced again in the near future. A cache can thereby create the illusion of fast memory, by copying recently accessed data from main memory in anticipation of its re-use in the near future. Main memory is consulted only when the required data is not in cache, known as a cache-miss.

However, the use of cache violates the principles of the RAM model. First, not all addresses in memory are accessible. A cache is a transparent layer of memory that cannot be accessed by a program and is administrated solely by the underlying hardware. Second, memory access does not have a uniform cost — a cache is at least a magnitude faster than main memory. Nonetheless, typical implementations of existing string data structures such as the standard hash table, burst trie, and the BST continue to assume uniform access costs and are therefore oblivious to cache. As a consequence, these fundamental data structures are likely to attract excessive cache misses that will dominate their performance on current cache-oriented processors.

Although a cache is not accessible, a program can make better use of it — that is, become cache-conscious — by improving its *spatial* and *temporal* access locality through regular and predictable memory access patterns. Spatial locality is improved by accessing data items that are near to each other; temporal locality is improved by frequently re-using a small set of data items. To be efficient on current cache-oriented processors, data structures therefore need to be designed to combine computationally efficient algorithms with cache-conscious optimizations [Kowarschik and Weiß, 2003; Meyer et al., 2003]. However, the best-known techniques for improving the cache efficiency of programs have limited support or effectiveness on large and dynamic string data structures, such as the standard hash table and burst trie.

In this thesis, we address the problem of cache-inefficiency in fundamental string data structures, by re-considering the principles of their design. We show that simple cache-oriented redesigns of the string data structures can lead to substantial improvements in performance without compromising their dynamic characteristics. In particular, we re-design the current best alternatives — the hash table, burst trie, and BST — by exploring novel cache-conscious pointer elimination techniques that challenge the principles of the RAM model by minimizing random access to memory, at the expense of increasing the number of instructions executed. However, as the speed of processors continues to race ahead of main memory, an increase in the computational cost of a program will often be compensated if it reduces the number of cache misses incurred [Meyer

et al., 2003].

The use of pointers in dynamic string data structures are the fundamental cause of cache-inefficiency, as they can lead to random memory accesses. Rao and Ross [2000] noted the importance of pointer elimination in improving the performance of integer-based tree data structures:

> "... pointer elimination is an important technique in cache optimization since it increases the utilization of a cache line ... however removing pointers completely often introduces some restrictions ... partial pointer elimination is a general technique and can be applied to other in-memory structures to improve their cache behavior ..."

Nonetheless, pointer elimination has yet to be presented in literature as a viable and effective means at improving the cache-efficiency of dynamic string data structures. For such situations, researches have proposed techniques such as software prefetch [Karlsson et al., 2000; Callahan et al., 1991], custom memory allocation [Berger et al., 2002; Truong et al., 1998], and different kinds of pointer cache and prefetchers [Yang et al., 2004; Collins et al., 2002; Roth and Sohi, 1999]. Chilimbi et al. [1999b] suggested packing the homogeneous nodes of a BST into contiguous blocks of memory. Rubin et al. [1999] proposed a similar idea for linked lists. These techniques do not eliminate pointers, but improve spatial access locality by grouping nodes that are likely to be accessed contemporaneously.

However, to our knowledge, there has been no practical examination of the impact of these techniques on dynamic pointer-intensive string data structures, nor is there support for these techniques on current platforms. Chilimbi et al. [1999a] note the applicability of their methods to chained hashing, but not with move-to-front on access. This is likely to be a limiting factor, as move-to-front can itself be an effective cost-adaptive reordering scheme [Zobel et al., 2001]. Nor is it clear that such methods will be of benefit in the environments that we are concerned with, where the volume of data being managed may be hundreds of times larger than cache.

Our first technique, which is called a *compact chain*, is a straightforward way of improving the spatial locality of string data structures that use *standard* chains as substructures; by storing each string directly in its node, as opposed to using a string pointer. This approach saves up to 12 bytes of space per string (assuming a typical current system architecture) and eliminates a potential cache-miss at each node access, at no cost other than the difficulty of coding variable-sized nodes. Compact chains can demonstrate the value of eliminating a pointer traversal per node, in contrast to retaining pointers and simply clustering the nodes of a linked list contiguously in memory [Chilimbi et al., 1999a; Rubin et al., 1999].

Our second technique explores 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 likely to be fetched simultaneously and subsequent fetches have high spatial locality. We illustrate the structural differences between standard, clustered, compact, and array-based linked lists in Figure 1.2. While our proposal,

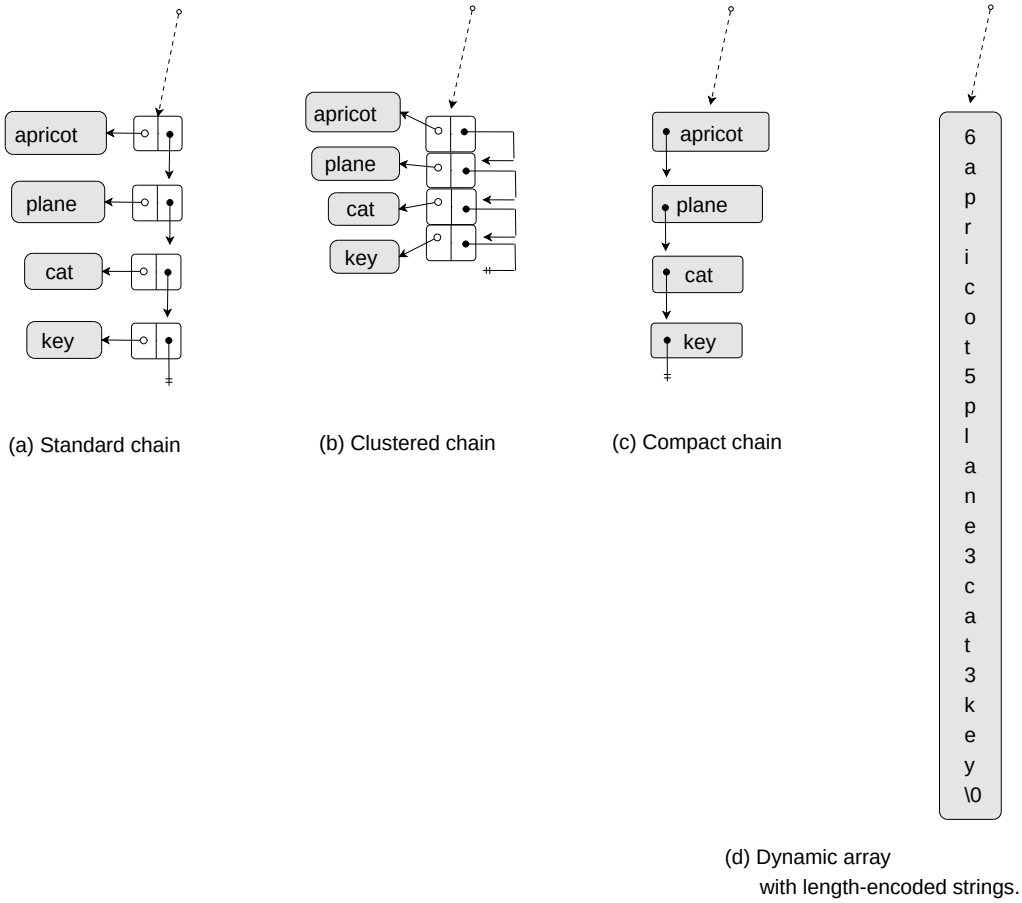


Figure 1.2: A standard chain is the typical implementation of a linked list. A clustered chain physically groups the nodes of a list together, to improve spatial access locality. However, no pointers are eliminated [Chilimbi et al., 1999a]. A compact chain, in contrast, eliminates string pointers by storing the string directly within the node. The dynamic array combines clustering and pointer elimination by storing the strings in a contiguous re-sizable space.

which consists of the elementary step of dumping every string in a list into a contiguous array, might be seen as simplistic, it nonetheless is attractive in the context of current architectures.

A potential disadvantage of using arrays is that, whenever a string is inserted, the array must by dynamically resized. As a consequence, a dynamic array can be computationally expensive to access. However, it is also cache-efficient, which can make the array method dramatically faster to access in practice, while simultaneously reducing space due to pointer elimination.

We experimentally evaluate the effectiveness of our pointer-elimination techniques, using sets of up to 178 million strings. We apply our compact chains to the standard hash table, burst trie, BST, splay tree, and red-black tree, forming *compact* variants. We then replace the chains of the hash table, burst trie, and BST using dynamic arrays, creating new cache-conscious *array* representations called the array hash, array burst trie, and array BST respectively.

On practical machines with reasonable choices for parameters, our experiments show that for all array-based data structures where the array size is bounded, as in the BST and the burst trie, the analytical costs are equivalent to their standard and compact-chain representations. For all the data structures, the expected cache costs of their array representations are superior to their standard and compact equivalents, even for the array hash, where on update, the asymptotic costs are greater than its chaining equivalents, yet in practice greater efficiency is observed.

Clustering is arguably one of the best methods available at improving the cache-efficiency of pointer-intensive data structures [Chilimbi et al., 1999a]. However, to the best of our knowledge there has yet to be a practical evaluation of its effectiveness on string data structures. We experimentally compare our compact and array data structures against a *clustered* hash table, burst trie, and BST. Our experiments measure the time, space, and cache misses incurred by our compact, clustered, and array data structures for the task of inserting and searching large sets of strings. Our baseline consists of a set of current state-of-the-art (standard-chained) data structures: a BST, a TST, a splay tree, a red-black tree, a hash table, a burst trie, the adaptive trie [Acharya et al., 1999], and the Judy data structure. Judy is a trie-based hybrid data structure, composed from a set of existing data structures [Baskins, 2004; Silverstein, 2002; Hewlett-Packard, 2001]. It is structurally similar to a burst trie, but with a key distinction: it dynamically changes the structural representation of its nodes based on the number of strings they store. Judy is arguably faster and more space-efficient than most existing string data structures, but has yet to be experimentally evaluated against the known best.

Our results show that, in an architecture with cache, our array data structures can yield startling improvements over their standard, compact, and clustered chained variants. A search for around 28 million unique strings on a standard hash table that contains these strings, for example, can require over 2300 seconds — using 2^{15} slots — to complete while the table occupies almost a gigabyte of memory. Although this is an artificial case, it demonstrates the fact that random memory accesses are highly inefficient. The equivalent array hash table, however, required less than 80 seconds to search while using less than a third of the space, that is, simultaneously

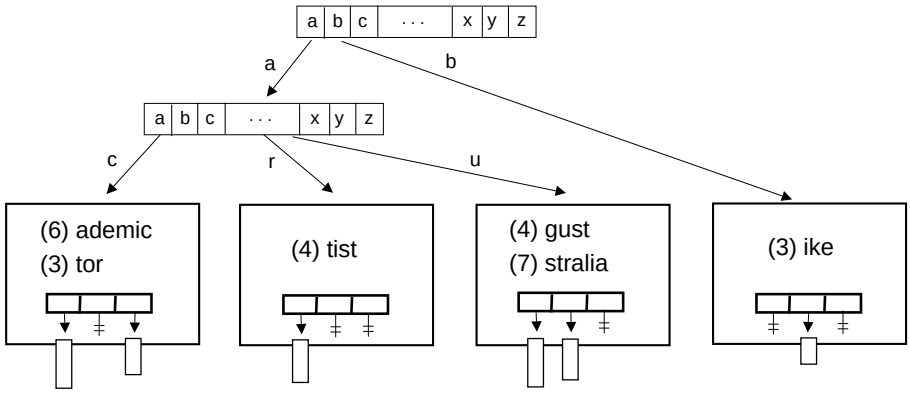


Figure 1.3: We replace the containers of a burst trie with array hash tables to create a new data structure, the HAT-trie. These hashed containers can grow much larger than their chained or dynamic array equivalents without attracting high cache or instruction costs. As a result, the HAT-trie is a more scalable data structure. Strings are length-encoded (shown as brackets).

saving around 97% in time and around 70% in space. Similar savings were obtained for insertion. Hence, despite the high computational costs of growing arrays, our results demonstrate that cache efficiency more than compensates. The array burst trie also demonstrated similar improvements as did the array BST, with up to 74% in speed and up to 80% in space.

The splay tree, red-black tree, TST, and Judy trie were in most cases, slower to access than our array BST. Our compact data structures displayed consistent improvements in both speed and space over their standard variants, and in no cases were the standard data structures superior. The clustered data structures were found to be considerably slower than the standard data structures in the presence of a skew data distribution. Clustering only becomes effective when there is no skew in the data distribution. In such cases, the clustered hash table, burst trie, and BST are slightly faster to access than their compact variants, but are not as space-efficient. Our array-based data structures, however, remain superior.

These results are an illustration of the importance of considering cache in algorithm design. The standard chain hash table, burst trie, and the BST have previously been shown to be the most efficient structures for managing strings [Heinz et al., 2002; Williams et al., 2001; Zobel et al., 2001], but we have greatly reduced their total space consumption while simultaneously reducing access time. These are dramatic results.

Our next contribution is the development of a new dynamic cache-conscious data structure for strings. We introduce the *HAT-trie*, which effectively combines the use of trie nodes and the array hash, to develop an alternative data structure to the burst trie that is faster to access while retaining efficient sorted access to containers. The HAT-trie addresses the major shortcoming

of the burst trie, which is the expense of accessing and growing large containers. Use of large containers is preferable, as it reduces the overall space consumed. However, large containers can be expensive to access, particularly when there is no skew in the data distribution. In such cases, maintaining large containers in a standard, clustered, or compact burst trie can attract high numbers of cache misses. In contrast, the array burst trie will remain cache-efficient, but can incur high computational costs that can impact performance.

By representing containers as array hash tables — which is the core concept of our HAT-trie — we can greatly increase the size of containers without compromising access speed. That is, we can increase scalability. We illustrate the HAT-trie in Figure 1.3. Our results show that, compared to our set of standard, clustered, compact, and array data structures, the HAT-trie is the fastest trie-based data structure for strings, up to 50% faster than the array burst trie. The HAT-trie is not as space-efficient as the array burst trie, due to the space overhead imposed by maintaining a set of slots per container. However with large containers now an option, the HAT-trie is able to rival both the speed and space efficiency of the array hash table, while maintaining efficient sorted access to containers.

We then extend our research to disk-resident data structures for common string processing tasks such as vocabulary accumulation, dictionary management, and document processing. We propose a novel variation of a *B-trie* which can maintain a burst trie efficiently on disk. The standard burst trie is not a suitable candidate because of the way it splits containers: strings are distributed amongst several child containers whose occupancy can be low. Accessing containers at random during a split will attract high performance penalties, due to excessive disk accesses.

While the concept of the B-trie has been outlined in previous literature [Szpankowski, 2001], it has not been formally defined or explored. In particular, we are not aware of any previous discussion of node splitting strategies, which are a critical element of a practical implementation. A major contribution of our B-trie is a novel approach to container splitting which allows the B-trie to reside efficiently on disk, by using a splitting routine that minimizes both the number of containers created and the random accesses that are caused when a container splits.

Suffix trees are well-known trie-based data structures that are also implementable on disk. Suffix trees are designed to support efficient pattern matching on text, but are typically 4–20 times larger than the text they index [Kurtz, 1999]. As a consequence, once moved to disk, they can incur high space and update costs, making them unsuitable for common string processing tasks such as vocabulary accumulation [Grossi and Vitter, 2000; Tian et al., 2005; Cheung et al., 2005].

In such cases, the B^+-tree has been shown to be the most efficient disk-based data structure for external string management [Garcia-Molina et al., 2001; Sedgewick, 1998; Bayer and McCreight, 1972]. Variants of B^+-tree have been successfully applied to many tasks, including spatial and geographic databases, multimedia databases, text retrieval systems, and high-dimensional databases commonly associated with data warehouses [Ooi and Tan, 2002].

We conduct thorough experiments to compare our B-trie against efficient variants of B$^+$-trees. These variants include a prefix B$^+$-tree [Bayer and Unterauer, 1977], where internal nodes only store the shortest distinct prefix of strings that are promoted from leaf nodes, and the Berkeley B$^+$-tree [Oracle, 2007], which is a high-performance open-source B$^+$-tree implementation. Other variants we consider are the string B-tree [Ferragina and Grossi, 1999], which can maintain unbounded length strings efficiently, and a cache-oblivious string B-tree [Bender et al., 2006], which is theoretically designed to perform well on all levels of the memory hierarchy (including disk) without prior knowledge of the size and characteristics of each level [Kumar, 2003].

The B-trie was in most cases superior to the B$^+$-trees, with typical speed gains of 5%–15% and up to 50% in the presence of skew in the data distribution. The B-trie uses more main memory, but the amount of space (memory) involved is small. In most cases, the overall disk space required by the B-trie was less than the equivalent B$^+$-tree, due to the elimination of shared prefixes in containers. Drawing together the themes we have sketched, this thesis is based on four research questions:

- Can we develop cache-conscious alternatives of fundamental string data structures by eliminating pointers?
- How effective are cache-conscious data structures for strings?
- Can we apply our pointer elimination techniques to develop a new data structure for strings that inherits the sorting properties of tries and the speed of hash tables?
- Can a trie-based data structure, such as the burst trie, reside efficiently on disk?

The structure of this thesis is as follows:

Chapter 2: We describe existing string data structures, their structural properties and known results. We then present a detailed discussion of the cache hierarchy used by current processors and describe the fundamental cause of cache-inefficiency in existing string data structures. We then discuss current techniques available to exploit cache.

Chapter 3: We consider the three best current data structures — the hash table, the burst trie and the BST — and describe our proposals for re-designing these structures to exploit cache, using our compact chain and dynamic array techniques. We then present a discussion on the expected cache costs of these data structures, as well as the implications caused by eliminating pointers in string data structures.

Chapter 4: We experimentally evaluate the performance of the standard, clustered, compact, and array data structures against several baseline data structures including the TST and Judy, on large sets of strings with varying distributions and characteristics.

Chapter 5: We introduce the HAT-trie and describe implementation issues, limitations, and expected performance. We then extend the experiments conducted in Chapter 4, to compare the performance of the HAT-trie against the compact-chain and array-based data structures.

Chapter 6: We address the issue of maintaining a burst trie on disk by exploring a novel variation of a B-trie that can efficiently maintain strings on disk. We describe implementation issues and explore its performance for vocabulary accumulation and dictionary management against variants of B^+-tree.

Chapter 7: We summarize the contributions made in this thesis and discuss the results acquired as well as the significance of eliminating pointers in existing string data structures. We summarize the effectiveness of maintaining a trie structure on disk for common string processing tasks, and conclude with discussions on promising directions for future work.

Chapter 2

In-memory data structures for strings

A key decision when developing in-memory computing applications is the choice of a mechanism to store and retrieve variable-length strings. The tools available to programmers for this task are a range of data structures for storing variable-sized objects such as null-terminated strings. In this chapter, we discuss a selection of data structures that are well-known for string management (i.e., insertion, deletion, and equality search), paying particular attention to the current most compact and most efficient options — the chaining hash table with move-to-front, the burst trie, and arguably, the binary search tree.

We highlight the differences of structural properties among these data structures, along with their performance in respect to time and space and their practicality for many common string processing tasks, such as index construction, dictionaries, and vocabulary accumulation. It is important to understand the differences as the choice of data structure often governs the performance of the application. A search engine for a large volume of text, for example, must manage millions or tens of millions of strings efficiently. In this example, the data structure used should provide efficient access to strings, particularly under skew — where some strings are accessed more than others — while conserving memory usage. Another example is a string dictionary, where a data structure with a bound on access time should be used.

Our focus, however, is to address the problem imposed by the increasing access latencies between the CPU and main memory, and to investigate how to exploit caches on current processors to help to alleviate these costs. Typical implementations of the chaining hash table with move-to-front, the burst trie, and the binary search tree use large numbers of nodes and pointers; this is not efficient in use of cache, and can thus incur severe performance penalties.

We discuss recent research that improves the cache usage of existing data structures or in-

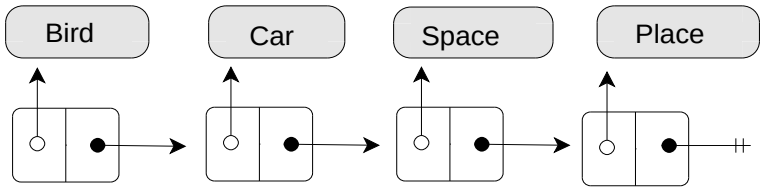


Figure 2.1: The strings "Bird", "Car", "Space", and "Place", are inserted into a simple unsorted linked list. A string can only be inserted when it does not exist in the chain (assuming that duplicates are not maintained). A string is inserted by encapsulating it into a node, which is then appended to the chain.

troduces new structures that can exploit cache while remaining computationally efficient. These approaches are called *cache-conscious*, *cache-aware*, *cache-efficient*, or *cache-friendly* [Chilimbi et al., 1999a; Acharya et al., 1999; Ladner et al., 2002]. Current cache-conscious data structures, however, have unfortunately yielded small gains in performance and have limited or no applicability in situations where large and dynamic sets of variable-length strings are to be managed. As we explain, the best existing data structures for strings can be highly inefficient on current processors.

2.1 Linked lists and arrays

The simplest dynamic data structure is the linked list. Developed in 1955 by Newel, Shaw, and Simon for use in their Information Processing Language [Newell and Tonge, 1960], it consists of a chain of nodes where each node stores a pointer to a data item (in our case a string), along with a pointer to the next node in the chain. To store a string, we traverse the chain and access every node to check whether it matches the string to insert. When a match is not found, the string can then be encapsulated in a node and attached to the head or tail of the list. Linked lists require on average and at worst $O(N)$ comparisons per search, where N is the number of strings. We show an example in Figure 2.1. Variations of the linked list include the doubly-linked linked list, where nodes have a *to* and *from* pointer, and the double-linked list, where each node contains a pointer to another independent list [Newell and Tonge, 1960].

The linked list quickly became popular in computer science [Green, 1961; Raphael, 1966] and was used to develop several programming languages such as COMIT and LISP, as well as file structures for operating systems. Linked lists are, however, impractical for tasks such as vocabulary accumulation, due to their inability to scale. A *scalable* data structure is one that remains efficient to access as the number of data items stored increase. When used as a queue, stack, last-in-first-out, or first-in-first-out structure, the linked list is very efficient because in a typical implementation, only the first or last nodes are manipulated.

Strings can also be stored sequentially using a fixed-size or dynamic array, with an access cost

of O(N) if the strings are not sorted, or an insertion cost of O(N) if the strings are sorted. An array of homogeneous data items is also known as a compressed list [Rubin et al., 1999]. The cost of resizing a dynamic array must also be taken into consideration; this involves accessing every string which can be expensive for large N. Hansen [1981] describes several schemes for storing strings in fixed-sized nodes (or arrays). The techniques include middle gap, binary search, partitioned, and square root structures. These are designed to better utilize the space in a node, at the cost of some search performance. With the middle gap scheme, for example, strings of fixed-length are sorted and partitioned into several groups separated by unused space. When a new string is inserted into a group, the existing strings must be moved to maintain sort order (to allow binary search).

An array is well suited for use as a queue, stack, LIFO, and a FIFO structure, but, being similar to a linked list, it cannot scale and is thus inefficient for use in common string processing tasks, where typically millions of strings are processed. However, arrays, unlike linked lists, have properties that can make good use of cache. We describe and analyze these properties in Chapter 3, and show how a dynamic array can be applied to most existing string data structures, in particular the current best, to greatly improve the utilization of both cache and space.

2.2 Trees

Trees combine the search characteristics of arrays with the dynamic characteristics of lists to achieve efficient and scalable performance. They are popular for tasks such as dictionaries, indexes, and vocabulary accumulation. We discuss several well-known tree structures below.

2.2.1 Binary search tree

In a *standard* binary search tree (BST), each node has a left and right child-node pointer to subtrees of strings that are respectively, lexicographically less than or greater than the string that is contained by the node. To search for a string, we conduct a binary search starting from the root node moving down the tree until the desired node (its string) is found, or until a null pointer is encountered. In the latter case, we can insert the string by encapsulating it into a BST node, which is then attached to the pointer that caused the search to fail.

The standard BST has an expected O($\log N$) search cost, but with an $O(N)$ worst case [Nievergelt, 1974]. However, with strings as keys and assuming that strings share long common prefixes, traversing a binary search tree can involve redundant prefix comparisons which can slow the performance of search by a multiplicative factor [Crescenzi et al., 2003]. Although its average case performance is attractive, its worst-case makes it undesirable in many applications that require a bound on access time; for example, a dictionary lookup should be fast for all searches. To ensure good performance regardless of the distribution of strings, it is necessary to balance or reorganize the tree. However, the cost of reorganization can outweigh the gains in the average case, as

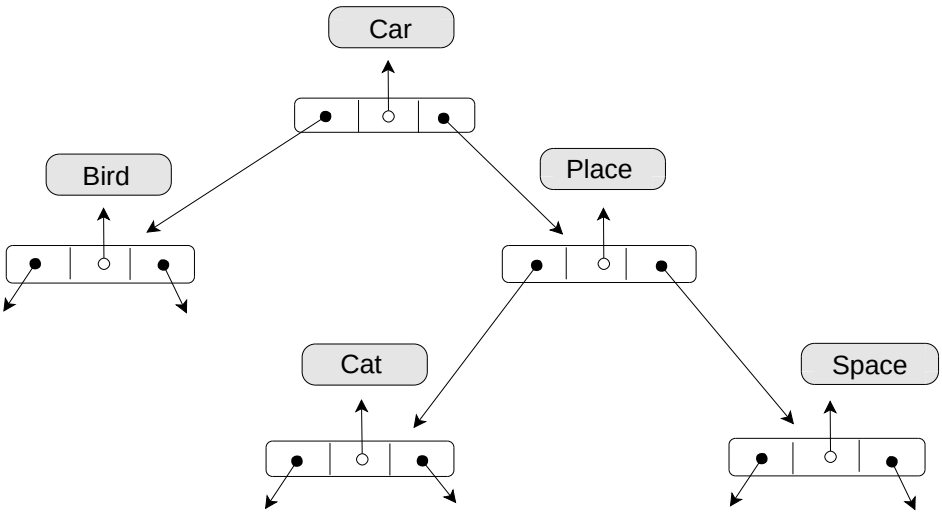


Figure 2.2: The strings "Car", "Bird", "Place", "Cat", and "Space", are inserted into a standard binary search tree.

frequently accessed strings may be moved away from the root, and reorganization can be more expensive overall, than search. We discuss such variants below.

2.2.2 AVL tree

The first proposal for a height-balanced binary search tree was the AVL tree [Adelson-Velskii and Landis, 1962], named after its two inventors. The AVL tree is an example of a balanced BST where for any node, the height of the left and right subtrees may differ by at most 1. A flag is kept within each node to indicate which of its two children, if any, violate this constraint. Once a new string is inserted — as described for the standard binary search tree — the steps taken are retraced back up to the root, rotating any nodes with subtrees that have become unbalanced during the insertion.

A node is rotated by exchanging it with its parent. This will adjust the structure of the binary search tree, but will not interfere with the order of elements [Sedgewick, 1998]. There are two basic cases where adjustments are needed in an AVL tree; each of these cases has a corresponding mirror rotation. The first case is satisfied with a single rotation, whereas the second requires two rotations to balance a subtree. By enforcing this constraint, the AVL tree can maintain a binary search tree that is balanced to offer a worst-case performance of $O(\log N)$ comparisons per search, regardless of the distribution of data. The AVL tree is theoretically sound, but is generally not used in practice because the alternatives — described below — are more efficient.

2.2.3 Red-black tree

The red-black BST [Bayer, 1972; Guibas and Sedgewick, 1978] maintains a *color* attribute in every node: new nodes are colored red, the root node is always black, red nodes must have a black parent, and every path has the same number of black nodes. These constraints ensure that the depth of any leaf node is no more than $2 \log N$. First proposed as a symmetric binary B-tree [Bayer, 1972], it was later refined and given its modern name by Guibas and Sedgewick [1978]. When a node is inserted — as described for the standard binary search tree — the steps are retraced back up the tree (but not necessarily up to the root), rotating nodes and flipping colors to maintain these constraints. Tree maintenance during insertion has a constant overhead with at most two rotations per insertion; k insertions require at most k changes of node color [Williams et al., 2001].

2.2.4 Splay tree

The splay tree [Sleator and Tarjan, 1985] is an example of a self-adjusting binary search tree that achieves good worst-case behavior through *splaying*, a technique that moves the most recently accessed node to the root of the tree. While the tree is not balanced, the splaying operation can achieve worst-case $O(k \cdot \log N)$ access time, over k accesses and insertions (for sufficiently large k), while making the worst-case cost of $O(N)$ highly unlikely [Williams et al., 2001; Heinz and Zobel, 2002]. A key distinction between the splay tree and the AVL or red-black tree is that it modifies the tree after every successful search. Another distinction is that nodes do not need to maintain additional information to support splaying. There are three primary methods of splaying: *zig*, *zig-zig* and *zig-zag*, and three others that are reflections of these.

A splaying method is chosen by checking the orientation of a node against its parent and grandparent, that is, whether it is a *left* or *right* child. Figure 2.3 illustrates the application of these methods on a binary search tree. Further examples are described by Sleator and Tarjan. After an insertion or successful search — when a node is found that matches the query (the candidate node) — a splaying operation is performed; if the parent of the candidate node is the root node, then only a single rotation — a *zig* — is needed. When the parent is not the root, then a double rotation is required. A *zig-zig* occurs when both the candidate node and its parent are either *left* or *right* children of the grandparent. In such a case, the parent is rotated first, followed by the candidate node. A *zig-zag* occurs when the candidate node is a left child of its parent and the parent node is the right child of its parent. In such a case, the candidate node is rotated first, followed by its parent. This splaying process propagates up the tree until the candidate node becomes the new root.

Although splaying is effective in theory, the cost of rotating nodes on every access can prove excessive, especially when maintaining a large number of strings. Williams et al. [2001] proposed the use of heuristics, the most effective being to splay the tree periodically rather than on every access, giving the n-splay tree — where n is the number of successful searches before a splay is

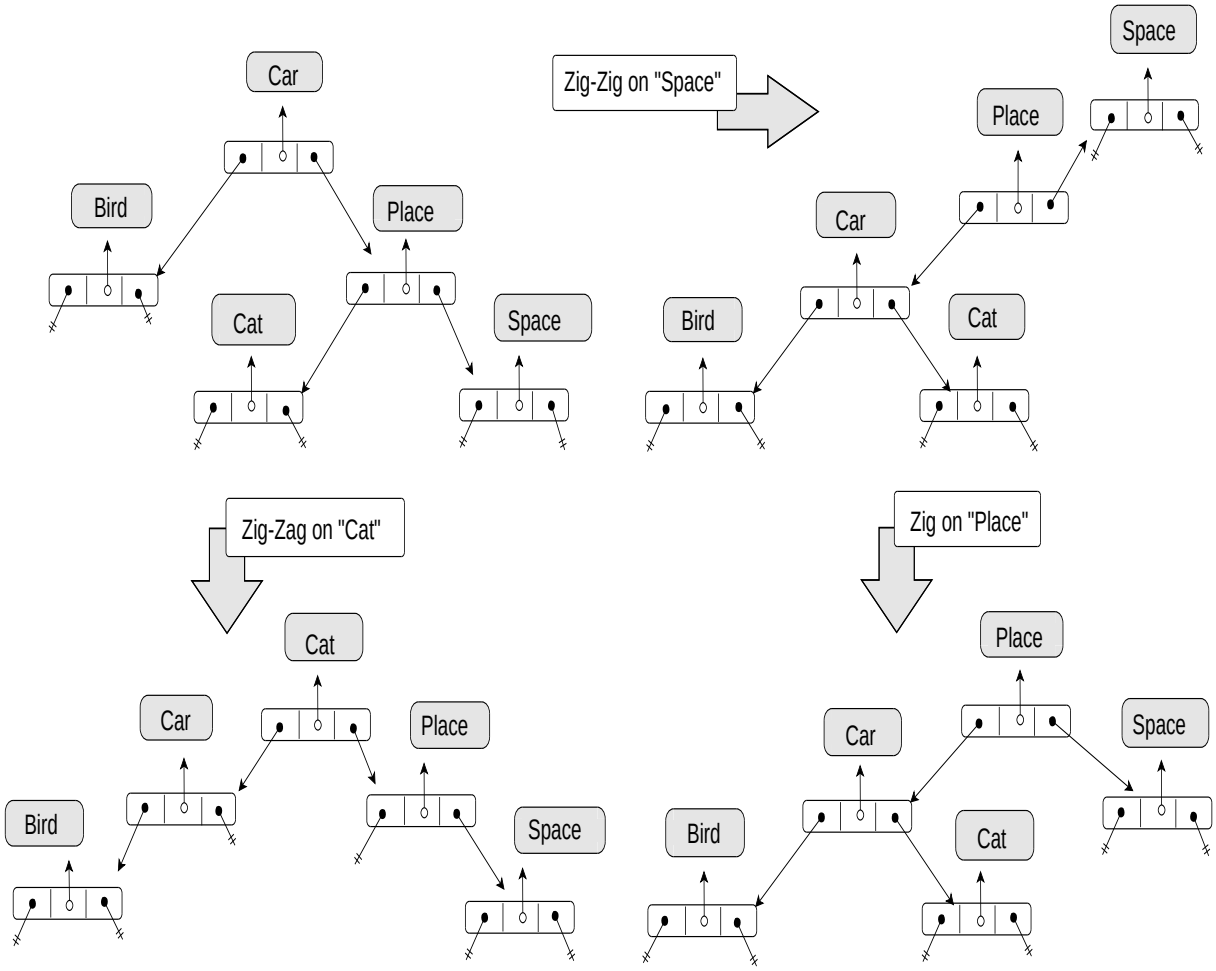


Figure 2.3: The strings "Car", "Bird", "Place", "Cat", and "Space", are inserted into a splay tree. In this example, we assume that no splaying is performed during construction, hence, the tree begins as a standard BST. We then splay on search. A binary search for the word "Space" leads to the right-most child-node. To make this node the new root, a Zig-Zig is required, that is, the parent node is rotated first. We then search for "Place", which requires a single-rotation or a Zig to move it to the root. Alternatively from the original BST, we can search for the string "Cat" which requires a Zig-Zag to move it to the root, which is similar to a Zig-Zig except that the child node is rotated first.

triggered. However, as we discuss later, even periodic splaying is expensive compared to the access cost of a standard BST.

2.2.5 Randomized binary search tree

The randomized binary search tree [Martinez and Roura, 1998] is another self-adjusting BST where a heap order is maintained on nodes. A node's weight (a positive random number) is generated on initial allocation, and is then updated when it matches a string during search. When a node's weight is greater than its parent, it is moved up the tree using single rotations until the heap order is restored. In contrast to the splay tree, however, each node must store its weight, and so, the overall space requirements of a randomized BST is higher. However, far fewer rotations are performed, thus avoiding the principal bottleneck of splaying. A randomized BST is illustrated in Figure 2.4.

Several modifications and heuristics have been proposed to improve the time and space bounds of the randomized search tree [Seidel and Aragon, 1996; Martinez and Roura, 1998]. A parent pointer can be maintained in each node to permit a simpler bottom-up maintenance algorithm, which would improve the efficiency of rotations, but at a cost in space. To save space, a hash function can be used to generate a weight on-the-fly, by using the node's key. Similarly, a weight can be derived as a function of the size of a nodes subtrees. These techniques save space but at the expense of access time. Another randomized structure is a skip list [Pugh, 1990], which uses a hierarchy of sorted lists above a primary sorted list, giving tree-like expected behavior but with relatively poor efficiency in practice with variable-length strings [Williams et al., 2001; Ciriani et al., 2002; Ergun et al., 2001].

2.3 Tries

A trie is a multi-way tree structure that stores sets of strings by successively partitioning them across letters of an alphabet [de la Briandais, 1959; Fredkin, 1960; Jacquet and Szpankowski, 1991]. It was originally proposed by de la Briandais [1959] and was later called a *trie* by Fredkin [1960]. Tries have two properties that cannot be easily imposed on data structures based on binary search. First, strings are clustered by shared prefix. Second, there is an absence of (or great reduction in the number of) string comparisons — an important characteristic, as a major obstacle to efficiency is excessive numbers of string comparisons [Williams et al., 2001]. The binary search tree and its variants have on average logarithmic search costs, but impose a mandatory string comparison on every node accessed, which can be expensive with long strings that share long prefixes [Crescenzi et al., 2003]; a typical example being website URLs.

Tries, in contrast, can be rapidly traversed and offer good worst-case performance without the overhead of balancing or binary search [Szpankowski, 1991; Devroye, 1992; Knuth, 1998]. Tries have been applied to a range of applications including text compression [Bell et al., 1990],

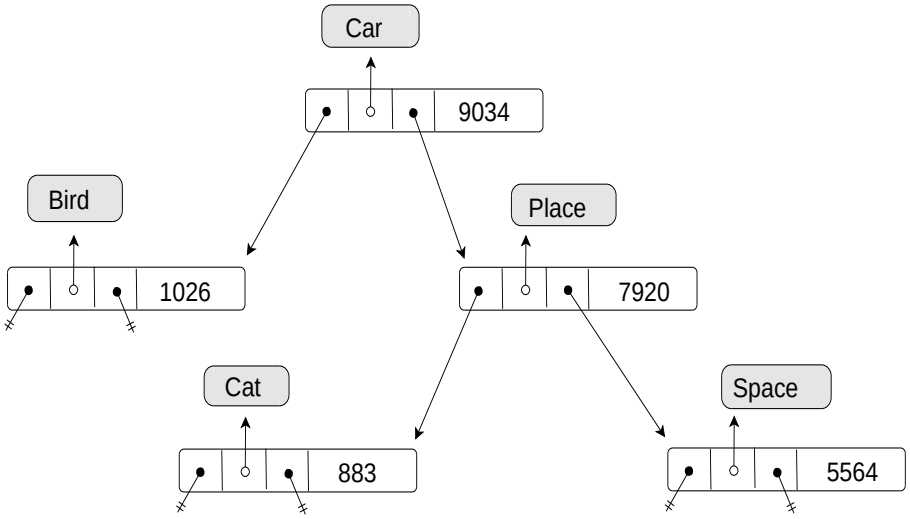


Figure 2.4: A randomized binary search tree containing the strings "Car", "Bird", "Place", "Cat", and "Space". A node's weight is randomly allocated when it successfully matches a query. If the node's weight exceeds its parent, it is rotated up the tree until it becomes the new root node or until a parent is encountered with a larger weight.

dictionary management [Aoe et al., 1992; Knuth, 1998], pattern matching [Flajolet and Puech, 1986], natural language processing [Baeza-Yates and Gonnet, 1996], compilers [Aho et al., 1986], and IP routing [Nilsson and Karlsson, 1999].

Although fast (the access cost equals the length of the string) tries are space-intensive, which becomes a serious problem in practice [Comer, 1979b; Heinz et al., 2002; McCreight, 1976] — their use has been restricted to tasks that do not involve large volumes of strings. Measures must be taken to reduce the space consumed, if the trie is to remain applicable to current applications that typically manipulate many millions of strings. There are two approaches to reducing the size of a trie. One is to reduce the size of each trie node by changing its structure. The other is to reduce the number of nodes. We discuss these variants below.

2.3.1 The array trie

A simple implementation of a trie node is an array of pointers, one for each letter of the alphabet [Fredkin, 1960]. A string of length k is stored as a chain of k nodes. Storing strings in this manner forms an array trie that branches from a single root, as illustrated in Figure 2.5. Access to a trie node involves taking the lead character of a string as an offset to acquire a pointer. The lead character is then removed and the next trie node is accessed (or created). The search or insertion procedure terminates once the string is exhausted, in which case, the last node accessed or created

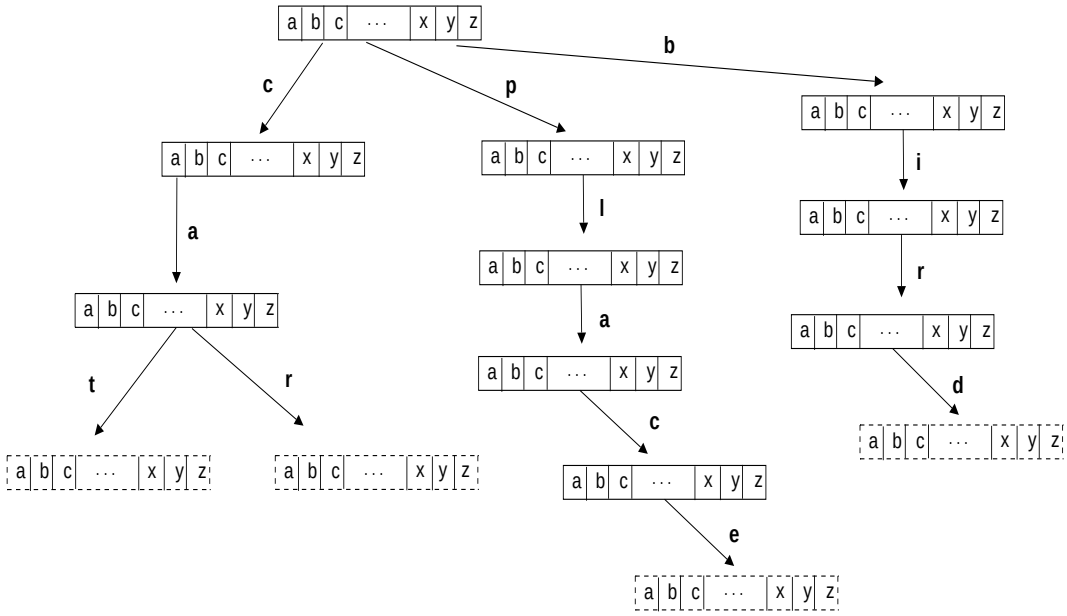


Figure 2.5: The strings "Cat", "Car", "Place", and "Bird" are inserted into an array trie. The dashed trie nodes have their end-of-string flags set to indicate that they have consumed or exhausted a string, which represents the path taken to acquire the node.

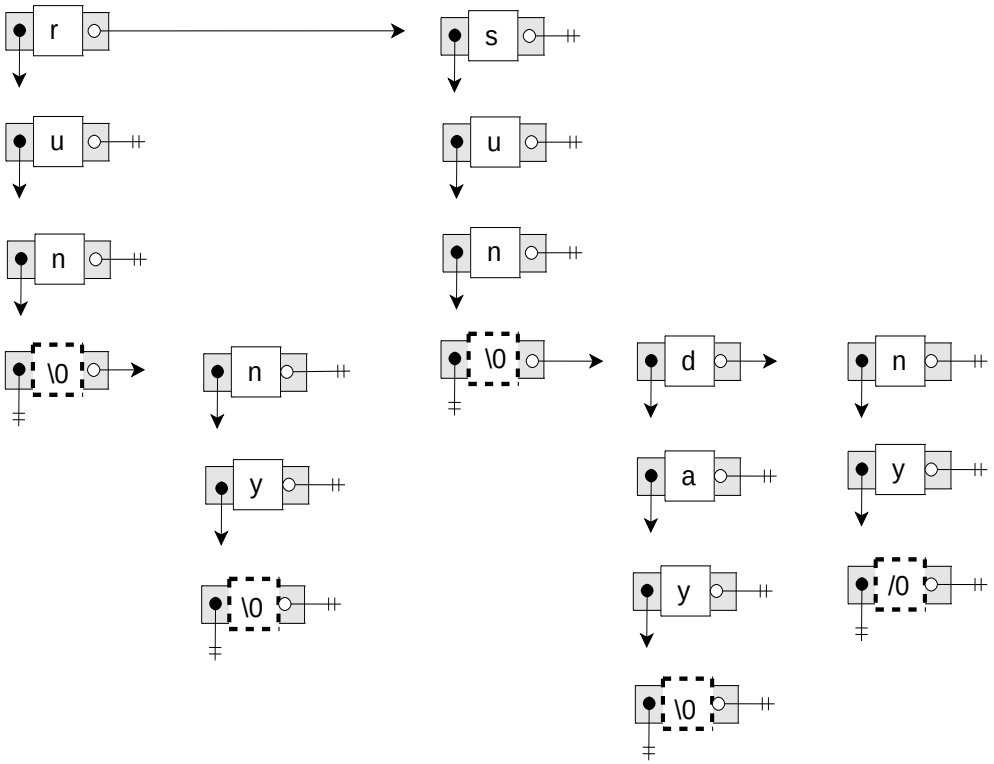


Figure 2.6: The strings "run", "sunday", "runny", "sun", and "sunny", are inserted into a list trie. The list trie is identical to an array trie, except that a trie node is represented as a linked list where each node contains a sibling pointer and a child pointer. A sibling pointer points to the next character in the trie node, whereas a child pointer points to the next trie node. Sibling pointers are horizontal. The dashed nodes have their end-of-string flags set to indicate that they have consumed or exhausted a string.

will have its end-of-string flag checked on search, and set on insertion. In a typical implementation, the end-of-string flag is represented by utilizing the space of an unused pointer. Access to these nodes can be rapid, even for long chains. However, with tasks such as vocabulary accumulation, for trie nodes at or near the leaves, the majority of pointers are likely to be unused, which wastes space.

2.3.2 The list trie

The de la Briandais [1959] trie, also called a list trie [Knuth, 1998], is perhaps the simplest approach at reducing the space-intensity of the array trie, by changing the representation of nodes to linked lists that store only non-null pointers. The array trie allocates space for A pointers per node,

where A is the size of the alphabet. A list trie, in contrast, creates only as many pointers as needed. That is, a trie node is structured as a linked list, where each node in the list stores a character, a pointer to a sibling node, and a pointer to a next trie node. Sibling nodes represent the characters used by the current trie node. An example of a list trie is shown in Figure 2.6.

Search proceeds as described for the array trie, except that when a trie node is accessed, its sibling nodes are traversed until a match for the required character is found. Once found, the lead character of the string is removed and the corresponding next node pointer is followed. The list trie is traversed in this manner until all lead characters are exhausted. A search fails if a sibling node is not found or when the string is exhausted and the acquired end-of-string flag is not set. String insertion proceeds in a similar manner, except that when a sibling node is not found, it is created, and when the string is exhausted, the end-of-string flag is set in the last trie node accessed, which completes the insertion.

The list trie can save space over an array trie under the assumption that each trie node has only a few children and that an array representation would consist largely of unused pointers. However, the space is saved at the considerable expense of access time, due to the mandatory task of traversing a linked list per trie node [Severance, 1974].

2.3.3 Ternary search trie

Another approach to eliminating unused pointers in a trie is to structurally represent the sibling nodes in a list trie as a binary search tree [Bentley and Sedgewick, 1997]. This will permit logarithmic access costs, on average, to search sibling nodes. An example of a TST is shown in Figure 2.7. Every node in a TST stores a character c and three child pointers: left, middle, and right. On access, the lead character of a string is compared. If it matches to c, the lead character from the string is removed, and the middle pointer is followed to acquire the next trie node. On a mismatch, the left child is followed if the lead character (which is then consumed) is lexicographically smaller than c, otherwise the right pointer is followed. The process continues until the string is exhausted, whereby the end-of-string flag is set in the respective node. With three pointers per node, the TST is not as compact as the list trie, but is more efficient to access and can require less space than the array trie.

Badr and Oommen [2005] explored the application of splaying [Sleator and Tarjan, 1985], conditional rotations [Cheetham et al., 1993], and randomized rotations [Martinez and Roura, 1998] on the TST. Representing the trie nodes of TST as a splay tree or a randomized BST was found, in most cases, to lead to poor performance relative to trie nodes that were structured as a standard BST. Use of conditional rotations (which restructures a BST periodically using weight-based heuristics), however, showed small gains in performance but at a cost in space.

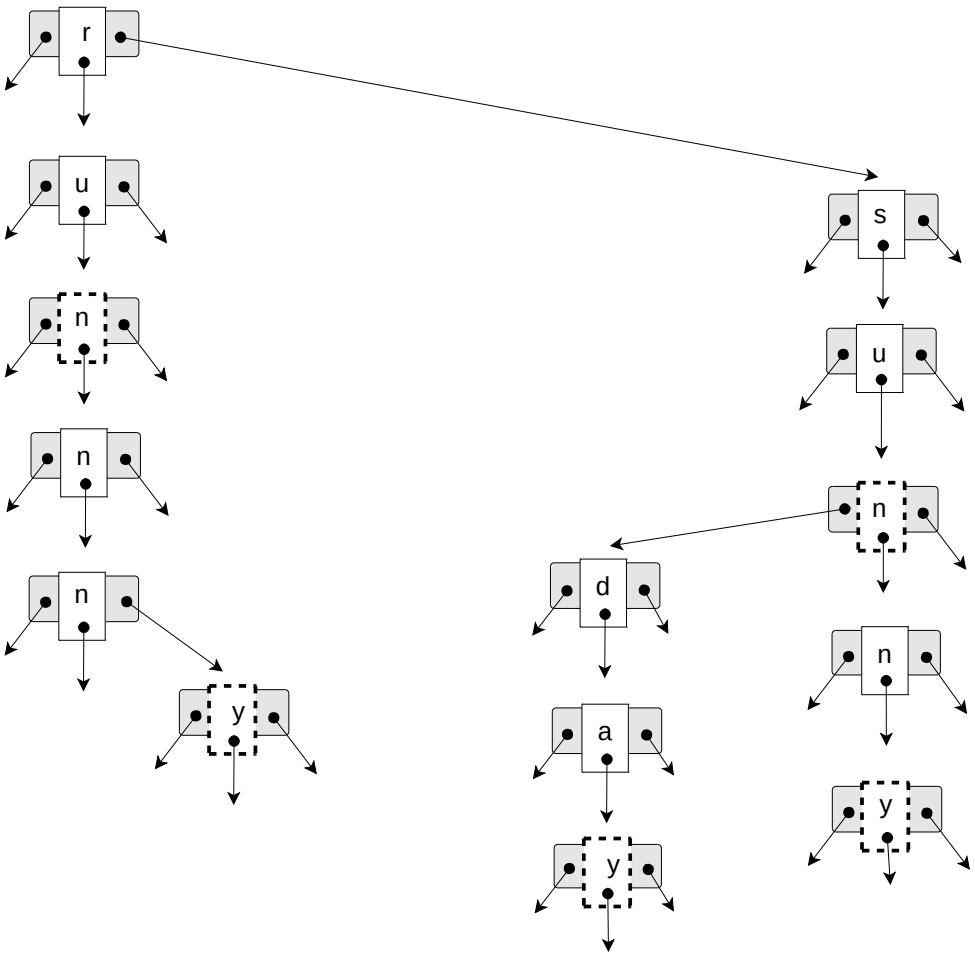


Figure 2.7: The strings "run", "sunday", "runny", "sun", and "sunny", are inserted into a ternary search trie. The TST is similar to the list trie, except that trie nodes are structured as binary search trees. That is, each trie node contains three pointers, with the middle pointer used to point to the next trie node, and the left and right pointers to sibling nodes that are lexicographically less than or greater than the character represented by the node. The dashed nodes have their end-of-string flags set to indicate that they have consumed or exhausted a string.

2.3.4 Compact trie

At the cost of a small increase in complexity, the array trie can be modified to exploit the fact that in natural-language applications, such as dictionary management or vocabulary accumulation, nodes near the leaves of the tree are likely to be sparse. The array trie can be reduced in size, by omitting chains of nodes or (*sticks*) that descend to a single leaf. This approach is known as the compact trie [Sussenguth, 1963] and has been shown to be effective at reducing space while retaining rapid access to nodes. Despite this improvement however, the compact trie remains a space-intensive data structure, especially with strings that do not have a typical natural-language distribution, such as genome data [Heinz et al., 2002].

2.3.5 Patricia trie

The Patricia trie [Morrison, 1968; Gonnet and Baeza-Yates, 1991] extends the idea of a compact trie by omitting all single-descendent nodes, not just those that descend directly into leaves. It achieves this by storing a counter at each branch of the trie that is used to indicate how many elements of a query can be skipped before determining the next pointer to follow. The Patricia trie evaluates the lead bits of a string, rather than its bytes or characters. This is a type of digital trie, where nodes contain pointers representing 0 and 1 which lead to unique string suffixes stored in leaves.

Digital tries are potentially more compact than character-based trie structures and are useful in situations where space is highly restrictive. However, they are generally slow, requiring between 8 to 32 pointer traversals even for short strings, which is an unacceptable cost when compared to that of a compact or array trie [Sedgewick, 1998; Flajolet and Sedgewick, 1986; Rais et al., 1993]. The Patricia trie can easily be modified to evaluate characters, eliminating this inefficiency. However, the space saved by the Patricia trie is offset by the complexity of its structure. Furthermore, as a result of skipping lead bits or characters during search, there is the possibility of a false match. As a result, once a leaf node is acquired, a string comparison is required to determine whether the suffix stored in the leaf is a match. The requirement of a string comparison is expensive compared to the comparison-less traversals of the array and compact tries. A similar digital trie is the LPC-trie [Nilsson and Tikkanen, 1998], which employs level compression in addition to the path compression of Patricia tries. Although more space-efficient, the LPC-trie is slower and more expensive to maintain with large string datasets [Heinz et al., 2002].

2.3.6 Burst trie

The techniques presented above are insufficiently effective for large and dynamic sets of strings, due to the high costs of time and space involved. In the burst trie [Heinz et al., 2002], dynamic containers are used to store small sets of strings that share a common prefix. Containers — such as linked lists with *move-to-front* — can reduce the number of trie nodes by up to 80%, at little

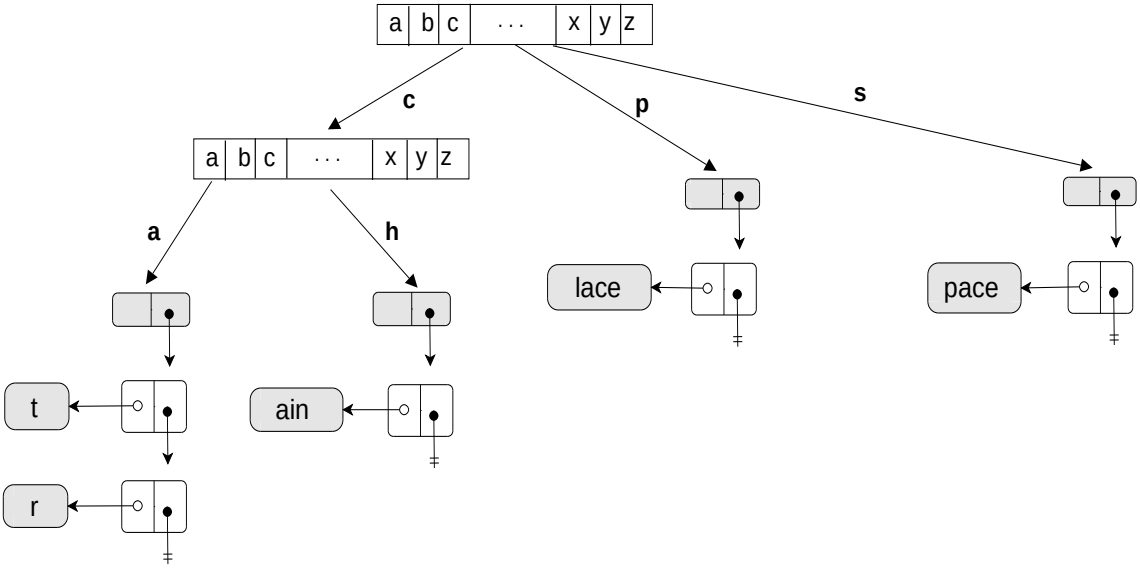


Figure 2.8: The strings "cat", "car", "chain", "place", and "space" are inserted into a standard burst trie with a container threshold or capacity of 2 strings.

to no cost in speed. Move-to-front on access, also known as a self-adjusting list, is an effective cost-adaptive scheme that moves the node that matched a query during search to the start of the list [Knuth, 1998].

A similar approach to the burst trie takes a fully-built array trie and collapses selective nodes into static containers [Ramesh et al., 1989]. The burst trie, however, is built dynamically. An example of a burst trie is shown in Figure 2.8. It starts with a single container that is populated with strings until full. Once full, the container is *burst*, forming smaller containers — one for each letter of the alphabet — that are parented by a new node. Strings are distributed according to their lead character, which can then be removed. An example of how to burst a container is shown in Figure 2.9. Unlike the array and compact tries, an insertion does not necessarily proceed until the string is exhausted; once a container is acquired, the lead character is removed and the remaining characters of the string — the suffix — are compared to those stored in the container. If the search fails, then the string is appended to the container.

The burst trie uses array trie nodes with an alphabet of 128 characters — mapping to the 128 characters of the ASCII table. Alternative trie structures such as the Patricia trie can be used for situations where space is highly restrictive. However, Heinz et al. [2002] showed that the number of trie nodes, and thus the space overhead, can be kept low by containers. Hence, the space saved by employing a Patricia trie instead of an array trie, for example, was too small to justify tolerating its high access cost.

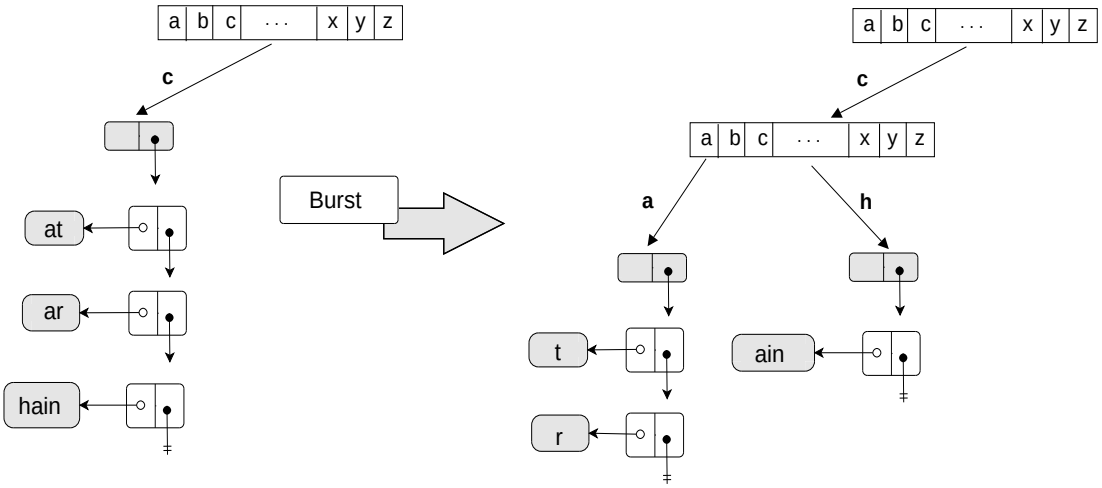


Figure 2.9: We show how a container is burst in a burst trie. First, the container is detached from its parent and the parent pointer is assigned to a new parent trie node. We then distribute the strings in the container to at most A new containers that are anchored from the new parent trie node, based on the leading character of each string, which can then be removed.

Bursting is computationally efficient and presents three useful properties. First, it can efficiently fragment a large container into smaller containers. The smaller containers — linked lists — store fewer strings and are thus more efficient to access. Second, the number of trie nodes created can be successfully throttled; a trie node is created only when a container is burst, which can be infrequent depending on the distribution of strings and the capacity of containers. Third, bursting will remove the lead character from every string in the container, which saves space and reduces the cost of string comparisons as the structure grows in size (shorter strings are generally faster to compare). A burst can be triggered using several heuristics. The simplest and most effective being a *limit*, which bursts a container whenever it stores more than a fixed number of strings. Other heuristics include *ratio* and *trend*, but these were generally less effective.

The performance of the burst trie was shown to be dependent on the distribution of strings, with the worst-case observed using genomic data where the strings are of equal-length and have a near uniform probability distribution — a distribution of strings that is exceedingly rare in natural language applications. However, even its worst-case remained comparable to other tree-based data structures.

2.3.7 Judy

Judy is a trie-based data structure developed by Doug Baskins at Hewlett-Packard Research labs [Baskins, 2004; Hewlett-Packard, 2001]. The structural design of Judy is similar to the burst

trie, but with a key difference: it dynamically changes the structural representation of its nodes in accordance to the number of data items they store. Judy can therefore be considered as a hybrid data structure, composed from a set of existing data structures and algorithms. Judy is able to maintain efficient sort access to data items and logarithmic access costs. Further implementation details are discussed by Silverstein [2002].

Judy has been claimed to be faster and more space-efficient than many of the existing data structures in computer science, such as variants of BST and hash tables (described below) [Hewlett-Packard, 2001]. However, it has yet to be compared against well-known string data structures such the burst trie, for common string processing tasks such as accumulating the vocabulary of a large text collection in memory. Fritchie [2003] has evaluated the performance of the Judy trie using fixed-length keys. Judy was found to be faster to search than the variants of BST and hash tables (without move-to-front), but was often slower to update. We experimentally evaluate the performance of the Judy trie for common string processing tasks against a baseline of high-performance string data structures in Chapter 4.

2.3.8 Alternative trie structures

There are other ways at reducing the size of a trie, however they are generally of academic interest and are slower, less effective, and more complicated than the more popular methods discussed. An example includes changing the order in which the characters of a query are evaluated during search [Comer and Sethi, 1977]. In this approach, the query string is viewed as a tuple of k attributes, where k is the length of the query. Changing the order of evaluation can influence the overall size of a trie structure, because unused pointers can be reduced or eliminated. The task of finding the sequence of characters that would lead to the smallest structure is, however, NP-complete. Furthermore, it is only applicable with *static* data (where the strings are known in advance). With the use of heuristics, however, it is possible to approximate a space-efficient static trie [Comer, 1981]. Another technique involves overlaying the nodes of a static trie in an attempt to make use of the space occupied by unused pointers [Al-Suwaiyel and Horowitz, 1984].

Aoe et al. [1996] proposed an algorithm for trie compaction that builds two tries using the leading and trailing characters of strings, based on the observation that strings typically share common prefixes and suffixes. However, the two-trie structure remained space-intensive relative to the compact trie, and is thus not a practical solution for managing a large set of strings [Heinz et al., 2002].

2.4 Hash tables

A hash table is a data structure that scatters keys amongst a set of lists that are anchored over an array of slots. 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. To store a string in a

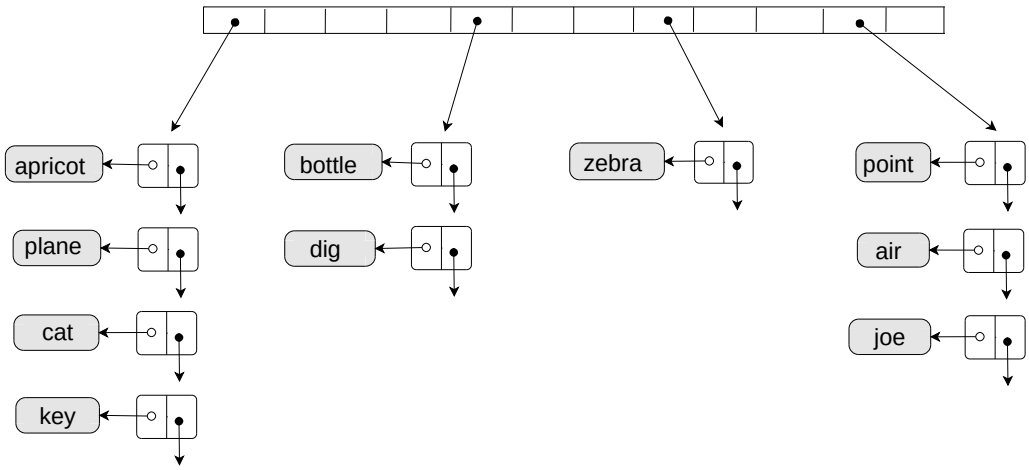


Figure 2.10: The strings "apricot", "plane", "cat", "key", "bottle", "dig", "zebra", "point", "air", and "joe" are inserted into a standard hash table. A standard hash table uses a bitwise hash function [Ramakrishna and Zobel, 1997] to distribute strings across slots. Collisions — where more than one string is hashed to a slot — are resolved by appending the strings to a standard linked list with move-to-front on access. Each slot maintains its own list.

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 a hash function that can uniquely distinguish keys that are not known in advance [Knuth, 1998]. In principle, the cost of searching a hash table depends only on load average (that is, the ratio of items to slots) — though, as we discuss in the next chapter, factors such as cache efficiency can be more important — and thus hashing is asymptotically expected to be more efficient than a tree. An example of a hash table is shown in Figure 2.10.

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 [Sarwate, 1980], 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 from a universal class is the bitwise method of Ramakrishna and Zobel [1997]. Hash functions commonly described in text books are based upon a modulo, and repeated multiplication and division, which are simply too slow and are not effective at distributing strings [Sedgewick, 1998; Heinz et al., 2002].

Since the origin of hashing — proposed by H.P. Luhn in 1953 [Knuth, 1998] — 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.

Open addressing, proposed by Peterson [1957], is a class of methods where items are stored directly in slots and collisions are resolved by searching for another vacant slot. The simplest scheme is called *linear probing*, where the hash table is scanned from left to right until a vacancy is found. Instead of scanning, heuristics can be used to guess the location of a vacant slot. This is known as *double hashing*; another hash function independent of the first is used to skip to a location that may be vacant. This approach requires, on average, fewer probes than linear probing. Another is a *reordering scheme* [Brent, 1973] that moves items around to reduce expected probe costs. *Cuckoo hashing* [Pagh and Rodler, 2004] is another open addressing scheme where two hash tables and two hash functions are used to manage fixed-length keys in-memory. A key is stored in either one of the hash tables, but not both. When a collision occurs in the first hash table (using the first hash function), the key occupying the slot is evicted and replaced by the new key. The evicted key is then hashed using the second hash function and stored in the second hash table. If this causes another collision, then the eviction process repeats until all keys are hashed without collision. However, as the load average approaches 1, the performance of open addressing drastically declines.

The linear probing and doubling hashing open addressing schemes were surveyed and analyzed by Munro and Celis [1986], and were found to be unsuitable for applications that manage moderate to large sets of dynamic and variable-length strings. Alternative techniques include coalesced chaining, which allows lists to coalesce to reduce memory wasted by unused slots [Vitter, 1983], and to combine chaining and open-addressing, known as pseudo-chaining [Halatsis and Philokyprou, 1978]. However, 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 [Larson, 1982] and extensible hashing [Rathi et al., 1991] — which allow an on-disk hash table to grow and shrink gracefully. The primary disadvantage of hashing is that strings are randomly distributed among slots. This means that hashing is not a candidate structure for applications in where it is preferable that strings be available in sort order. In most situations, however, maintaining strings in sort order is an unnecessary overhead because sorted access to strings is never or infrequently required. In the latter case, a fast sorting algorithm such as burst sort may suffice [Sinha and Zobel, 2005].

2.5 Comparisons of well-known data structures

Williams et al. [2001] experimentally compared the performance of the splay tree, the randomized BST, and the red-black tree, against the standard BST for task of vocabulary accumulation. For this data, the splay tree was never faster than a standard BST, and was 25% slower on average.

Furthermore, splaying on every access proved ineffective for skew data, with words such as "the" — the most frequent word in plain text — sometimes found deep in the tree. Similarly, the randomized BST was only 3% faster than the splay tree, despite performing around thirty times fewer rotations, on average. This was believed to be due to the high costs of generating random numbers [Williams et al., 2001]. The red-black tree was slower for skew data than the splay tree, with only a slight improvement when little or no skew was present. Thus, while the standard BSTs worst case might prohibit its use in practice, in the average case it provides a reasonable benchmark for efficiency [Heinz and Zobel, 2002].

A heuristic splay tree [Williams et al., 2001], which splays a node after three successful searches, for example, was found to be more efficient than normal splaying, but at best was only about 3% faster than a standard BST. Splaying has been applied in areas where amortized performance is important, such as sorting [Moffat et al., 1996], index construction [Witten et al., 1999], and compression [Grinberg et al., 1995]. However, as a result of its poor performance against a standard BST, it has generally seen little use in common string management tasks such as vocabulary accumulation and string dictionaries. Nonetheless, in contrast to the binary search tree, it can avoid a terrible worst-case.

Clement et al. [2001] analytically and empirically compared the three primary representations of a trie — array, list, and ternary search trie (TST) — which were found to have logarithmic(expected) access costs but with different constants. The array trie requires no character comparisons and is thus the fastest, but requires the most space. The ternary search trie required less space than the array trie, but at a cost in access speed. The list trie was the most space-efficient trie but was also the slowest to access, requiring almost three times the comparisons of a ternary search trie. The TST can therefore offer the best time-space tradeoff of the types of trie [Clement et al., 1998]. The Patricia trie has been shown to reduce the space required by a compact trie by around 30%, but can remain larger than a ternary search trie [Sedgewick, 1998; Heinz et al., 2002].

Heinz et al. [2002] experimentally compared the performance of the burst trie against several data structures: a compact trie, a TST, a standard BST, a splay tree, and the standard hash table — the chaining hash table with move-to-front chains and the bitwise hash function — for the task of vocabulary accumulation. The burst trie was consistently faster and more compact than the trees and tries. It was faster than a compact trie while requiring about a sixth of the space; more compact than the BSTs while being over two times faster; and could approach the efficiency of hash tables while maintaining sorted access to strings (that is, containers can be accessed in sort order). The results were consistent with the theoretical analysis of similar data structures [Knessl and Szpankowski, 2000; Ramesh et al., 1989; Szpankowski, 1991] that expect logarithmic access bounds.

Zobel et al. [2001] compared the performance of several data structures — including the BST and splay tree (with heuristic splaying) — for the task of accumulating the vocabulary of a large

Data Structure	Space overhead in bytes
Array	8
Linked list	$24s$
Binary ST	$28s$
Random ST	$32s$
Red-black	$32s$
Splay	$32s$
AVL	$32s$
TST	$21s$
Array trie	$520t$
Burst trie	$24\bar{s} + 520t + 13b$
Chained hash	$4m + 24s$

Table 2.1: The space overhead (in bytes) of well-known data structures, relative to the number (s) *of strings stored, the number* ($\bar{s}$) *of strings stored in containers, the number* (m) *of slots used by the hash table, the number* (t) *of tries, and the number* (b) *of containers used by the burst trie.*

text collection in memory, and found the standard hash table to be the fastest and most compact data structure available for maintaining fast access to variable-length strings. For strings with a skew distribution, using move-to-front in the individual chains allows the load average to 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.

The current state-of-the-art of data structures for strings is, therefore, the chaining hash table with move-to-front chains using the bitwise hash function, the burst trie with move-to-front chains, and also the binary search tree — being a standard choice of sorted data structure due to its simplicity of implementation and good average case performance. The burst trie is considered to be the best choice when sorted access is required on large sets of strings [Heinz et al., 2002]. All three, however, use linked lists as a substructure, which can lead to poor use of cache, as we discuss later.

2.5.1 Space requirements

Data structures consume memory in several ways: space allocated for the strings and for pointers; space allocated for structural components such as slots, tries, and nodes; and overhead due to compiler-generated structures, operating system overheads, and space fragmentation. Typically 8 bytes of space is reserved by the operating system on every memory allocation call or *malloc* [LaMarca and Ladner, 1996; Silberschatz et al., 2004]. (For brevity, we assume that no custom memory allocators are used). These additional bytes are administrated by the operating system for housekeeping purposes and are not accessible by the programmer. Nonetheless, it is important

not to overlook this space overhead, for as we show below, the amount of memory reserved by the operating system can become excessive.

In a linked list, each string requires two pointers and (in a typical implementation) two *malloc* system calls [Esakov and Weiss, 1989; Horton, 2006]. Nodes can also store additional information, such as a counter or a file pointer, but we assume no additional parameters. The space overhead for a linked list is therefore $24s$, where s is the number of strings inserted. A fixed-sized or dynamic array has no space overhead, apart from the initial call to *malloc*.

In a chaining hash table a further 4 bytes is required per slot, resulting in a space overhead of $4m + 24s$ bytes, where m is the number of slots used. For the standard BST, each string requires three pointers and two *malloc* system calls. The space overhead is therefore $28s$ bytes. In AVL trees and red-black trees, every node must store a bit to indicate its weight or color, respectively. However, to store a single bit in memory, compilers will typically allocate an entire word (4 bytes) in order to maintain word-alignment. Alternatively, with some programming effort, each node can be represented as a 13 byte array; the left-child, right-child, and string pointers can be stored in the first 12 bytes, with the remaining byte used to store the color or weight bit, giving a total space overhead of $29s$ bytes. In our case, however, we assume that an entire word is allocated, resulting in a space overhead of $32s$ bytes for both the AVL and red-black trees. In a randomized binary search tree, we assume that the weight of a node is stored as a 4-byte integer, resulting in an overhead of $32s$ bytes. Similarly, in a typical implementation, every node in a splay tree stores a 4-byte pointer to its parent to simplify the implementation of splaying [Sedgewick, 1998].

In an array trie, strings are represented as a chain of trie nodes. We assume that each trie node has 128 4-byte pointers, mapping to the 128 characters of the ASCII table. The space overhead of an array trie is therefore $520t$ (including the call to *malloc*), where t is the number of trie nodes. For a ternary search trie, nodes store a single character and three pointers. The space overhead is therefore reduced to $21t$ bytes. For the burst trie, assume that b is the number of containers. In a typical implementation, each container stores a 4-byte pointer to the start of its linked list and reserves a byte for housekeeping; this is used to maintain the end-of-string flag (also known as the string-exhaust flag), which indicates whether the container has consumed a string. We discuss these implementation details in the next chapter. The space overhead of the burst trie is $24\bar{s} + 520t + 13b$ bytes, where $\bar{s}$ represents the number of strings stored in containers. We summarize these space overheads in Table 2.1.

2.6 The memory hierarchy

In the mid-1960s, Moore observed that the number of transistors that would be incorporated on a silicon die should double every 18 months for the next several years [Intel, 2007]. Over the past three and a half decades, the computing power and complexity of CPUs (which corresponds roughly the number of transistors) has grown in close relation, which is known as Moore's law [Hennessy

Year introduced	Capacity	$ per MB
1980	64 Kbit	1500
1983	256 Kbit	500
1985	1 Mbit	200
1989	4 Mbit	50
1992	16 Mbit	15
1996	64 Mbit	10
1998	128 Mbit	4
2000	256 Mbit	1
2002	512 Mbit	0.25
2004	1024 Mbit	0.10

Table 2.2: DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter doubling approximately every two years [Patterson and Hennessy, 2005].

and Patterson, 2003]. Processing speeds improved at a rate of about 35% per year until 1986, at 55% per year until 2003, and then slowing thereafter [Patterson and Hennessy, 2005]. In contrast, manufacturers of main memory technologies pursued an increase in memory cell density rather than access speed. Although memory access speeds is indeed improving, it is at a much slower rate than the speed of processors, at around 7% a year [Patterson et al., 1997; Patterson and Hennessy, 2005].

As a consequence, since the mid 1980s there has been a growing performance disparity between the processor and main memory, which effectively throttles the performance of processors. The time required to access data from main memory — memory access latency — has rapidly become the dominant factor of performance, and the effects on software were noted in the early 1990s by several researchers [Boland and Dollas, 1994]. Wulf and McKee [1995] stated:

> "We all know that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed — each is improving exponentially, but the exponent for microprocessors is substantially larger than that for DRAMs. The difference between diverging exponentials also grows exponentially; so although the disparity between the processor and memory speed is already an issue, downstream someplace it will be a much bigger one".

In order to take advantage of faster and more powerful processors, the cost of accessing memory must be substantially reduced. In virtually every computer sold since 1975, the main memory uses dynamic random access memory (DRAM) technology [Hennessy and Patterson, 2003; Dennard, 1968]. DRAM is popular because it is cheap to manufacture. One DRAM cell (a single bit) requires only a single transistor and capacitor. Although cheap to buy, DRAM is expensive to

access. Furthermore, because a DRAM cell uses a capacitor, a read can disturb its information. To prevent loss of information, DRAM cells must be periodically refreshed which forces main memory to be temporarily unavailable [Hennessy and Patterson, 2003; Patterson and Hennessy, 2005]. In 2004, the typical access time for DRAM memory was 50 to 70 nanoseconds [Patterson and Hennessy, 2005].

There are two possible solutions to substantially reduce the access cost of main memory. The first is to use static RAM (SRAM). A memory cell (a single bit) in SRAM consists of six transistors, which, although more complex, offers rapid access for both read and writes. Additionally, static RAM cells do not lose their information when read. The typical access time for SRAM memory is about 0.5 to 5 nanoseconds [Patterson and Hennessy, 2005], so if used for main memory, it can dramatically reduce (but not eliminate) access latency. However, there are several reasons why SRAM is not an economically viable solution [Anderson and Shanley, 1995]: SRAMs are typically ten times more expensive than DRAM memory; SRAMs are physically larger than DRAMs, requiring more silicon for the same amount of bits; SRAM cells typically operate at a much higher frequency, and, with six times as many transistors as DRAM, they will generate more heat and thus require sufficient cooling.

An alternative solution is to use a cache, forming a memory access hierarchy between the processor and main memory. The need for a memory hierarchy was understood as far back as 1946 by Burks, Goldstine, and Neumann, in a preliminary discussion of the logical design of an electronic computing instrument:

> "Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding, but which is less quickly accessible" [Patterson and Hennessy, 2005].

We rely on a range of sources, but in particular Patterson and Hennessy [2005], in the discussions that follow.

2.6.1 Caches

A *cache* is a small piece of SRAM that is physically located between the CPU and main memory [Handy, 1998]. Its goal is to keep the *working set* of a program (frequently accessed data) close to the CPU, to give the illusion of accessing fast memory, when in fact, main memory uses DRAM [Denning, 1968; Denning and Schwartz, 1972]. A cache works because programs tend to have the property of *locality of reference*, which states that recently accessed items are more likely to be accessed again in the near future [Denning and Schwartz, 1972]. The usefulness of cache is therefore directly tied to how often a program requests the same data or code. If a program was intentionally designed to never access data more than once and issued frequent random memory

accesses, then a cache would provide no improvement in performance [Anderson and Shanley, 1995]. The principle of locality is comprised of two components: *temporal* and *spatial*. A program can improve its temporal locality by increasing the rate at which it re-uses recently accessed data; and spatial locality can be improved by accessing data that is near previously accessed locations [Madison and Batson, 1976]. Cache behavior is a critical factor in our techniques described in later chapters, so we describe it in detail here.

The first type of cache used in mainstream computers was external, located outside the processor but before main memory. Many manufacturers of 386-based PCs included an external cache [Anderson and Shanley, 1995], consisting of a relatively small amount of SRAM coupled with a cache controller. The cache controller copied frequently accessed data and instructions from main memory into its cache, to reduce access time for future requests.

Being external meant that, on every access, the CPU was forced to consult the system bus. When the CPU requests data or instructions from main memory, it has to contend for ownership of the system bus, issue a bus cycle to communicate with main memory, then wait until main memory responds and sends back the data. Along with the time needed for memory to respond, the speed of the bus (which is typically a factor of 10 less than that of the processor), and its bandwidth (the number of bits that can be transferred per cycle) can greatly affect performance [Hennessy and Patterson, 2003]. An external cache can therefore only offer limited performance gains, reducing (but not eliminating) the stall time required for frequently accessed items.

To yield substantial improvements in performance, the cache must be moved *on-die*. By moving the cache on board the processor, a memory request that is found in cache can be immediately forwarded to the processor, greatly reducing both costly CPU stalls and system bus contention. The 486 was the first Intel processor to use an on-die cache, which became known as the first-layer (L1) of the memory hierarchy [Intel, 2007].

Caches are bound by fundamental laws that govern their performance. No signal can propagate faster than light, so, given a desired access latency, only a finite amount of data is reachable. Therefore, the larger the cache, the more surface-area it needs (requiring longer connections) and, thus, the slower it must be [Handy, 1998]. As a consequence, growing the size of the single on-die L1 cache will not suffice, as it will lead to an increase in access latency. Therefore, current processors implement a hierarchy of *on-die* caches, to permit smaller but faster caches to reside closer to the CPU core, while slower but larger caches are placed further away [Hinton et al., 2001]. Processors typically implement two layers of cache, but with the increasing performance-gap between processor and main memory, the depth of the cache hierarchy is likely to increase [Sanders, 2003]. On-die third level caches have been integrated on Intel Itanium 2 processors and some Pentium 4 chips [Intel, 2007].

The first layer (L1) is located near the CPU core. It typically operates at the same speed as the processor and has an access latency of 1 to 3 CPU cycles, but is usually no bigger than a few kilobytes. The second layer (L2) is much larger (typically between 512 KB and 8 MB) and has an

access latency of around ten cycles. Communication between the caches and the processor is via a private high-speed bus. A third-level cache (L3) can have a capacity in the tens of megabytes, and thus require several tens of clock cycles to access, which is still over a magnitude faster than main memory, which typically requires around 700 or more clock cycles per access [Meyer et al., 2003].

Caches are organized into equal-sized blocks known as cache-lines. Cache-line sizes typically vary between L1 to L3 cache and are often between 64 and 256 bytes long [Intel, 2007]. A *cache-lookup* is the process of searching a cache to determine whether a cache-hit or cache-miss occurs. A *cache-hit* occurs when the requested data is found in some cache-line. Otherwise, its a *cache-miss* and the required bytes must be fetched from the next level of the memory hierarchy.

The cache-line is the smallest unit of access in a cache. When a line of information is copied from main memory during a cache-miss — a process also known as a cache-line fill — the transfer begins at an address boundary that is evenly divisible by the cache line size [Intel, 2007]. The advantage of accessing a block of memory, rather than a single byte or word (4 bytes), is that it greatly improves spatial locality and permits better utilization of memory and system bus bandwidth. Dynamic RAMs have therefore been optimized for block access to reduce the cost of cache-line fills. RAMBUS RDRAM for example is a variant of DRAM that permits up to 16 bytes to be accessed in only twice the time required for a single byte [Sanders, 2003].

Large cache-lines can lower the overall cache-miss rate, but, if their size is a significant fraction of the overall cache size, the number of lines available in cache can be too small. This will lead to a great deal of competition for acquiring cache-lines, resulting in cache *thrashing*, where frequently accessed lines are evicted from cache before all their words are accessed. This will also increase cache *pollution*, as cache-lines will most likely contain larger portions of infrequently accessed data. A more serious problem associated with larger cache-lines is that the cost of a cache-miss increases.

To access a cache, a mechanism must be in place to determine whether or not the required data is available, and how to retrieve or update it. The simplest approach is to map a word of memory — using its address — to a single cache-line. This is called a *direct-mapped* cache. Almost all direct-mapped caches use the mapping:

$$\text{Cache-line number} = (\text{Address}) \text{ modulo } (\text{Number of cache-lines})$$

Figure 2.11 illustrates a simple direct-mapped cache with eight 64-byte cache-lines that are associated with tags and valid bits. These are used to determine whether the cache-line selected is a cache-hit or miss.

A direct-mapped cache is cost effective to implement, but because it allows many different locations in memory to be mapped to a single cache-line, the cache-miss rate can be high. It is more appealing to design a cache that offers more flexible placement of data in cache-lines, to reduce the cache-miss rate. There are two well-known schemes. The first, known as a *fully*

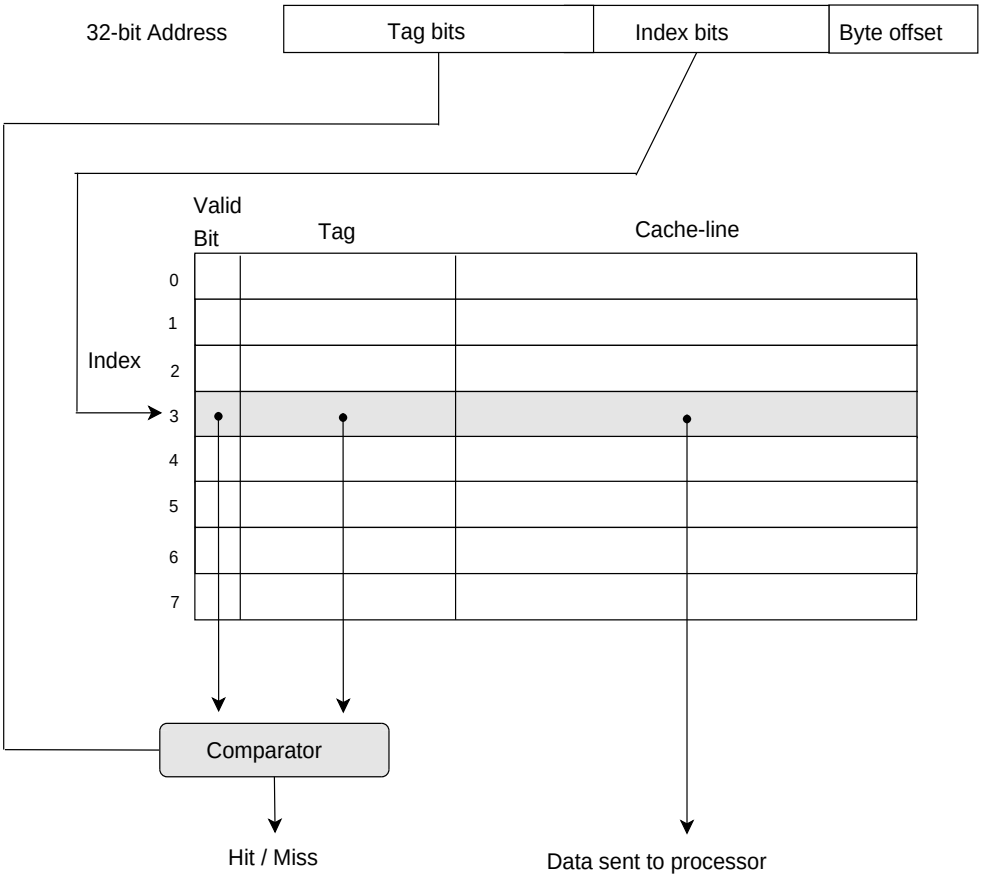


Figure 2.11: Cache design. A 32-bit address is divided into three portions: tag, index and offset. The index is used to select the required cache-line. The tag is used to check whether the cache-line contains data from the required address. The block offset is the location of the data within the cache-line. The valid bit is used to confirm that the cache-line holds data that is mapped to the tagged address. A cache-hit occurs when the selected cache-line has a matching tag and the valid bit is set. Otherwise it is a cache-miss. The result of the cache look-up (miss or hit) is sent to the processor, along with the data (which is ignored on a miss).

associative cache, allows a block of data to be placed in any cache-line. This approach can greatly improve the utilization and thus, the temporal locality of cache-lines. However, a cache-lookup is considerably more complicated: a comparator must be associated with every cache-line, which is used to compare the tag and valid fields in order to determine whether its a cache-miss or hit (Figure 2.11). A fully associative cache is therefore expensive to manufacture. A more serious problem is that it is also more expensive to search. Fully associative caches are primarily used in situations where the cost of a cache-miss is very high and must be minimized (for example, where it might involve accessing an external memory device).

To reduce the cache-miss rate while minimizing the cost of a cache-lookup, a *set associative* cache can be used. An n-way set-associative cache groups cache-lines into sets, each of which contains n cache-lines. A block of data from memory is mapped to a set, where it can then be placed in any cache-line that is a member of the set. For example, almost all set-associative caches use the mapping:

$$\text{Set number} = (\text{Address}) \text{ modulo } (\text{Number of sets})$$

The advantage of a set associative cache is that it limits the number of comparators required. This results in faster cache-hits — relative to a fully associative cache — while offering some reduction cache misses. For example, in a 4-way set associative cache, a set is selected and only four cache-lines are searched in parallel (that is, only four comparators are required). Increasing the degree of associativity or n usually decreases the cache-miss rate, but at the expense of increasing the cache-lookup time. As a result, levels of the memory hierarchy are implemented with varying degrees of associativity. L1 cache is usually of low associativity — typically between 2- and 4-way — as a fast cache-lookup is more important than the cost of a cache-miss, which is small. L2 cache, however, is typically implemented with higher associatively — between 8- to 16-way, due to the high costs of accessing main memory.

When a cache-miss occurs in a set associative or fully associative cache, a cache-line must be chosen to be replaced. To choose a cache-line, the most commonly used scheme is *least recently used*, which replaces the cache-line that has been unused for the longest time. LRU replacement schemes, however, become more complicated to implement as associativity increases. Hence, current processors typically employ an approximate LRU scheme or evict cache-lines at random [Intel, 2007].

Hill and Smith [1989] classify cache misses as either compulsory misses, capacity misses, or conflict misses. A *compulsory* miss occurs when data is accessed for the first time. A *capacity* miss occurs on access to a cache-line that was evicted due to a full cache. A *conflict* miss occurs when the number of frequently accessed pages that map to a cache set exceed the associativity of the cache.

On a cache-miss involving a read, the processor stalls until the required information is transferred to it. When data is written to cache, however, extra steps are needed to ensure the caches

and main memory remain consistent. The simplest way to keep main memory and cache consistent is to always write or update data in both before continuing, known as a *write-through* cache. Write-through caches are simple to implement, but can lead to frequent CPU stalls on write-intensive programs. The alternative to a write-through cache is called a *write-back* cache. With this scheme, cache-lines are updated and are only forwarded to main memory when they are evicted from cache.

2.6.2 Virtual memory and the TLB

Since the 386, Intel processors have been designed to address all memory locations supported by the underlying hardware, even though the actual number of physical locations (the amount of main memory) was limited [Hennessy and Patterson, 2003]. As a result, whenever the processor requests data from main memory (due to a cache-miss), the location or address it sends out is *virtual*. It must first be translated into a *physical* address through a combination of hardware and software. This concept is known as virtual memory, which effectively uses main memory as a form of *cache* to secondary storage; which is usually a hard disk.

In a virtual memory system, main memory is divided into a set of equal-sized blocks called *pages*. The physical location of a page can either be in memory or on disk, and is determined by searching an index called a *page table*. The page table is maintained by the operating system and resides in memory. When a page is not found in main memory, a *page fault* occurs. This involves accessing the hard disk to fetch the required page, which can take millions of clock cycles.

The major disadvantage of virtual memory is that every memory reference issued requires at least two memory accesses: the first to access the page table and the second to access main memory (the physical location of the page table is known by the CPU). Having to translate virtual addresses on every request can be expensive. To make the address translation faster, a special on-die cache is used, called a translation-lookaside buffer (TLB). The TLB stores a subset of entries in the page-table to permit rapid translations of frequently accessed pages. Accordingly, a *TLB-hit* occurs when a virtual page address is found in the TLB. Otherwise, it is a *TLB-miss*, which involves accessing the page-table.

The TLB however, is of limited size, usually between 16 and 128 entries, and is usually fully-associative; that is, it is designed to minimize the number of TLB-misses. Unlike a page-table, however, the TLB only holds translations for pages that currently reside in memory. In addition, an L1, L2, and L3 cache-miss, a TLB-miss, and a page-fault can occur independently from one another when a virtual address is translated. In the best case, a virtual address is translated by the TLB and sent to the cache where the appropriate data is found. In the worst case, a virtual address can miss in all three components: the TLB, the page table, and the cache [Rahman, 2003]. Table 2.3 summarizes the possible combinations of events.

The size of a page in memory can also influence the number of TLB misses incurred. A large page size can reduce the number of TLB misses, as a single TLB entry can hold more entries from

TLB	Page table	Cache	Possible?	Description
hit	hit	miss	yes	Page table is never checked if TLB hits.
miss	hit	hit	yes	A TLB-miss causing a page-hit and the data is in cache.
miss	hit	miss	yes	A TLB-miss causing a page-hit but the data is not in cache.
miss	miss	miss	yes	A TLB-miss causes a page-fault and the data is not in cache.
hit	miss	miss	no	Impossible: if a page is not in-memory, it cannot be in TLB.
hit	miss	hit	no	Impossible: if a page is not in-memory, it cannot be in cache.
miss	miss	hit	no	Impossible: if a page is not in-memory, it cannot be in cache.

Table 2.3: The possible combinations of events in the virtual memory system [Patterson and Hennessy, 2005].

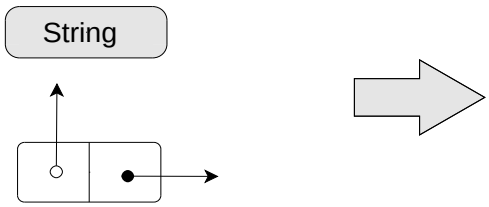
the page table. However, the use of a large page size will also make a TLB miss more expensive, particularly when a page-fault is involved. Hence, to achieve a good balance between TLB misses and their cost, pages on current machines are often set between 4 to 8 KB in size [Silberschatz et al., 2004], with 64 KB being an example of a large page size that is often found in mainframe servers.

2.7 Cache inefficiency in string data structures

Chains are simple structures, but are inefficient with cache. Nodes and their contents are typically scattered across main memory, and with an unpredictable input sequence, neither temporal nor spatial locality are high. Figure 2.12 illustrates the basic form of a node along with its common implementation in C [Esakov and Weiss, 1989], of which, is used widely in programs and user libraries [Musser and Saini, 1995].

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 at least two pointers per stored string. Access to a node can therefore result in two cache misses (that is, two random accesses to main memory): one to access the node and another to access its string. Hence, the primary cause of cache inefficiency in string data structures is the use of pointers. The reason why chains are so expensive is because they enforce pointer de-references — known as pointer chasing — where the address of the next node cannot be known until it is requested [VanderWiel and Lilja, 2000]. Coupled with scattered allocation, traversing a chain of nodes can result in excessive cache misses. Mowry [1995] showed the impact of memory latency on scientific programs, which spent more than half of the time waiting for data.

As the cost of accessing memory continues to increase relative to the speed of the CPU [Patterson et al., 1997], arithmetic operations alone are no longer adequate for describing the compu-




```
typedef struct node
{
  char  * string;
  struct node  * next_node;
}
node;
```

Figure 2.12: A graphical representation of a standard node used by most string data structures, along with an example of its representation in C.

tational cost of a data structure [Kowarschik and Weiß, 2003]. For programs to remain efficient in practice, measures must be taken to exploit cache; however, cache is not under the control of the programmer [Rahman, 2003; Hinton et al., 2001]. Nonetheless, programs can be designed to make good use of cache by improving the regularity and predictability of memory accesses [Hennessy and Patterson, 2003], in order to reduce or eliminate pointer chasing and random accesses to memory.

2.8 Current techniques for exploiting cache

To make good use of cache in current processors, programs must improve their locality of access, that is, minimize random accesses to main memory by using more predictable and regular memory access patterns. The locality of access can be improved by increasing the rate at which a program re-uses recently accessed data (*temporal* locality), and by accessing data that is near by (*spatial* locality), thus utilizing cache-lines and hardware prefetch (described below) to reduce cache misses and consequent CPU stalls [Lebeck, 1999]. The importance of minimizing TLB-misses has also been noted [Agarwal, 1996; Moret and Shapiro, 1994; Romer et al., 1995; Rahman et al., 2001], which is achieved through improved temporal locality and by reducing the amount of memory used by a program.

Code transformations and optimizations at compile-time are the simplest means for improving access locality. These involve changing the order in which instructions in a program are executed, in particular, the iterations of a program loop, to improve cache-line utilization and to apply instruction scheduling to optimize the execution of instructions [Kerns and Eggers, 1993]. Some common techniques include loop unrolling, loop skewing and loop fusion. Detailed descriptions of these techniques can be found in compiler-based literature [Beyls and D'Hollander, 2006; Allen and Kennedy, 2001; Muchnick, 1997; Leung and Zahorjan, 1995; Manjikian and Abdelrahman, 1995; Bacon et al., 1994; Carr et al., 1994]. However these compile-time optimizations are only effective at reducing capacity misses and instruction costs. They cannot address compulsory misses [Kowarschik and Weiß, 2003] and are generally no better at reducing conflict misses [Rivera and Tseng, 1998].

Calder et al. [1998] introduced a compile-time data placement technique that relocates stack and global variables, as well as heap objects to improve the data access locality of programs. Although effective for stack and global variables, there was little improvement for heap objects. In addition, a training run (a program profile) is required to determine how to relocate variables to achieve good performance. This has obvious limitations on dynamic data structures that rely on heap-allocated storage, and which have access patterns that are typically not known in advance. Ding and Kennedy [1999] proposed a similar technique for scientific simulation applications such as structural mechanics, where data is re-organized at run-time with the assistance of the compiler. However, compile-time techniques can do little to improve the cache performance of dynamic pointer-intensive data structures, leaving much of the cache optimization effort to the programmer [Granston and Wijshoff, 1993; Loshin, 1998; Burger et al., 1996].

Data prefetching is a common technique used to reduce compulsory misses and improve spatial locality by fetching data into cache before it is needed [VanderWiel and Lilja, 2000]. Hardware prefetchers such as stride prefetching [Smith, 1982], work independently from the processor and require no intervention from the programmer. They typically intercept memory requests and use simple arithmetic (and often a pool of recent addresses) to predict future data accesses [Kowarschik and Weiß, 2003; Intel, 2007].

Although hardware prefetchers are effective at reducing compulsory misses, random memory accesses are impossible to predict. Hence, for pointer-intensive data structures such as the standard hash table, a hardware prefetcher can pollute the cache with unwanted data. More sophisticated prefetchers, such as the Markov predictor [Joseph and Grunwald, 1997] or dependence-based prefetching [Roth et al., 1998], can improve prefetch precision but are often more complex and expensive [Baer and Chen, 1991; 1995; Fu et al., 1992].

Hardware prefetchers work well with programs that exhibit regular or stride access patterns; arrays are prime candidates. Unfortunately, most data structures for strings use linked lists as substructures. As discussed in Section 2.7, traversing a linked list can lead to inefficient use of cache, because nodes and their strings can be scattered in main memory resulting in poor access locality. Furthermore, nodes are accessed via pointers which can hinder the effectiveness of hardware prefetch, due to pointer chasing [VanderWiel and Lilja, 2000].

For such situations, software prefetchers [Callahan et al., 1991; Karlsson et al., 2000] have been proposed, the simplest being the use of *prefetch* instructions such as greedy prefetching [Luk and Mowry, 1996]. These prefetch instructions are manually inserted into code by programmers or is added by compilers [Mowry, 1995; Lipasti et al., 1995]. In contrast to hardware prefetchers, software prefetchers have some CPU cost and often require intervention from the programmer. Roth and Sohi [1999] introduced jump-pointers, a framework of pointers that connect non-adjacent nodes in a linked list, to be used by a software prefetcher to overcome pointer-chasing. However to be effective, an appropriate jump-point interval must be found which is non-trivial; Roth and Sohi do not describe how to adapt this interval to an application. Consequentially, once installed,

jump-pointers do not adapt to changes in access patterns, which can lead to inefficiency [Yang et al., 2004]. In addition, jump pointers cannot prefetch the sequence of nodes between jump intervals.

Karlsson et al. [2000] extend jump pointers by introducing a prefetch array. Prefetch arrays consists of a number of jump pointers to nodes that are located consecutively in memory. These pointers can be used to prefetch several nodes at once — by using hardware or software prefetchers. Prefetch arrays can also be used to prefetch nodes in-between jump pointers, and can be effective in applications where the traversal path is not known in advance. Prefetch arrays, however, require more space than jump pointers, and remain ineffective for dynamic pointer-intensive data structures, such as trees with high branching factors.

Speculative execution uses a separate processor (or threads) to run ahead of the main program during a cache-miss, to de-reference pointers and cache data before it is requested [Dundas and Mudge, 1997; Luk, 2001]. A pointer cache [Collins et al., 2002] can be useful here, as it stores frequent pointer transitions. However, overall effectiveness is poor when applied to dynamic pointer-intensive data structures, as, to be effective, knowledge of access patterns is needed, and there must be sufficient delays between cache misses and node processing.

Stoutchinin et al. [2001] observed that there tends to be a regularity in the allocation of nodes in a linked list, and thereby proposed a compiler-based software prefetch method, called speculative induction pointer prefetching, which modifies a compiler to detect a linked list scanning loop. Once detected, the compiler assumes that nodes accessed at run-time will generate regular memory reference patterns. The compiler can then insert code that computes the stride offset for induction pointers — pointers that are updated in each loop iteration, such as the next-node pointer. Once the induction pointers are identified, data whose addresses are computed relative to the induction pointers are speculatively prefetched. This technique works well on programs that exhibit stride access patterns, but is not particularly effective on programs with irregular memory access patterns.

There are also proposals for combining hardware and software prefetchers that attempt to overcome pointer chasing in pointer-intensive applications. Yang et al. [2004] for example, proposed programmable hardware prefetch engines, with similar techniques by Hughes and Adve [2005] for multi-processor systems. Guided region prefetching is another example where the compiler generates prefetch hints that guide hardware prefetchers [Wang et al., 2003].

The techniques discussed above are concerned with the consequences of cache misses, as opposed to the cause, which is poor access locality. Chilimbi et al. [1999b] showed that careful layout and relocation of nodes in pointer-intensive data structures can improve access locality without changing programs semantics. Two cache-conscious relocation techniques were introduced: *clustering* and *coloring*. Clustering attempts to pack nodes that are likely to be accessed contemporaneously into blocks of memory that are sized to match the cache-line. Hence, clustering is a cache-conscious heap allocator similar to *malloc*, expect that it requires the programmer to supply

a pointer to an existing node that is expected to access the new node, such as a parent node. An incorrect choice of pointer will not affect program correctness, but can affect performance.

Coloring maps contemporaneously accessed nodes to non-conflicting regions of cache, that is, it segregates nodes based on access frequency obtained from program profiles. In some cases, such as with a binary search tree, access patterns can be derived from structural topology (that is, nodes near the root will be accessed more frequently than those near the leaves). Coloring however, is only compatible with static tree-like data structures and incorrect usage can affect program correctness [Chilimbi, 1999].

Clustering and coloring are currently only compatible with homogeneous objects, such as the fixed-sized nodes of a binary search tree; they do not support allocation or relocation of variable-length objects, such as strings. In addition, these techniques assume that the size of a node in bytes, is less than half the cache-line size. In cases where nodes are larger, their internal fields can be reorganized by separating frequently accessed fields into smaller segments that can fit into a cache-line. However, this technique has limited application as it can affect program correctness [Chilimbi, 1999; Chilimbi et al., 2000].

Another important characteristic of clustering is that nodes are grouped together to improve spatial access locality without eliminating their pointers. Hence, a clustered pointer-intensive data structure can still make poor use of hardware prefetch, due to pointer chasing. Chilimbi [1999] suggests eliminating the next-node pointers of a linked list and accessing nodes using arithmetic offsets, forming an *implicit* clustered chain. However, this technique assumes homogeneous nodes and involves non-trivial programmer intervention, and was thus not pursued for coloring and clustering.

Chilimbi et al. [1999a] note the applicability of their methods to chained hashing, but not with move-to-front on access. This is likely to be a limiting factor, as move-to-front is itself an effective cost-adaptive reordering scheme. Nor is it clear that such methods will be of benefit in the environments that we are concerned with, where the volume of data being managed may be hundreds of times larger than cache. Furthermore, there is currently no implementation support for clustering or coloring on current platforms. Nonetheless, implementing a clustered linked list is straightforward: the nodes of a linked list are simply allocated contiguously in memory, in order of occurrence. A clustered list is likely to make better use of cache during search, but updating a single array of nodes (that maintain next-node pointers) can be expensive. Hence, Chilimbi [1999] suggests grouping nodes into fixed-sized memory blocks that are sized to match the cache-line. This forms a chain of blocks that permit efficient update, but at some cost in space and cache efficiency (the individual blocks can be scattered in memory). We revisit these implementation issues in more detail in the next chapter, where we describe how to implement a clustered hash table, burst trie, and BST for strings.

Virtual cache lines, proposed by Rubin et al. [1999], is a similar technique to clustering. In this approach, the nodes of a linked list are stored within fixed-sized blocks of memory that

match the size of the cache-line. Access to a block will prefetch the next set of nodes in the list, which improves spatial access locality. However, the virtual cache line technique is currently only compatible for linear linked lists. Its effectiveness on tree-based data structures such as the BST, is unknown. In addition, nodes are assumed to be of fixed-size and no pointers are eliminated; hence variable-sized objects, such as a strings, remain randomly allocated in memory, which is inefficient. Another similar technique to clustering is the use of hash buckets to improve the cache-efficiency of the hash join algorithm in SQL [Graefe et al., 1998]. In this approach, the two chaining hash tables that are used to perform the join are constructed with nodes that are sized to match the cache-line. Hence, initial access to a node in a linked list will prefetch an entire cache-line of fixed-length keys (integers), which is more efficient than accessing a single key per node.

Truong et al. [1998] introduced a field reorganization and interleaving technique that groups the homogeneous nodes of a linked list into fixed-sized memory blocks called *arenas*, using a dynamic cache-conscious heap allocator called *ialloc*. This approach clusters the fields of nodes according to their expected frequency of access, which is determined by the programmer. Hence, a cache-line will store frequently accessed components from a number of nodes, which can improve access locality. However *ialloc* does not support variable-sized fields, such as arrays of integers or a string; such fields must remain randomly allocated in memory. In addition, pointer chasing remains an issue and the process of interleaving nodes can affect program correctness. Kistler and Franz [2000] proposed a similar field relocation technique for Java applications, which relocates data items in real-time by collecting access patterns while the program is being run.

Panda et al. [2001] proposed an array splitting technique that can transform the memory layout of aggregate data structures — such as an array of `structs` in C — to reduce cache pollution caused by the prefetch of infrequently accessed data fields. However, this technique is only intended for embedded memory systems which can address the issue of program correctness, by making the entire program visible to the compiler. Alternative data relocation techniques include identifying frequently repeated sequences of consecutive data references in programs, and clustering them accordingly to improve the use of cache [Chilimbi and Shaham, 2006]. Similar techniques involve the use of memory pools that cluster objects of the same data type to improve access locality [Lattner and Adve, 2005; Zhao et al., 2005].

Berger et al. [2002] tested eight high-performance custom memory allocators and found that six were no better and often led to worse cache utilization in pointer-intensive programs, than a high-performance general-purpose allocator. The remaining two, which involved the use of memory pools, led to slight improvements in access times, but at some cost in space. Open-address hash tables have also been investigated in the context of cache [Heileman and Luo, 2005], but as noted previously, open-address hash tables are not efficient or practical solutions for managing strings.

A study by Badawy et al. [2004; 2001] tested the effectiveness of combining software prefetch on data that was cache-consciously allocated, and found software prefetching to be of little value compared to the performance gained through careful layout. The authors concluded that careful

layout of data through clustering often outperformed software prefetching (such as jump pointers), particularly on machines with limited memory bandwidth. Hence, allocating or relocating data in a cache-conscious manner has been shown to be the best current way of exploiting cache in pointer-intensive programs [Hallberg et al., 2003].

A software cache can also be used in-place of a software prefetcher [Aggarwal, 2002], which caches recently stored (homogeneous) data items in pointer-intensive data structures, to reduce the access cost of frequently accessed data items. Data access locality can also be improved by a combination of split caches, victim caches, and stream buffers, as discussed by Naz et al. [2004]. In addition, LaMarca and Ladner [1996] studied the performance of an implicit heap, and showed how padding techniques that align fixed-sized heap elements to cache-lines, can yield high reductions in cache misses.

Acharya et al. [1999] proposed data layout schemes that addressed — in the context of cache — the unused pointers in sparse tries [Knuth, 1998]. The authors dynamically change the representation of trie nodes for very large alphabets, into either an associative array, a bounded-height B-tree, or a chained hash table, to form an *adaptive* trie. A trie node begins as a cache-line sized associative array. When full, the trie node is represented as a set of cache-line sized associative arrays that are organized as a B-tree [Comer, 1979a]. On further growth, the set is managed as a chaining hash table. The adaptive trie was compared against the the ternary search trie [Bentley and Sedgewick, 1997] for the task of searching a small set of strings, and was found to be faster than the TST, due to better use of cache. Crescenzi et al. [2003] also compared the adaptive trie against other string data structures, such as the Patricia trie, an open-address (linear probing) hash table, a BST, an AVL tree, and the TST, to determine their efficiency for searching small string datasets with and without skew. The adaptive trie was shown to operate efficiently with no skew in the data distribution, but was consistently slower than the TST, BST, and the open-address hash table with skew. In addition, the authors note that the adaptive trie can be more space-intensive than the TST.

Ghoting et al. [2006] studied the cache performance of the frequent-pattern mining algorithm — FPGrowth — on current processors. The FPGrowth algorithm uses an annotated prefix tree (or FP-tree) to compactly represent a transaction dataset [Han et al., 2000]. FPGrowth performs successive bottom-up traversals of the FP-tree, which involves accessing nodes that are not contiguously allocated in memory. As a consequence, these traversals can attract excessive cache misses. The authors proposed replacing the FP-tree with a static cache-conscious prefix tree, that stores the nodes contiguously in memory. First, a standard FP-tree is built and a single fixed-sized block of memory is allocated, sized to fit the entire tree. The FP-tree is then traversed in a depth-first order, copying its nodes sequentially into the block of memory. As a result, spatial access locality can be improved. A similar copying technique to eliminate string pointers for string sorting was also proposed by Sinha et al. [2006], who demonstrated that doing so can halve the cost of sorting a large sets of strings, due to improved spatial access locality.

Rao and Ross [1999] proposed a new indexing technique called a cache-sensitive search tree (or CSS-tree), which stores a directory structure on top of an existing sorted array of homogeneous keys, such as integers. The directory structure is a BST where nodes are sized to match the cache-line. The key advantage offered by a CSS-tree, compared to a binary search using a single sorted array, is that the binary search is localized within nodes that are contiguously allocated, which improves access locality. However, the CSS-tree is a static structure that is build upon an existing sorted array. It cannot handle updates efficiently and is not designed for variable-length strings. To support updates using fixed-length keys, the authors developed a variant called a cache-sensitive B-tree (CSB-tree) [Rao and Ross, 2000]. The CSB-tree places all the child nodes of a given node contiguously in memory, to eliminate child node pointers. Hence, a parent node need only retain the first pointer to its child node. The rest of its children can be found through arithmetic offsets, which improves spatial access locality.

Torp et al. [1998] proposed a similar cache-efficient B-tree, called a pointer-less insertion tree (PLI-tree), which eliminates all pointers in nodes. The PLI-tree allocates homogeneous keys in a specific order, such that child nodes can be found through arithmetic calculations. Although compact and fast, the data structure cannot handle random insertions or variable-length objects such as strings; all insertions must be in incremental order which is potentially useful for append-only operations, such as maintaining logs or time-stamped data, for example.

Oksanen [1995] studied the memory reference locality of a balanced BST and a in-memory B-tree, and presented a new general purpose memory model for assessing the cost of memory references of these two data structures. Oksanen also presented two methods for enhancing the access locality (and hence, the use of cache) of a balanced BST. The first method rebuilds a balanced BST so that nodes are located optimally in memory, but requires the tree to be static. The second method changed the structural representation of nodes in a B-tree to AVL trees — known as an AB-tree — which was shown to improve access locality while sustaining comparable memory overheads to a B-tree.

Cache-oblivious data structures are designed to perform well on all levels of the memory hierarchy (including disk) without prior knowledge of the size and characteristics of each level [Frigo et al., 1999; Kumar, 2003]. Brodal and Fagerberg [2006] for example, theoretically investigated a static cache-oblivious string dictionary. However, their study shows no actual performance measures against well-known data structures. Similarly, a dynamic cache-oblivious B-tree [Bender et al., 2000] has been described, but with no analysis of actual performance. The cache-oblivious dynamic dictionary [Bender et al., 2004] has been compared to a conventional B-tree, but on a simulated memory hierarchy. These studies assume a uniform distribution in data and operations, which is typically not observed in practice [Bender et al., 2002].

Recent studies have evaluated the practicality of these data structures [Ladner et al., 2002; Brodal et al., 2002; Arge et al., 2005a] and reported potentially superior performance compared to conventional data structures for fixed-length keys, but not when compared to those that have been

tuned to a specific memory hierarchy [Arge et al., 2005b; Rahman et al., 2001]. Bender et al. [2006] have also proposed, in theory, a cache-oblivious string B-tree that can handle unbounded-length strings, and a cache-oblivious streaming B-tree, designed to efficiently support random insertions of fixed-length keys and range search operations [Bender et al., 2007]. The data structures that we explore in this thesis are not cache-oblivious, as they have been designed specifically to exploit the cache between main memory and the CPU.

2.9 Summary

The chaining hash table with move-to-front chains and a bitwise hash function, which we call a *standard* hash table, has been shown to be fastest and most compact data structure for storing and retrieving strings, in memory. However, the standard hash table cannot provide efficient sorted access to strings and is therefore not suited for applications such as database management, where rapid sorted access to data can be required. In such cases, the chaining burst trie with move-to-front chains, which we call a *standard* burst trie, has been shown to be the most efficient form of trie. The *standard* binary search tree is another sorted data structure that is faster, on average, than most other tree structures, but with poor worst-case performance that often makes it an unattractive option in practice. Many of the structures discussed in this chapter, namely the TST, BST, splay tree, red-black tree, burst trie, and hash table, are experimentally evaluated in later chapters. The outcomes of these experiments are consistent with the earlier work discussed here. The adaptive trie and the Judy trie are also experimentally evaluated in later chapters.

The standard hash table, burst trie, and BST are fast data structures because they allow for a reduction in the number of instructions executed. However, they have been designed assuming a RAM model and are thus oblivious to the presence of caches. As a consequence, these data structures can attract high performance penalties on current cache-oriented processors. Existing techniques at improving the cache performance of data structures include software and hardware prefetching; custom memory allocators and memory pools; node relocation techniques such as clustering and coloring; speculative execution and use of pointer and software caches; programmable prefetch engines and compiler-based techniques that can change the placement of data to improve access locality.

However, current techniques do not address the use of pointers in the standard hash table, burst trie, and BST, which is the fundamental cause of their inefficient use of cache. Pointer elimination techniques are well-known, but assume homogeneous nodes and fixed-length keys, in order to replace pointers with arithmetic offsets. A linear representation of a linked list — where nodes are stored contiguously in a static array — is a well-known example. Although pointers have been known to cause inefficient use of cache, attention to their use in the standard hash table, burst trie, and BST, has attracted virtually no attention in literature; in particular, techniques that reduce or eliminate the use of pointers in these data structures without compromising support

for variable-length strings or efficient update.

In the next chapter, we reconsider the principles of designing a standard hash table, burst trie, and BST, in order to exploit cache through pointer elimination while retaining support for variable-length strings and without compromising dynamic characteristics.

Chapter 3

Redesigning data structures to exploit cache

To design a cache-conscious data structure, it is necessary to reduce or eliminate the use of pointers, to exploit cache by using more regular and predictable memory access patterns. In the previous chapter, we discussed well-known data structures that can efficiently store and retrieve strings in memory. We showed that the fastest and most compact data structures available are the chaining hash table, burst trie, and BST [Heinz et al., 2002; Williams et al., 2001; Zobel et al., 2001]. However, these three data structures are typically designed assuming a random access memory model; that is, all memory locations are accessible, and, have uniform access costs. Hence, these data structures are oblivious to cache, and as a consequence, are likely to attract excessive cache misses on current processors which will dominate their performance. The fundamental cause of their inefficiency is the use of pointers. Pointers cause two problems. First, the memory address they point to cannot be known until the pointer is dereferenced. As a result, hardware prefetchers are rendered ineffective. Second, they can cause random access to memory which results in poor use of cache.

However, effective techniques at eliminating pointers in dynamic string data structures have yet to be proposed. As discussed in the previous chapter, current techniques such as clustering [Chilimbi, 1999] can improve the cache-efficiency of linked-list data structures, by storing their homogeneous nodes contiguously in main memory, but without eliminating pointers.

In this chapter, we explore cache-conscious pointer elimination techniques — the *compact chain* and the *dynamic array* — and describe how to re-design the current best string data structures, with the aim of creating cache-conscious alternatives that can potentially yield substantial gains in performance, without compromising dynamic characteristics. We experimentally and theoretically analyze, in the context of cache, the expected gains in performance and the potential disadvantages

of pointer elimination. These techniques are applicable to string data structures in virtually any computing application. 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 is used. We expect these techniques to yield similar performance gains on 64-bit processors.

3.1 Clustered chain

In a typical implementation of a *standard* chained data structure, every node accessed incurs at least two pointer traversals: one to reach the node and the other to reach its string [Esakov and Weiss, 1989; Horton, 2006]. As nodes and their strings are likely to be scattered across main memory, access to a node can incur up to two cache misses. Clustering is a well-known technique that improves the cache-efficiency of existing string data structures [Chilimbi et al., 1999a], by storing the nodes of a linked list into a contiguous block or blocks of memory.

However, pointers are not eliminated, which is a key distinction from the techniques that we present in this chapter. Another distinction is that clustering is only compatible with homogeneous nodes. Variable-sized objects, such as strings, must remain randomly allocated in memory, which can be inefficient. A potential disadvantage of clustering is the effectiveness of move-to-front on access, which is likely to be a limiting factor as move-to-front is itself an effective cost-adaptive reordering scheme. We illustrate the structural difference between standard and clustered chains in Figure 3.1.

To maximize the effectiveness of clustering, the entire standard chain should be stored in a single contiguous block. However, adding new nodes requires the block to be resized, which can become computationally expensive; the physical location of the block can change in memory, and, as a consequence, every node in the block must be accessed and updated. Chilimbi [1999] suggests clustering nodes into fixed-sized blocks that are sized to match the cache-line. Once a block is full, a new block is allocated and the node is inserted without resizing. This technique effectively creates a chain of blocks, similar to virtual cache lines [Rubin et al., 1999]. However, unless the blocks are randomly allocated — which reduces the effectiveness of clustering — adding a new block to a set of contiguously allocated blocks can be expensive to maintain, due to the computational costs of resizing.

Details regarding these implementation issues are not generally available in the literature [Chilimbi, 1999]. As a result, we implement a *clustered* data structure by first building it as standard chain, then converting it by accessing each chain and storing its nodes contiguously in a block of memory. We describe how to convert a standard hash table, burst trie, and BST in the discussions that follow. However, we do not analyze the cache complexities of these clustered data structures, due to these implementation issues. Instead, we experimentally evaluate the benefits of clustering a standard chain hash table, burst trie, and BST, using large sets of strings, in the next chapter. In addition, we also consider the variant of clustering called *implicit* clustering, where nodes are ac-

cessed via arithmetic offsets [Chilimbi, 1999]. Implicit clustering eliminates the next-node pointers in a chain, and is therefore expected to be more cache-efficient than clustering alone. However, as a result of eliminating next-node pointers, implicit chains cannot support move-to-front on access efficiently. (We could reorder the pointers to the strings, but as we demonstrate with clustering in the next chapter, this approach is computationally expensive and is thus likely to offset the benefits of move-to-front on access). Furthermore, variable-length strings remain randomly allocated in memory, which can be inefficient. An example of an implicit chain is shown in Figure 3.1.

3.2 Compact chain

A straightforward way of improving the cache efficiency of a chained data structure is to store each string in its node, rather than storing it in a separate location. This halves the number of random accesses and saves 12 bytes per string: 4 bytes for the string pointer, and 8 bytes imposed by the operating system for string allocation (Section 2.5.1). Each node consists of 4 initial bytes, containing a pointer to the next node, followed by the string itself. The total space overhead of a compact chain is therefore $12s$, where s is the number of strings stored. This is half the overhead of a standard chain, at no cost. We call this variant of chaining *compact*. However, this procedure requires that nodes themselves be of variable length, which, depending on the programming language, can be difficult to implement. Figure 3.1 shows the structural difference between compact, clustered, and standard nodes.

The cache advantages of compact-chains are obvious: each node will involve only a single memory access; spatial locality is improved, and the reduction in total size will improve the likelihood that the next node required is already cached. Nonetheless, compact nodes are randomly allocated and remain chained. As a consequence, they cannot maximize the use of cache. However, chains are computationally inexpensive to maintain. Only pointers need to be manipulated for update operations, such as inserting a new node or when move-to-front is initiated. In practice, there are no cases — apart from ease of implementation — where a standard chain would be preferable. Hence, compact chains can be readily applied to improve the cache and space efficiency of dynamic data structures for strings that use linked lists as substructures.

3.3 The dynamic array

Compact chains are simple and effective at improving the cache and space efficiency of existing chained data structures, but the random allocation of their nodes and use of pointers can hinder the effectiveness of hardware prefetchers. Also, cache-line utilization is not optimal, as four bytes of cache-line space is wasted (the next-node pointer) per node.

We therefore propose an alternative — to eliminate the chain altogether, and store the strings in a contiguous dynamic array. Each array can be seen as a resizable *bucket*. The strings in a bucket are stored contiguously, which guarantees that access to the start of the bucket will automatically

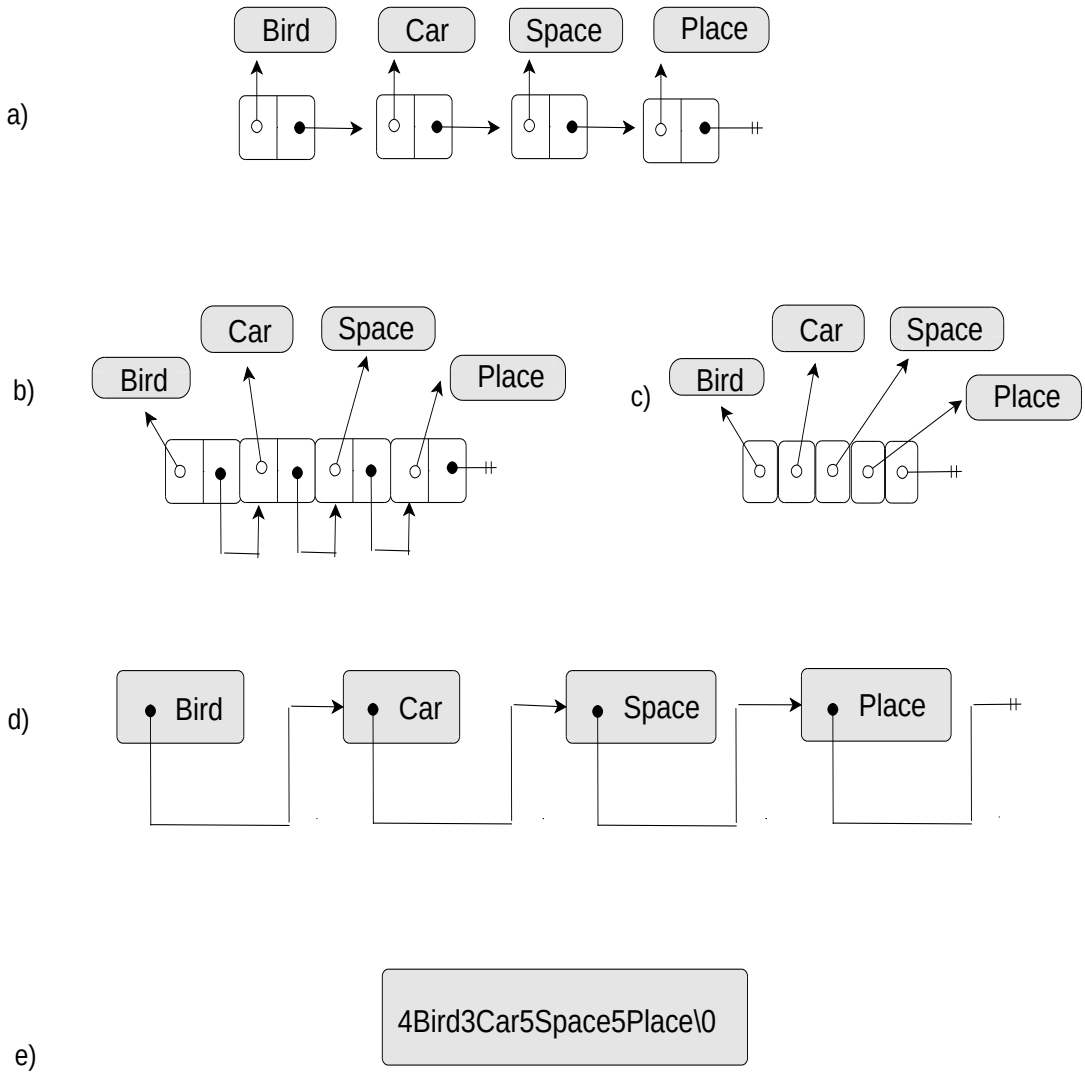


Figure 3.1: The strings "Bird", "Car","Space"", and "Place", are stored in a standard chain (a), a clustered chain (b), an implicit chain (c), a compact chain (d), and a dynamic array (e). Clustered chains do not eliminate pointers, but improve spatial access locality by storing nodes contiguously in main memory [Chilimbi, 1999]. Implicit chains, in contrast, eliminate the next-node pointers — nodes are accessed via arithmetic offsets. Compact chains eliminate string pointers by storing the string directly within the node. The dynamic array eliminates all pointers and stores strings, in order of occurrence, contiguously in a resizable array or bucket. Strings are length-encoded to permit word skipping, which is cache-efficient. The null character serves as the end-of-bucket flag.

fetch the next 64, 128, or 256 (cache-line) bytes of the bucket into cache. With no pointers to traverse, pointer chasing is eliminated and with contiguous storage of strings, hardware prefetching schemes are highly effective. Access locality is therefore maximized, creating a cache-conscious alternative to compact and standard chains.

Ghoting et al. [2006] showed that copying the nodes of an FP-tree [Han et al., 2000] into a fixed-sized memory block can greatly increase the spatial access locality of data-mining applications. A similar copying technique to eliminate string pointers for string sorting was proposed by Sinha et al. [2006], who demonstrated that doing so can halve the cost of sorting a large sets of strings. It is plausible that similar techniques can lead to substantial gains for dynamic chained data structures for strings.

While our proposal, which consists of the elementary step of dumping every string in a list into a contiguous resizable array, might be seen as simplistic, it is nonetheless attractive in the context of current architectures. Furthermore, by eliminating chains, the space overheads imposed by pointers and their subsequent memory allocation requests are eliminated. The space saved can be substantial for large sets of strings, prompting better usage of cache as there is less competition for cache-lines. Similarly, the TLB hit rate will also improve, as fewer pages of memory are required.

3.3.1 Traversing a dynamic array

The simplest way to traverse a bucket (a dynamic array) is to inspect it one 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 [Aho et al., 1974]. 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 generally not sensible to store strings of more than 2^{15} characters, as mandatory string-processing tasks, such as hashing, will utterly dominate search costs. The structural difference between a standard chain and a dynamic array is shown in Figure 3.1.

3.3.2 Growing a dynamic array

We explored two methods at growing buckets: *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. Resizing a bucket involves creating a new bucket that can fit the old bucket and the string required. The old bucket is then copied, character by character, into the new bucket; the new length-encoded string is appended followed by the end-of-bucket

flag. The old bucket is then destroyed. We analyze the cache and CPU (instruction) cost of array growth (which is quadratic to the bucket size) in Section 3.4.

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 bucket contains only a few strings. When grown, the old bucket can be copied into the new, a word (4 bytes) at time. Paging should reduce both the copying and computational overhead of bucket growth, but uses more memory. The value of 64 bytes was chosen to match the L1 cache-line size found in current Intel Pentium processors [Shanley, 2004].

3.3.3 Limitations of dynamic arrays

In contexts where lists are stored and searched, dynamic arrays are an attractive option. A potential disadvantage however is that these arrays must be of variable size; whenever a new string is inserted in a bucket, it must be resized to accommodate the additional bytes. Hence, depending on the size of the array, the frequency of growth, and the growth scheme used, array growth may become computationally expensive. Another potential disadvantage is that move-to-front — which in the context of chaining requires only a few pointer assignments — involves copying large parts of the array.

A further potential disadvantage of using buckets 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. For example, a 4-byte data field can be stored before each string. The impact of these fields is likely to be much the same for all kinds of chaining data structures.

3.4 Comparing the cache behavior of linked lists and arrays

In the environment of a computer with a cache hierarchy, the behavior of an algorithm in practice is not easy to predict analytically. Relationships between factors such as data distribution, search versus insertion rates, data volume versus cache capacities, CPU and memory speed, and the system bus, mean that modeling of expected cache performance is far from straightforward. To conduct a cache analysis, we must assume a simpler, more controlled environment.

For this, we choose a common method used to model memory access: the *ideal* cache [Frigo et al., 1999]. In this model, there is only one layer of cache; an optimal replacement policy is used to minimize cache thrashing; the cache is fully-associative; the programmer has no control over cache; and there is no interference from other programs. Other factors that can influence performance, such as the system bus, processor speed, cache associativity, write-back caches, hardware prefetch and instruction counts, are ignored.

We investigate caching in the context of traversing and growing a standard and compact linked list (or standard and compact chains, respectively), compared to an equivalent dynamic array in two cases: when they can reside entirely within cache; and when they cannot. In our ideal cache model, we assume a 512 KB cache using 128-byte cache-lines and a 64-entry TLB. To simplify measures of space, we generate fixed-length eight-character strings, by selecting characters from the English alphabet at random. The strings generated are distinct, that is, there are no repetitions.

The standard chain, compact chain, and dynamic array were implemented in C and experimentally compared on an Intel Pentium IV 2.8 GHz processor with a 64-entry TLB, 8 KB of L1 cache using 64-byte lines, and 512 KB of L2 cache using 128-byte lines. We measured the actual number of instructions, TLB misses, and L2 cache misses incurred during construction and search using PAPI [Dongarra et al., 2001]. We did not count L1 misses, as they have only a small performance penalty.

3.4.1 Case 1: the data structure fits within cache

To insert a string into a standard chain, it must first be encapsulated in a standard node. A standard node requires 8 bytes of memory, which are used to store a string pointer and a next-node pointer. We assume that nodes and their strings are stored in order of allocation, that is, a node always precedes its string. Furthermore, we assume that the pages of memory that are allocated to the data structure are also stored contiguously, in order of allocation.

In a standard chain, cache-lines can store up to 8 nodes and their strings, allowing up to $32,768$ strings to remain cache-resident; this is a total of 512 KB of memory or 128 4 KB pages. Once constructed, and assuming an empty cache, access to the first node in the chain will cause a cache-miss, fetching the next 8 nodes and their strings into cache. A full traversal — that is, when all nodes are accessed — would therefore cause 4096 cache misses. With 128 pages of memory and assuming an empty TLB, then at most 128 TLB misses will occur. Subsequent traversals of the chain should ideally incur no further cache misses and up to 128 TLB misses.

In a compact chain, string pointers are eliminated, thus halving the space overhead per string. This improves cache-line utilization, as up to 10 nodes and strings can be fetched per line, allowing up to $40,960$ strings to reside in cache. Furthermore, with less space, the number of virtual page addresses to translate is reduced, which will improve TLB performance. Storing $32,768$ strings in a compact chain requires 393 KB of space or 96 pages. Once constructed and assuming an empty cache, the cost of a full traversal reduces to 3277 cache misses and 96 TLB misses. Subsequent traversals should incur no further cache-miss and up to 96 TLB misses.

The dynamic array eliminates pointers, creating a cache-conscious alternative to chains which maximizes cache-line utilization by allowing up to 16 strings per cache-line, a total of $65,536$ strings in cache. Memory consumption is also substantially reduced, to yield further improvements in TLB performance. Storing $32,768$ strings, a dynamic array requires only 0.262 KB of memory or a total of 64 pages. Once constructed and assuming an empty cache, the cost of a full traversal is only

Data structure	Cache misses		TLB misses	
	Expected	Actual	Expected	Actual
Standard chain	4096	572	128	271
Compact chain	3277	295	96	157
Dynamic array	2048	149	64	75

Table 3.1: We compare the expected cache misses and TLB misses on an ideal cache, to the actual L2 and TLB misses reported on a Pentium IV. The task is to fully traverse (that is, access all nodes) a standard chain, compact chain, and dynamic array which are small enough to completely reside in cache. The actual values reported are averaged over a series of 200 runs. We flush the cache before each run to simulate an empty cache.

2048 cache misses and up to 64 TLB misses. All subsequent traversals should ideally incur no further cache or TLB misses.

We experimentally compare the cache performance of a standard and compact chain against the dynamic array, by measuring the actual number of L2 and TLB misses incurred on the Pentium IV. These three data structures are built using $32,768$ distinct 8-byte random strings, and are then searched using a string not stored (prompting the cost of a full traversal). We repeat the search 200 times (flushing the cache prior to each run by flooding main memory with random data which is then randomly accessed and modified), and report the average cost of L2 and TLB in Table 3.1.

The dynamic array is clearly the superior data structure, reducing both L2 and TLB misses relative to chaining by almost a factor of four. Yet the average number of L2 cache misses reported for all three data structures was considerably less than those expected on an ideal cache. For instance, the standard chain incurred only 572 L2 cache misses out of the possible 4096. With a multi-level cache hierarchy and the use of a hardware prefetcher — both of which are not considered in an ideal cache — the actual number of L2 cache misses incurred can be significantly reduced.

However, the number of TLB misses reported by these data structures was higher than those anticipated on an ideal cache. We assumed that nodes and their strings are stored contiguously in order of allocation, but in practice, this is not guaranteed. Pages are likely to be scattered across main memory and their contents are not necessarily stored in order of allocation. Hence, it is possible for a node to be stored in one page, while its string resides in another. As a consequence, up to two TLB misses can occur per node access, and, considering the limited number of TLB entries available and the demand for TLB entries from other concurrent processes run by the operating system, TLB thrashing is likely to be high. As a result, temporal access locality is reduced.

In a compact chain, string pointers are eliminated, which halves the space required by a standard chain. This has obvious advantages, in particular with the TLB. Strings are now stored

Data structure	Cache misses		TLB misses	
	Expected	Actual	Expected	Actual
Standard chain	4096	564	128	259
Compact chain	3277	251	96	132
Dynamic array	2048	88	64	73

Table 3.2: We compare the cache misses and TLB misses that are predicted on an ideal cache, against the actual L2 and TLB misses reported on a Pentium IV. The task is to fully traverse (that is, access all nodes) a standard chain, compact chain, and dynamic array which are small enough to completely reside in cache. The actual values reported are averaged over a series of 200 runs. However, in contrast to Table 3.1, we do not flush the cache before each run, in order to promote cache-residency.

directly after their next-node pointers, resulting in at most a single TLB-miss on access. As a result, the cache cost for traversing a compact chain should be around half that of a standard chain, as indicated by Table 3.1.

Our next experiment attempts to make these data structures cache-resident, by repeating the previous experiment but without flushing the cache. Intuitively, runs after the first should incur no further cache misses. We report the costs in Table 3.2. Surprisingly, the average L2 and TLB misses remained almost as high as our previous experiments in Table 3.1. Considering the limited number of TLB entries available and the potentially high competition to acquire cache-lines caused from other (mandatory) processes that are run by the operating system, the L2 or TLB entries are unlikely to persist long enough to be re-used. Hence, although these data structures can be small enough to reside in cache, the ideal situation of incurring no cache misses during traversal is unlikely to occur in practice.

3.4.2 Case 2: the data structure is larger than cache

Assuming an initially empty ideal cache, the cost (in cache misses) for fully traversing a standard chain is $S/8$, where S is the number of strings stored. In addition, up to $S(L+8)/P$ TLB misses will occur, where L is the string length (8 bytes in our case) and P is the page size in bytes. A compact chain can store up to ten nodes and strings per cache-line. The cost of a full traversal therefore reduces to $S/10$ cache misses, with at most $S(L+4)/P$ TLB misses. The dynamic array maximizes cache-line utilization, storing up to 16 strings per cache-line. As a result, the cost of a full traversal falls to $S/16$ cache misses and at most $(SL)/P$ TLB misses.

We experimentally compare the cache performance of the standard and compact chains against a dynamic array on the Pentium IV. We build these structures using our fixed-length (8-byte) random strings, starting with a set of $16,384$ strings, doubling the dataset size up to $1,048,576$

strings. Figure 3.2 shows the average L2 cache misses, TLB misses, and instructions executed (averaged over a series of ten runs), to fully traverse these data structures.

As observed in Case 1, the actual number of L2 cache misses for all three data structures was considerably less than those expected on an ideal cache. As discussed, factors such as the presence of a multi-layer cache hierarchy and hardware prefetchers, can significantly reduce cache-miss rates. Similarly, the number of TLB misses reported was higher than anticipated, as a result of TLB thrashing caused by the small number of TLB entries available. In addition, the ideal cache assumes an optimal replacement algorithm, which is not realistic. Processors typically use a LRU or random replacement policy [Intel, 2007], which is likely to increase cache-line thrashing. Nonetheless, the relative difference in cache performance between these three data structures was as anticipated. The compact chain demonstrated better cache performance over a standard chain, with the dynamic array displaying the best cache utilization.

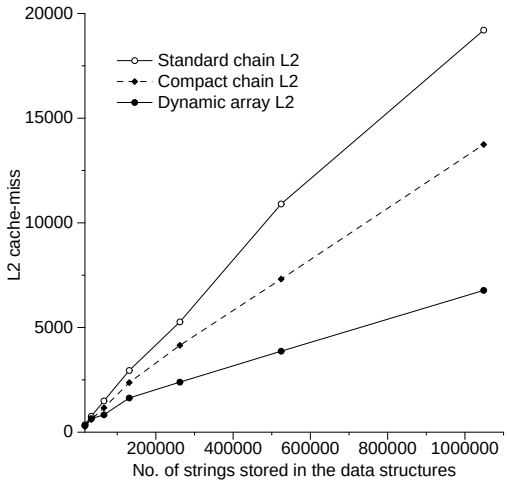
The number of instructions required to traverse a data structure can also have an impact on overall performance, but the ideal cache model does not capture the cost of instructions. From the results shown in Figure 3.2, the dynamic array is clearly more cache-efficient than chaining, but is computationally expensive to access. This is due to the use of length-encoded strings, which must be decoded prior to comparison. This requires that extra instructions be executed, such as checking whether 1 or 2 bytes are used to store the string length. Similarly, the compact chain required slightly more instructions than a standard chain, due to the extra steps taken to skip over the first 4 bytes (the next-node pointer) of each string.

To investigate the impact of these instructions on overall access time, we compare the time taken to build and then fully traverse the standard chain, the compact chain, and the dynamic array. Results are shown in Table 3.3 and the instruction and cache costs are illustrated in Figure 3.3. As a result of its high computational cost, the dynamic array is slightly slower to search.

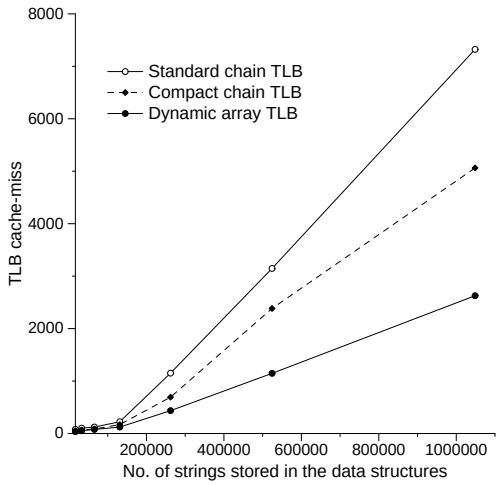
The difference in access time becomes more apparent when we compare the cost of construction in Table 3.3. Growing a dynamic array using *exact-fit* (as done in these experiments) implies that, on every insertion, the old array is copied into a new larger array. From Figure 3.3, we show that the dynamic array is cache-efficient to grow. However, the time saved in cache cannot compensate for the time lost due to the excessive numbers of instructions required to grow the array. The CPU (instruction) cost of array growth is quadratic to the array size.

As a consequence, with exact-fit, it is more efficient to grow a single large chain than it is to grow a single large array. Our *paging* technique can significantly reduce the computational cost of growing arrays, but at the expense of space. Although the compact chain requires more instructions to build than a standard chain, the improved use of cache greatly compensates, resulting in improved access times. Hence, it is more efficient to use a chain than a dynamic array, when strings are inserted and accessed in a single list.

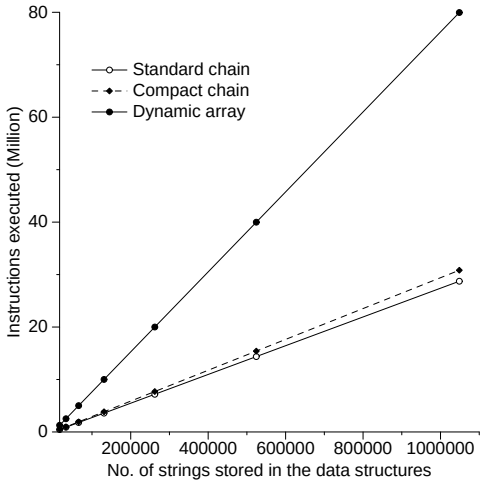
However, with an access cost of $O(N)$ — where N is the number of nodes — a linked list is



(a) L2 misses

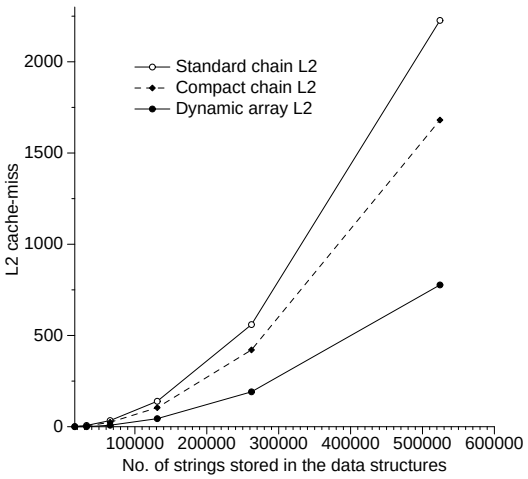


(b) TLB misses

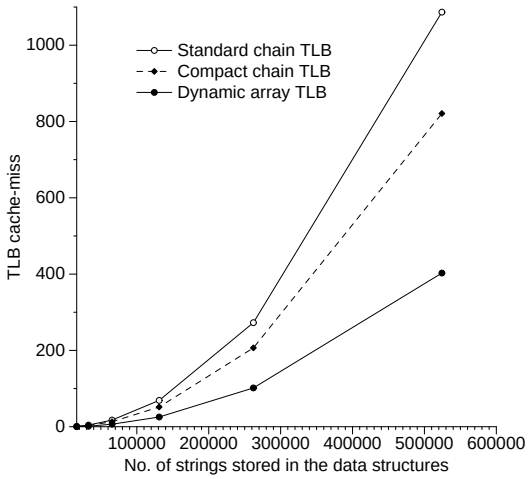


(c) Instructions

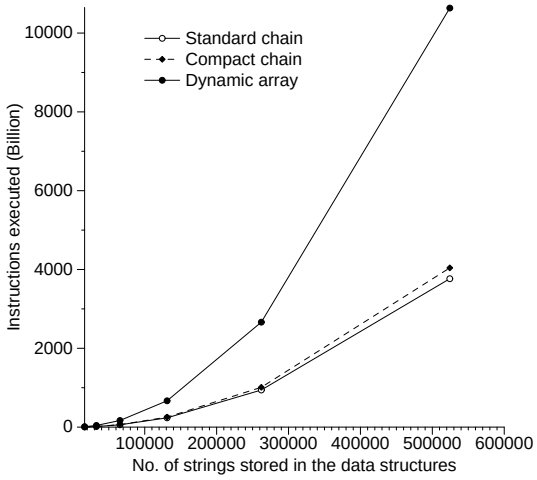
Figure 3.2: L2 cache misses, TLB misses, and instructions executed when fully traversing (that is, accessing all nodes) a standard, compact, and array-based list (a dynamic array), starting with a set of 16,384 fixed-length strings, doubling the dataset size up to 1,048,576 strings.



(a) L2 misses



(b) TLB misses



(c) Instructions

Figure 3.3: L2 cache misses, TLB misses, and instructions executed when building a standard, compact, and array-based list (a dynamic array), starting with a set of 16,384 fixed-length strings, doubling the dataset size up to 1,048,576 strings.

Data Structure	String Cardinality	Build time (seconds)	Full traversal (seconds)
Standard chain	16384	1.3	0.001
	32768	6.5	0.001
	65536	27.9	0.002
	131072	113.4	0.003
	262144	455.1	0.005
	524288	1819.5	0.010
	1048576	7295.3	0.019
Compact chain	16384	1.2	0.001
	32768	5.5	0.001
	65536	23.9	0.002
	131072	96.9	0.003
	262144	388.1	0.005
	524288	1553.7	0.010
	1048576	6210.6	0.017
Dynamic array	16384	3.1	0.001
	32768	12.9	0.002
	65536	56.1	0.003
	131072	226.5	0.008
	262144	897.1	0.012
	524288	3608.5	0.019
	1048576	14489.4	0.030

Table 3.3: The elapsed time in seconds required to build a standard chain, a compact chain, and a dynamic array, using randomly generated fixed-length strings. The cost of construction grows in proportion to the number of strings inserted, as expected. Despite its high reductions in cache misses, however, the dynamic array is more expensive to access than the equivalent chains, due to the computational overhead of resizing. The cost of a full traversal is similar, that is, we see a small increase in search time due to the added overhead of processing length-encoded strings.

obviously not a viable solution with large datasets. The various types of trees, tries, and hash tables can be used to reduce access costs. The key distinction in these data structures is that strings are scattered amongst a set of smaller chains, such as the slots of a hash table or the containers of a burst trie. This can have a profound impact on the cache performance of chains. Although computationally efficient, maintaining a set of chains, where any chain in the set can be accessed and grown at random, leaves the operating system with little opportunity to allocate the nodes of a particular chain contiguously in main memory. The nodes of a standard or compact chained data structure are therefore likely to be scattered in main memory. As a consequence, neither temporal or spatial locality is high, and, combined with pointer chasing, hardware prefetching is ineffective. Traversing these chains can therefore attract excessive cache misses, which will greatly overwhelm any savings acquired as a result of their computational efficiency.

We experimentally demonstrate this in Chapter 4, where we show the substantial gains in performance acquired by applying our compact-chain and dynamic array techniques to the standard chained hash table, burst trie, and BST. We now describe how to redesign these three data structures to create cache-conscious alternatives.

3.5 Cache-conscious hash tables

To develop a cache-efficient hash table, we focus on how collisions are resolved. A collision occurs when more than one string is hashed to a particular slot. The best method for resolving collisions for dynamic sets of strings are chains; simply append the new string at the end of the chain. With move-to-front on access, the chaining hash table or *standard* hash table, is the fastest and most compact data structure available when sorted access is not required [Zobel et al., 2001]. The use of standard chains can however result in poor use of cache, making this computationally efficient data structure vulnerable to severe performance penalties on current cache-oriented processors. We can potentially improve the cache-efficiency of the standard hash table, by converting it into a *clustered* hash table. That is, we visit every slot and store its nodes, in order of occurrence, contiguously in main memory.

We propose to replace the standard chains of a hash table with compact chains, to yield a more cache- and space -efficient alternative that we call a *compact* hash table. Although compact chains eliminate string pointers, nodes remain scattered in main memory. We eliminate these chains by using dynamic arrays. This will create a hash table with a cache-conscious collision resolution scheme that we call an *array* hash. Note however that this method does not change the size of the hash table; we are not proposing extendible arrays [Rosenberg and Stockmeyer, 1977]. The array hash is an attractive option for current cache-oriented architectures, as it can maximize cache usage while simultaneously reducing space. As discussed earlier, strings within dynamic arrays are expensive to reorder, so move-to-front on access is likely to become a performance bottleneck. The structural differences between a standard, compact, and array hash table, are shown Figure 3.4.

3.5.1 The array hash table

The procedures taken to insert, retrieve, and delete strings in the standard or compact hash tables are straightforward. The string is first hashed (using the bitwise hash function [Ramakrishna and Zobel, 1997], say) to acquire a slot. The slot is then accessed and its chain is traversed, checking each node until a match is found. When a match is found, the node is moved to the front of the chain through pointer manipulation. Alternatively, the node can be deleted. If the slot is empty, or no match is found, then the search fails, and the string can be inserted. In this case, the string is encapsulated in a standard or compact node, which is then added to the end of the chain. The steps taken to insert, search, and delete in an array hash are slightly more complex and are described below.

To search for a string

A search begins by first hashing the string (using the bitwise hash function) to acquire a slot. If the slot is assigned to a bucket, the search proceeds as outlined in Section 3.3.1: the first length-encoded string is compared, character by character. On mismatch, the next length-encoded string is accessed, and so forth, until a match is found or the end-of-bucket flag is seen, on which, the search fails. On a match, the string can be moved to the start of the array. However, due to the potentially high computational costs involved, move-to-front is optional.

To insert a string

We select a slot by hashing the string using the bitwise hash function. If the slot is empty, then a new bucket is created and assigned. The bucket is sized according to the growth policy used. The string is then length-encoded and copied into the bucket, followed by the end-of-bucket flag. Otherwise, if the slot has a bucket assigned, we search for the string as described above. On search failure, the bucket is grown according the growth policy used, and the string is length-encoded and appended (as described in Section 3.3.2).

To delete a string

We select a slot by hashing the string using the bitwise hash function. Assuming that the slot is not empty, we search for the string as described above. If found, we create a new bucket that is sized to match the space required by the old bucket, minus the length of the string to delete. If deleting the string results in an empty bucket, then the slot pointer is nulled and the deletion process is complete. Otherwise, we scan the original bucket and copy across all strings except for the string to delete. The old bucket is then destroyed and the new is assigned.

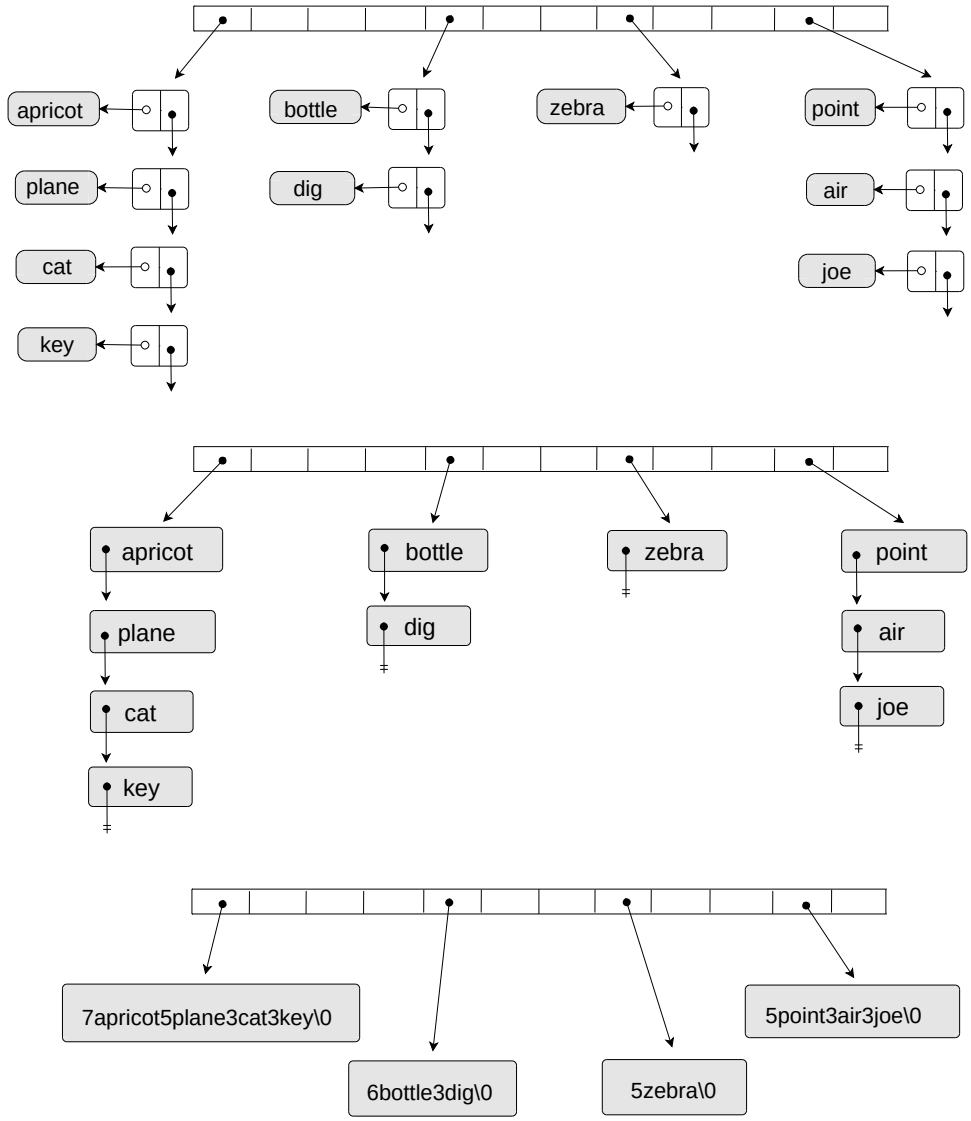


Figure 3.4: The application of compact chains and dynamic arrays to the standard chain hash table. In a compact hash table, string pointers are eliminated, allowing the strings themselves to be represented as nodes. In the array hash table, all pointers — apart from the slot pointers — are eliminated. Strings are stored contiguously in main memory to make good use of both cache and hardware prefetch. Strings are appended in order of occurrence and are length-encoded to permit word-skipping, which is cache-efficient.

Notation	Meaning
N	Number of strings inserted
M	Number of slots
C	Cache-line size in bytes
P	Page size in bytes
L	Length of the string (includes null character)
W	The size of a pointer in bytes
α	The average load factor, which is: N/M

Table 3.4: Notation used for analyzing the expected cache costs of a hash table.

3.5.2 Expected cache costs

We theoretically analyze the cache performance of the standard, compact, and array hash tables, to gain a better understanding of expected reductions in cache misses. We simplify our discussions by assuming the following:

- Use of an ideal cache, as described in Section 3.4.
- The hash tables are many times larger than cache.
- A uniform data distribution, that is, there is no skew.
- Strings are of uniform length and are randomly generated; each character is randomly selected from the English alphabet. There are no string duplicates.
- The hash function used is from a universal class [Sarwate, 1980] so that strings are distributed as well as possible, regardless of data distribution. The best hash function for strings that is thought to be from a universal class is the bitwise hash method [Ramakrishna and Zobel, 1997].

By using a uniform data distribution and a bitwise hash function, we can assume that every slot in the hash table has equal probability of access. Furthermore, the bitwise hash function makes it extremely unlikely for all strings to be hashed to a single slot. Hence, the best-and worst-case complexity of the chaining hash table is $O(N/M)$ [Zobel et al., 2001; Knuth, 1998].

We assume that every access to a hash table causes two initial cache misses. First, when the slot is accessed, and second, when the slot pointer is followed. All subsequent cache misses are incurred during slot traversal. In a standard hash table, this involves traversing a linked list. The number of nodes we expect to access on average is N/M or α. Once a node is accessed, a string pointer is followed, which can attract a further cache-miss. The nodes of a standard hash table contain two W-byte pointers. Hence, traversing a slot in a standard hash table will incur up to

$2W\alpha/C$ cache misses on node access, and up to a further $L\alpha/C$ cache misses on string access. Hence, the expected L2 cache misses for searching a standard chain hash table is up to:

$$\begin{aligned} & 2 + \frac{2W\alpha}{C} + \frac{L\alpha}{C} \\ = \quad & 2 + \frac{\alpha(2W + L)}{C} \end{aligned} \tag{3.1}$$

Similarly, the expected number of TLB misses is up to:

$$\begin{aligned} & 2 + \frac{2W\alpha}{P} + \frac{L\alpha}{P} \\ = \quad & 2 + \frac{\alpha(2W + L)}{P} \end{aligned} \tag{3.2}$$

In a compact hash table, string pointers are eliminated, allowing the strings themselves to be represented as nodes. Each string is preceded by W bytes, used to store the next-node pointer. Hence, the expected L2 cache misses for searching a compact chain hash table is up to:

$$2 + \frac{WL\alpha}{C} \tag{3.3}$$

Similarly, the expected number of TLB misses is up to:

$$2 + \frac{WL\alpha}{P} \tag{3.4}$$

In an array hash, all pointers — apart from slot pointers — are eliminated, and strings are contiguously allocated to maximize cache-line utilization. Apart from the initial two cache misses caused when accessing the slot and its pointer, all subsequent misses occur on array access. Hence, the expected L2 cache misses for searching an array hash is up to:

$$2 + \frac{L\alpha}{C} \tag{3.5}$$

Similarly, the expected number of TLB misses is up to:

$$2 + \frac{L\alpha}{P} \tag{3.6}$$

Although analytically similar to the compact hash table, the array hash offers a key advantage not reflected in the ideal cache model, which is the contiguous allocation of strings. In a compact hash table, we assume that nodes (strings) are contiguously allocated to allow $WL\alpha/C$ cache misses on traversal. In practice, however, nodes in a compact chain are likely to be scattered across main memory and are accessed via pointers. This will cause pointer chasing and hinder the effectiveness of hardware prefetchers. In an array hash, strings are guaranteed to be contiguously allocated, and, with no pointers to follow — apart from the initial slot pointers — use of cache and hardware prefetchers can be maximized, leading to substantial reductions in cache misses.

Update operations on a standard or compact hash table have the same cost as traversal. The small and constant overhead of adding or deleting a new node can be ignored. In an array hash, however, inserting or deleting a string involves array copying, which can attract additional cache misses. We assume that the array hash is grown using *exact-fit*. When an update takes place, the bucket to update is scanned twice: during initial search and when copied into a new larger array. Each scan will incur up to $L\alpha/C$ cache misses. Hence, the expected L2 cache misses to update an array hash is up to:

$$2 + 2\left(\frac{L\alpha}{C}\right) \tag{3.7}$$

Similarly, the expected TLB misses on update is up to:

$$2 + 2\left(\frac{L\alpha}{P}\right) \tag{3.8}$$

The array hash table is more expensive to update than to search, but as we have shown, its overall expected cache costs on update remain less than the update costs of the equivalent standard or compact-chain hash table.

3.6 Cache-conscious burst tries

The *standard chained* burst trie is currently the fastest and most compact data structure available for vocabulary accumulation, when sorted access to strings is required [Heinz et al., 2002]. Although computationally efficient, the burst trie uses standard chains as containers, which is neither cache- nor space -efficient. To improve the efficiency of the burst trie, we replace the standard chains with compact chains, to create a more efficient alternative called a *compact* burst trie. Similarly, we can eliminate these chains by representing containers as dynamic arrays. This gives us a cache-conscious burst trie or *array* burst trie.

The standard, compact, and array burst tries maintain array-based trie nodes that map directly to the 128 characters of the ASCII table. Each trie node is therefore 512 bytes long (assuming 4-byte pointers). The first 32 pointers and the last pointer, however, map to non-printing characters, which can be ignored. We therefore reserve the first pointer of each trie node to store a string-exhaust flag, which is explained later. Hence, our implementations of the burst trie are compatible with an alphabet of 94 characters. We could eliminate unused pointers, but we found that the amount of space saved in practice was too small to justify the increased cost in trie node access.

In a typical implementation, trie nodes are randomly allocated, which is not cache efficient. Our trie nodes are therefore maintained in a dynamic array, in order of allocation. This improves access locality and eliminates operating system overheads. Containers in a standard or compact burst trie occupy five bytes of memory. The first four bytes store a pointer to the start of the chain. The next byte maintains housekeeping information: the string-exhaust flag. The string-exhaust flag indicates whether all the characters of a string have been removed during traversal. Once a trie node or a container removes the last character of the string used to traverse the burst trie, their string-exhaust flag is set. The physical ordering of these two fields is important. For efficiency, pointers should always start at a word boundary (an address divisible by 4). If a pointer is stored between two words, and assuming a 32-bit system bus, two bus cycles are required to fetch the pointer into cache [Hennessy and Patterson, 2003], which is inefficient.

Alignment is not an issue for the containers of an array burst trie, because pointers are eliminated. In an array burst trie, each container reserves the first two bytes for housekeeping: to maintain the string-exhaust flag and a flag to indicate whether a string has been inserted. To implement a *clustered* burst trie, we convert a fully built standard burst trie by accessing all containers and storing their nodes, in order of occurrence, contiguously in main memory. We now describe how to insert and search for strings in a standard, compact, and array burst trie. The structural differences between a standard, compact, and array burst trie, are shown Figure 3.5.

3.6.1 Initialization

The burst trie begins as an empty container with no trie nodes. The container is populated with strings until a threshold is met. Three thresholds — ratio, limit, and trend — are evaluated by Heinz et al. [2002], with the simplest and most effective being a *limit*, which bursts a container once it stores more than a fixed number of strings. In the discussions that follow, we assume that there is at least one trie node present (a root trie).

3.6.2 To search for a string

Search proceeds as follows. The first character of the string (the lead character) is used as an offset into the root trie node to acquire a pointer. The pointer is followed to fetch the next node. Whenever a pointer from a trie node is traversed, the lead character of the string is deleted or

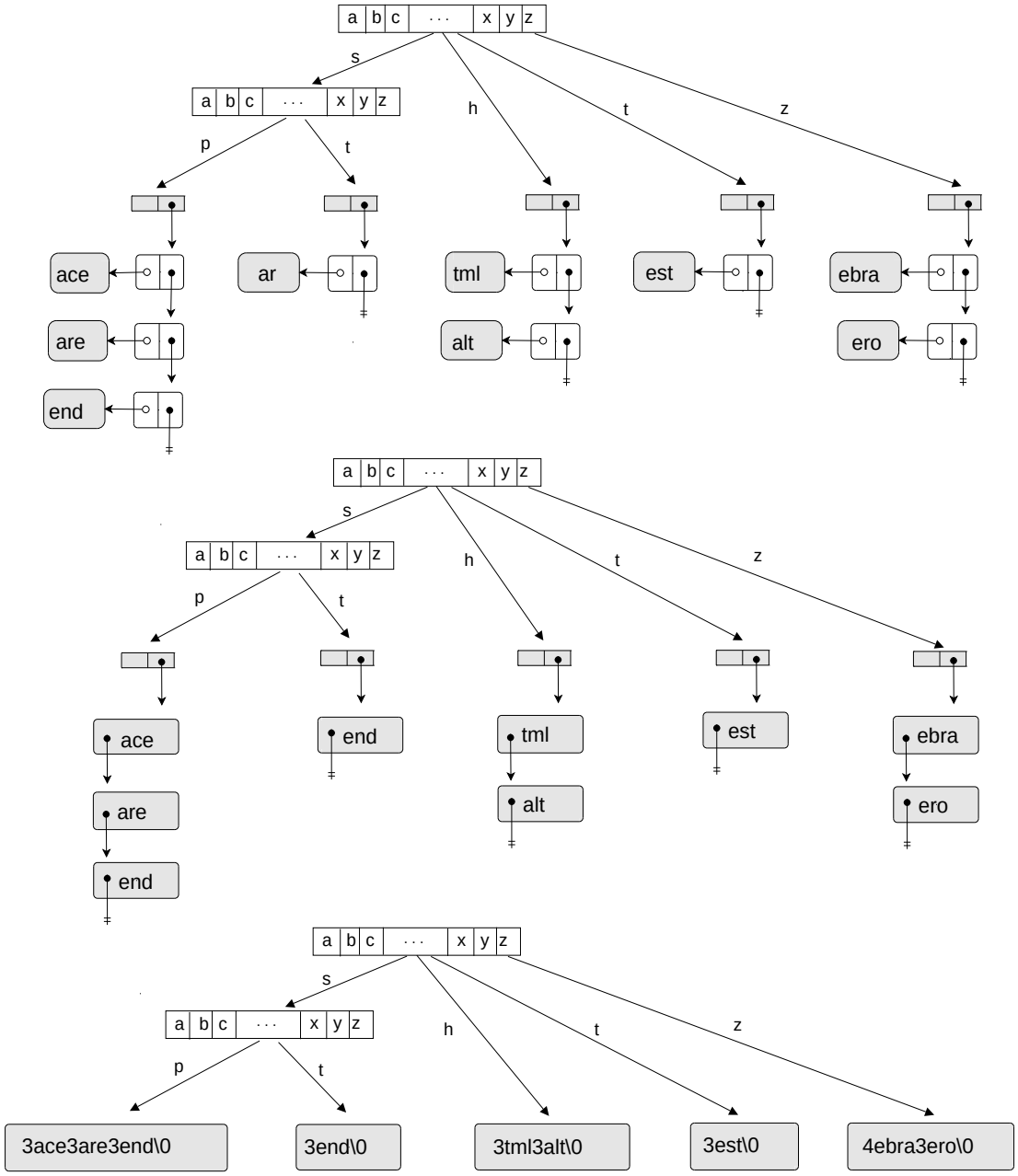


Figure 3.5: The strings "space", "spare", "spend", "star", "html", "halt", "test", "zebra", and "zero" are stored in a standard burst trie (top), a compact burst trie (middle), and an array burst trie (bottom).

consumed. If a null pointer is encountered, then the search fails. Otherwise, the process is repeated until a container is acquired.

It is possible to exhaust a string before acquiring a container, that is, to delete all of its characters. When this occurs, the first pointer (which represents the string-exhaust flag) from the last trie node accessed, is inspected. The search is successful if the flag is set, otherwise it is a failure. Similarly, the string can be exhausted on initial access to a container. When this occurs, the string-exhausted flag is accessed within the container, and, if set, the search is successful; otherwise it is a failure.

If the string is not exhausted, then a container is acquired and searched. In a standard or compact burst trie, the search involves comparing every string in the container, via a linked list traversal. On match, the node containing the required string is moved to the front of the list, and the search is successful. Otherwise, the list is exhausted and the search fails.

In an array burst trie, the container is a dynamic array and the search proceeds as described in Section 3.3.1: the first length-encoded string is accessed and compared, character by character. On mismatch, the next length-encoded string is accessed and so forth, until the end-of-bucket flag is encountered, on which the search fails. On a match, move-to-front is optional, due to the high computational costs involved.

3.6.3 To insert a string

An insertion can only proceed on search failure, which occurs in one of four ways: the string is exhausted during trie traversal; the string is exhausted on initial access to a container; a null pointer is encountered; or the string is not found in a container. The two cases that fail due to string exhaustion are resolved by setting the string-exhaust flag in the acquired trie node or container, completing the insertion process.

When the search fails due to a null trie-node pointer, we assign the pointer to a new empty container. This will immediately consume the lead character of the string, which could exhaust the string. In this case, the container remains empty and its string-exhaust flag is set, completing the insertion process.

Otherwise, the string is inserted into the container, which is also how the last of the four cases is handled. For a standard or compact burst trie, the string is first encapsulated in a standard or compact node, respectively, and then added to the end of the chain. In an array burst trie, the string is appended to the container through array resizing, as described in Section 3.3.2. After a string is inserted into a container, the container size must be checked to determine whether a burst is required.

3.6.4 To delete a string

To delete a string, we first search for it as described above. If the search leads to a string-exhaust flag, the flag is cleared and deletion is complete. Otherwise, a container is acquired and searched. Assuming that the string is found, deleting it in a standard or compact container is straightforward. In an array burst trie, however, a new container is allocated and sized to match the old container, minus the length of the string to delete. The old container is then scanned and its strings, apart from the one to delete, are copied into the new container. The old container is then deleted and the new is assigned. Empty containers are deleted, unless their string-exhaust flag is set. Deleting an empty container will clear its parent pointer, which may cause the parent trie-node to become leaf-less, in which case, it too is deleted; the deletion of trie nodes can propagate up to the root node.

3.6.5 Bursting a container

A container is burst once its size exceeds a selected threshold. The choice of threshold and the definition of a container's size is described below. To burst a container, it is first detached from its parent pointer, which is then assigned to a new trie node. The strings in the container are then accessed and distributed into at most A new containers (A being the alphabet size, which in our case is 94), according to their lead character, which is then removed. Bursting a container can therefore exhaust up to A strings. Once bursting is complete, the original container is deleted and the new trie node becomes the parent of the new containers.

Choosing a container size

A container is burst once it exceeds a certain size or limit, being the number of strings maintained. Choosing a limit requires some care. Large containers — a limit of over 100 strings for example — are not burst as often, and can therefore reduce the net number of nodes created, saving space but at a cost of access time. Yet to some degree, the impact on time depends upon the distribution of strings. Consider for example a large compact or standard-chain container that is accessed under heavy skew. With move-to-front, the majority of searches are likely to terminate on the first node. Similarly, under heavy skew, large containers in an array burst trie are likely to remain efficient, even without move-to-front, because on initial access an entire cache-line of strings is prefetched, with the next few lines being likely candidates for hardware prefetch.

When little to no skew is present, however, move-to-front is rendered ineffective, and as a consequence, large compact or standard-chain containers will become expensive to access. The performance of large array-based containers, however, are likely to remain competitive due to their high spatial access locality. Smaller compact or standard-chain containers will be more efficient to access in this case, but will also greatly increase the number of trie nodes and containers allocated, which will consume a lot of space. In addition, as the number of trie nodes increase, temporal

Notation	Meaning
N	Number of strings inserted
C	Cache-line size in bytes
P	Page size in bytes
L	Length of the strings (includes null character)
A	The size of the alphabet used
W	The size of a pointer in bytes
B	The maximum number of strings in a container

Table 3.5: Notation used for analyzing the expected cache costs of a burst trie.

access locality can be reduced, as previously cached containers are likely to be systematically flushed out of cache by trie nodes.

An alternative scheme for the array burst trie is to change the limit of a container to capture its physical size, rather than the number of strings it can store prior to bursting. The advantage of this approach is that containers can match the size of a cache-line, and thus incur at most, only a single L2 cache-miss on access. Long strings, however, can complicate matters, especially when they cannot fit within fixed-sized containers. In this case, a surplus of trie nodes is likely to be generated.

Choosing a good limit for the burst trie is therefore not a straightforward task, and is difficult to determine analytically. A small limit is likely to reduce access time but at the expense of space. A large limit will save space, but at a likely cost in access time. Key factors to consider when deciding on a limit include the expected distribution of strings, and the type of burst trie (standard, compact, or array) used. For the task of vocabulary accumulation, Heinz et al. [2002] suggested a limit of 35 strings for the standard burst trie, which they derived experimentally. In the next chapter, we derive a set of limits from our experiments, which should provide good performance in both time and space, for a variety of string distributions.

3.6.6 Expected cache costs

We theoretically analyze the cache performance of the standard, compact, and array burst trie, to gain a better understanding of the expected reductions in cache misses. We simplify our calculations by assuming the following:

- Use of an ideal cache, as described in Section 3.4.
- The burst tries are many times larger than cache.
- A uniform data distribution, that is, there is no skew.

- Strings are of uniform length and are randomly generated; each character is randomly selected from the alphabet A. There are no string duplicates.

The array trie (discussed in Section 2.3.1) has been shown to have logarithmic access complexity [Clement et al., 2001]. Assuming uniform length strings, the height of an array trie is:

$$\log_A (N) \tag{3.9}$$

The burst trie is an array trie where subtrees are stored in containers. The burst trie maintains the expected logarithmic access costs of the array trie [Heinz et al., 2002; Knessl and Szpankowski, 2000; Szpankowski, 1991], but with a reduction in the number of trie nodes, due to the use of containers. We assume that every trie node and container accessed incurs a cache miss. Hence, the expected cache misses on traversal is:

$$\log_A \left(\frac{N}{B}\right) \tag{3.10}$$

However, this only captures the cache complexity of traversing the trie index and the initial access to a container. Traversing a container will attract further cache misses. Containers can store up to B strings. In a standard burst trie, strings are encapsulated as standard nodes. Each node contains two W-byte pointers, and hence, traversing a standard chain container will incur up to $2WB/C$ cache misses. Each node accessed results in a string pointer traversal, which can attract a further cache-miss. Hence, the expected L2 cache misses for searching a standard burst trie is up to:

$$\begin{aligned} & \log_A \left(\frac{N}{B}\right) + \frac{2WB}{C} + \frac{LB}{C} \\ = \; & \log_A \left(\frac{N}{B}\right) + \frac{B(2W+L)}{C} \end{aligned} \tag{3.11}$$

Similarly, the number of TLB misses for search is up to:

$$\begin{aligned} & \log_A \left(\frac{N}{B}\right) + \frac{2WB}{P} + \frac{LB}{P} \\ = \; & \log_A \left(\frac{N}{B}\right) + \frac{B(2W+L)}{P} \end{aligned} \tag{3.12}$$

In a compact-chain burst trie, string pointers are eliminated, allowing the strings themselves to be represented as nodes. Each string is preceded by W bytes, used to store the next-node pointer.

Hence, the expected L2 cache misses for searching a compact-chain burst trie is up to:

$$\log_A \left(\frac{N}{B}\right) + \frac{WLB}{C} \tag{3.13}$$

Similarly, the number of expected TLB misses is up to:

$$= \log_A \left(\frac{N}{B}\right) + \frac{WLB}{P} \tag{3.14}$$

In an array burst trie, all pointers within containers are eliminated. Traversing the array-based container will therefore incur up to LB/C cache misses. Hence, the expected L2 cache misses for searching an array burst trie is up to:

$$\log_A \left(\frac{N}{B}\right) + \frac{LB}{C} \tag{3.15}$$

Similarly, the expected TLB misses is up to:

$$= \log_A \left(\frac{N}{B}\right) + \frac{LB}{P} \tag{3.16}$$

Although analytically similar to the compact burst trie, the array burst trie eliminates pointer chasing within containers by allocating strings contiguously. This results in better use of cache and hardware prefetch, which can lead to substantial reductions in cache misses. Increasing the size of B will reduce the number of trie nodes, and thus the number of cache misses incurred prior to traversing a container. However, this will also make container traversal more expensive, in particular for the compact and standard burst tries. The array burst trie, however, is expected to remain cache-efficient with large containers, due to high spatial access locality. Hence, the array burst trie is expected to reduce space — by using large containers — without compromising performance.

The expected cache costs for inserting or deleting a string in a standard or compact burst trie — assuming that a burst does not occur — is equivalent to their traversal costs. The small and constant overhead of adding or deleting a node can be ignored. In an array burst trie, an update involves array copying, which can attract further cache misses. Assuming that *exact-fit* is used and that at least one string remains in a container, on update, a container is scanned twice — once

to search and then again to copy into a new larger array. Hence, the expected L2 cache misses to update an array burst trie is up to:

$$\log_A \left(\frac{N}{B}\right) + 2\left(\frac{LB}{C}\right) \tag{3.17}$$

Similarly, the expected TLB misses to update is up to:

$$= \log_A \left(\frac{N}{B}\right) + 2\left(\frac{LB}{P}\right) \tag{3.18}$$

3.7 Cache-conscious binary search trees

Compact chains can be applied to all types of binary search tree. In this section, we concentrate on the *standard* BST, which is among the fastest and simplest of tree structures [Williams et al., 2001; Bell and Gupta, 1993]. We replace the standard nodes of a binary search tree with compact nodes, yielding a cache- and space -efficient variant called a *compact* BST. Search and insert in a compact and standard BST is straightforward. The root node is compared against the string, and, on mismatch, the left pointer is followed if the string is lexicographically less than or equal to the node; the right pointer is followed otherwise. This process continues down the tree until a match is found, or until an empty or null pointer is encountered. In this case, the search fails and the string can be inserted, by first encapsulating it as a compact or standard node, which is assigned to the empty pointer. To convert a fully built standard BST into a *clustered* BST, we contiguously allocate subtrees based upon the structural topology of the BST, as described by Chilimbi [1999]. That is, we first store the root subtree — the root node and its left and right child (if any) — into a block of memory. The block is sized to match the cache-line. We then traverse the tree, in a breath-first manner, and store the next layer of subtrees into the block. Once the block overflows, a new block is allocated and the traversal continues until all subtrees are packed into blocks of memory.

The application of a dynamic array to the standard BST, however, requires greater care, as, for efficiency purposes, not all pointers can be eliminated. An array-based representation of a BST is described by Knuth [1998]. This algorithm takes an existing BST and collapses the nodes (including their keys) into a pre-order traversal, in a single contiguous array, which is also known as a linear representation [Jonge and Tanenbaum, 1987]. The single array is guaranteed to be accessed in a left-to-right manner, which improves access locality and thus cache-efficiency. Furthermore, with nodes stored in a pre-order traversal, all left child-node pointers are now redundant and can

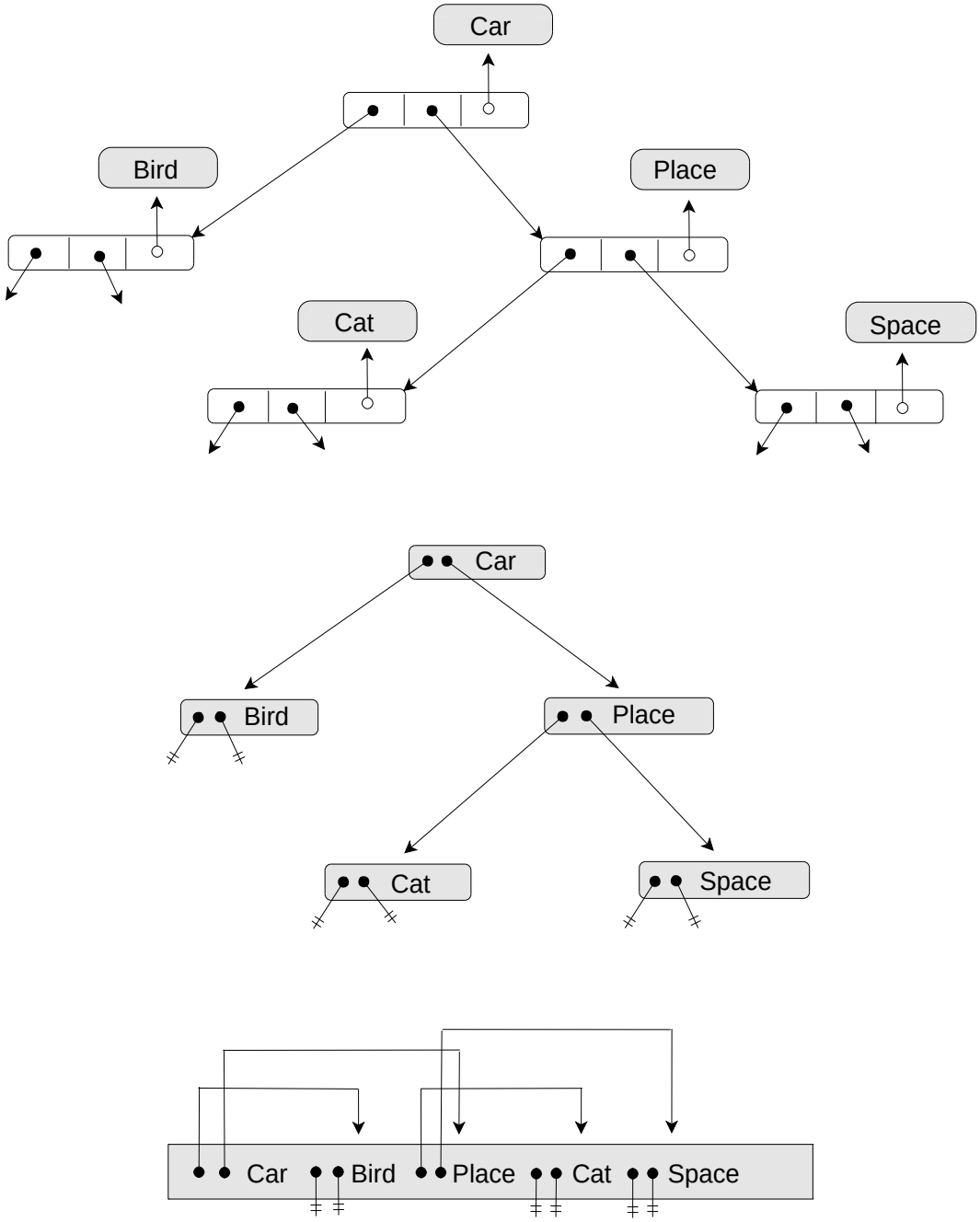


Figure 3.6: The strings "Car", "Bird", "Place", "Cat", and "Space" are stored in a standard binary search tree (top), a compact binary search tree (middle), and an array binary search tree (bottom). The strings are appended in order of occurrence in the array BST, to ensure left-to-right access on traversal, which is cache-efficient.

Notation	Meaning
N	Number of strings inserted
C	Cache-line size in bytes
P	Page size in bytes
L	Length of the strings (includes null character)
W	The size of a pointer in bytes
X	Number of strings in a subtree, where: $X < N$

Table 3.6: Notation used for analyzing expected cache costs of a binary search tree.

be removed, saving space. This algorithm, however, assumes fixed-length keys. It can be adapted to store variable-length strings, but as a consequence, it can become expensive to update and, in some cases, to search; although the left-child is guaranteed to be located after its parent, access to it involves a linear scan, and, to maintain pre-ordering of nodes, a large section of the array may need to be shifted to store a new string in the correct position.

Our array-based approach differs in that we do not build then collapse a standard BST into an array. Instead, we propose the novel approach of storing the entire BST in a single dynamic array, which is a key distinction to a clustered BST. On insertion, a string is represented as a *compact* node, with both left and right child-node pointers stored before the string. The node is then appended to the end of the array, which is grown (when full) using *paging*; we add 10 MB of space per call — a value found through preliminary trials that can achieve a good balance between time and space — to reduce the computational costs of copying. On search, the first node in the array (which is always the root node) is accessed and binary search proceeds as described. Although our *array* BST is not as space-efficient as Knuth's algorithm, retaining both child-node pointers permits efficient update and search. Moreover, with nodes being appended in order of occurrence, node access is guaranteed to proceed from left to right in the dynamic array. Figure 3.6 illustrates the structural difference between a standard, compact, and array BST.

The main disadvantage of the array BST — including Knuth's representation — is the overhead of deletion. Deleting a node involves one of three steps. First, the candidate node has no children, and thus, can be simply deleted. Second, the candidate node has a single child, and is simply replaced by its child. Third, the candidate node has two children. In this case, the candidate node is replaced by its left-most child from its right subtree. In a standard and compact chained BST, these steps require tree traversals and pointer manipulations. In an array BST, deleting a string involves both pointer manipulation and array copying; to delete a string, the array must be resized. However, resizing is cache-efficient, which is likely to compensate for the high computational costs involved. Hence, updating an array BST is expected to be as efficient as updating a chained BST, as we demonstrate in the next chapter.

3.7.1 Expected cache costs

We theoretically analyze the cache performance of the standard, compact, and array BSTs, to gain a better understanding of the expected reductions in cache misses. We simplify our calculations by assuming the following:

- Use of an ideal cache, as described in Section 3.4.
- The BSTs are many times larger than cache.
- A uniform data distribution, that is, there is no skew.
- Strings are of uniform length and are randomly generated; each character is randomly selected from the English alphabet. There are no string duplicates.

The binary search tree has a logarithmic access bound on average, but linear at worst [Knuth, 1998]. Each node is $3W$ bytes long, so traversing a standard chain BST can incur up to $3WN/C$ cache misses. Every node accessed results in a string pointer traversal, which can attract a further cache-miss. Hence, the expected L2 cache misses for searching a standard BST is up to:

$$\begin{aligned} & \frac{3WN}{C} + \frac{LN}{C} \\ = \quad & \frac{N\left(3W + L\right)}{C} \end{aligned} \tag{3.19}$$

Similarly, the number of TLB misses is up to:

$$\begin{aligned} & \frac{3WN}{P} + \frac{LN}{P} \\ = \quad & \frac{N\left(3W + L\right)}{P} \end{aligned} \tag{3.20}$$

The compact chain BST eliminates string pointers, allowing nodes to be represented by the strings themselves. Each string is preceded by $2W$ bytes. Hence the expected L2 cache misses for searching a compact BST is up to:

$$\frac{2WLN}{C} \tag{3.21}$$

Similarly, the number of expected TLB misses is up to:

$$\frac{2WLN}{P} \tag{3.22}$$

In theory, the array BST has the same cache-complexity as the compact BST. In practice, however, we expect further reductions in cache misses because nodes are contiguously allocated. Furthermore, the array is traversed from left-to-right, which is anticipated to make good use of hardware prefetchers.

Inserting a string into a standard or compact BST incurs the same cache costs as traversal. The small and constant overhead of creating a new node can be ignored. This is also the case for an array BST, that is, until the array housing the entire structure needs resizing, in which case the array BST can attract further cache misses, due to array copying. However, we assume that the array is grown with *paging*, hence, resizing is infrequent. The expected L2 cache misses for inserting a string into an array BST is up to:

$$\frac{2WLN}{C} + \frac{2WL(N+1)}{C} \tag{3.23}$$

To delete a string in a BST, we consider the most complex situation — where the candidate node is replaced by the smallest node in the its subtree. First, the BST needs to be searched to locate the node to delete, which will incur the traversal costs described above. Once found, the traversal continues down the subtree of the candidate node. We assume that the subtree has X nodes and a worst-case complexity. For brevity, we only report the L2 cache misses below. The expected L2 cache misses to delete a string in a standard BST is up to:

$$\frac{N\,(3W+L)}{C} + \frac{X\,(3W+L)}{C} \tag{3.24}$$

In a compact BST, the L2 costs are reduced:

$$\frac{N\,(2W+L)}{C} + \frac{X\,(2W+L)}{C} \tag{3.25}$$

In an array BST, there is the added cost of resizing the array:

$$\frac{N\,(2W+L)}{C} + \frac{X\,(2W+L)}{C} + \frac{2WL(N-1)}{C} \tag{3.26}$$

3.8 Space saved by eliminating pointers

We continue our discussion of Section 2.5.1, by considering the space saved when applying our compact and array-based techniques to the standard hash table, burst trie, and BST, which we show in Table 3.7. Every pointer eliminated saves twelve bytes of memory: 4 bytes for the pointer and 8 bytes of allocation overhead.

Hash tables consume memory in several ways: space allocated for the strings and for pointers; space allocated for slots; and overhead due to operating system overheads and space fragmentation. 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 $4m + 24s$ bytes, where m is the number of slots and s is the number of strings inserted. In a compact chain, string pointers are eliminated, reducing the overhead to $4m + 12s$ bytes. In a clustered chain, the 8-byte allocation overhead per node is eliminated, but each slot incurs an initial 8-byte overhead for allocating of an array of nodes. Hence, the space overhead of a clustered hash table is up to $12m + 16s$ bytes. In an implicit chain, the next-node pointers are eliminated, but with the requirement of allocating a null 4-byte pointer at the end of each chain, to serve as a delimiter. Hence, the space overhead of an implicit hash table is up to $16m + 12s$ bytes. The memory consumed by the array hash table is slightly more complicated to model. First consider exact-fit. Apart from slot pointers, all other pointers are eliminated. The space overhead is then notionally $4m$ bytes plus 8 bytes per allocated array — that is, up to $12m$ bytes in total — but the use of copying means that there is an 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 127 characters in length, an additional byte is required.

For paging, we 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 $12m + B/2$ bytes. When the load average is low — that is, $s < m$ — most slots are either empty, at a cost of $4m$ 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 $4m + s(B - l + 8)$ bytes.

For the standard binary search tree, each string stored has three pointers and issues two *malloc* system calls. The space overhead is therefore $28s$. In a clustered BST, the 8-byte allocation overhead is eliminated for every node. However, nodes are clustered into fixed-sized blocks, of which, incur an allocation overhead. Thus, the space overhead of a clustered BST is $20s+8(20s/C)$. In a compact BST, the space overhead is $16s$ bytes and with an array, it is reduced to $8s+8$ bytes which is smaller than the standard BST by a factor of more than three. We do not consider the space requirements of an implicit BST, because eliminating child-node pointers can compromise performance, as noted in Section 3.7.

For the standard burst trie, we assume that t and b are the number of trie nodes and containers,

	Standard	Clustered	Compact	Array
BST	$28s$	$20s + 8(20s/C)$	$16s$	$8 + 8s$
Burst trie	$24\bar{s} + 520t + 13b$	$16\bar{s} + 520t + 21b$	$12\bar{s} + 520t + 13b$	$520t + 10b$
Hash (exact-fit)	$4m + 24s$	$12m + 16s$	$4m + 12s$	$12m$
Hash (paging)	—	—	—	$4m + s(B - l + 8)$

Table 3.7: A comparison of the space overhead of the hash table, burst trie, and BST, using standard chains, clustered chains, compact chains, and dynamic arrays. The number (s) of strings stored, the number ($\bar{s}$) of strings stored in containers, the number (m) of slots used by the hash table, the number (t) of tries, and the number (b) of containers used by the burst trie or the number of slots used the hash table. The block size (B) represents the page-size used to grow a dynamic array, and the block size (C) represents the block size used to cluster nodes.

respectively. A trie node of 128 pointers requires 512 bytes and a call to *malloc*. We store trie nodes contiguously in arrays to eliminate allocation overheads and to improve TLB efficiency. However, in the measurements shown in Table 3.7, we assume that trie nodes are individually allocated; hence each trie node occupies 520 bytes of space.

A container in a standard or compact burst trie contains a 4-byte pointer to the start of its linked list, and reserves a byte for housekeeping. The total overhead for the standard burst trie is therefore $24\bar{s}+520t+13b$, where $\bar{s}$ is the number of strings stored in containers. In a clustered burst trie, the 8-byte allocation overhead incurred by each node in a container is eliminated, except for the initial 8 bytes when allocating the array of nodes. Hence, the space required by the clustered burst trie is $16\bar{s} + 520t + 21b$. In an implicit burst trie, the next-node pointers are eliminated, but with the requirement of allocating a 4-byte pointer at the end of each container, to serve as a list delimiter. Hence, the space overhead for the implicit burst trie is $12\bar{s} + 520t + 25b$. In a compact burst trie, string pointers are eliminated, resulting in an overhead of $12\bar{s} + 520t + 13b$, while, for the array burst trie, the space overhead is reduced to only $520t + 10b$, at little to no cost in performance.

3.9 Summary

The chaining hash table, burst trie, and BST, are currently the fastest and most compact data structures available for managing strings in-memory [Heinz et al., 2002; Williams et al., 2001; Zobel et al., 2001]. They are however, oblivious to cache and can, as a consequence, attract excessive cache misses on current cache-oriented processors, which will dominate their performance. The principle cause of their efficiency are the use of pointers, which can cause random access to memory

and can hinder the effectiveness of hardware prefetchers. However, there has currently been no attempt in literature at eliminating the use of pointers in these data structures, to develop cache-conscious alternatives. A cache-conscious data structure has high locality of memory access, thereby exploiting system cache and making its behavior more predictable. Clustering is arguably the best technique at making data structures cache-conscious, however, it does not eliminate pointers; it instead ensures that a chain is stored contiguously in static block of memory, to improve spatial access locality [Chilimbi, 1999].

In this chapter, we re-evaluated the design of the current best string data structures by exploring two pointer elimination techniques, the *compact chain* and *dynamic array*. We showed how to re-design the hash table, burst trie, and BST for strings, to develop cache-conscious alternatives that can exploit cache without compromising dynamic characteristics. We described the steps to insert, search, and delete strings, and provided a theoretical analysis of the expected gains in performance over chaining. In addition, our cache-conscious variants save a considerable amount of space, at little to no cost, which is valuable to many tasks, such as vocabulary accumulation, as it delays the exhaustion of main memory.

In the next chapter, we implement our cache-conscious hash tables, burst tries, and BSTs, and experimentally evaluate their performance in comparison to a baseline of well-known string data structures, using large sets of strings with varying characteristics.

Chapter 4

Cache-conscious data structures in practice

We have reviewed principles on which cache-aware algorithms should proceed for the storage and retrieval of strings in memory. In this chapter, we apply these principles in practice, to present experimental evaluations that compare the time and space required by the cache-conscious hash table, burst trie, and BST — described in Chapter 3 — against a baseline of existing high-performance string data structures. We demonstrate the value of exploiting cache on existing string data structures that are otherwise efficient for the task of storing and retrieving strings, in memory.

4.1 Experimental design

To evaluate the efficiency of the hash table, burst trie, and BST using their standard, clustered, compact, and array representations — discussed in the previous chapter — we compare the time required for construction and search, as well as the amount of memory consumed, against other string data structures: the standard hash table, burst trie, BST, splay tree, red-black tree, TST, the adaptive trie [Acharya et al., 1999], and the Judy data structure; we used the *Judy-SL* variant which is designed for string keys [Hewlett-Packard, 2001]. The elapsed or total time required by these data structures was averaged over a sequence of ten runs. After each run, main memory was flooded with random data in order to flush system caches.

Measurements of space include an estimate of the overhead imposed by the operating system, which in our case was 8 bytes per system call; we compared our measure of space with total memory reported by the operating system under the `/proc/stat/` table and found it to be consistent.

We used PAPI [Dongarra et al., 2001] to measure the actual number of instructions, TLB, and L2 cache misses of search (we did not count L1 misses as they have only a small performance

Dataset	Distinct strings	String occs	Average length	Volume (MB) of distinct	Volume (MB) total
DISTINCT	28772169	28772169	9.59	304.56	304.56
TREC	612219	177999203	5.06	5.68	1079.46
URLS	1289459	9999425	30.93	45.93	318.87

Table 4.1: Characteristics of the datasets used in our experiments.

penalty). We do not report the cache-performance of construction, as the results were similar to those for search. In addition, experiments involving deletion were omitted, as the results were found to be similar to those for construction.

The datasets used for our experiments are shown in Table 4.1. 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. When the same dataset is used for both construction and search, the process is called a *self-search*.

The TREC dataset is the complete set of word occurrences, with duplicates, in the first of the five TREC CDs [Harman, 1995]. This dataset is highly skew and contains only a small set of distinct strings. The DISTINCT dataset contains almost twenty-nine 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 URLS dataset, extracted from the TREC web data, is composed of non-distinct complete URLs.

To measure the impact of load factor on the hash tables, we varied the number of slots used. We commenced with 2^{15} slots, which we doubled to 2^{27} or until a minimum execution time was observed. For the standard and compact burst tries, we varied the container threshold (the number of strings a container needs to trigger a burst) by intervals of 10 from 30 to 100. For the array burst trie, we extended the sequence to include 128, 256, and 512 strings. Both compact and standard chaining methods are most efficient when coupled with move-to-front on access, as reported by Zobel et al. [2001]. We enabled move-to-front for the chained hash tables and burst tries, but disabled it for the array-based data structures, a decision which we justify in later discussions.

The primary machine used for our experiments was a 2.8 GHz Pentium IV. We conducted experiments on other machine architectures, namely a 3 GHz Intel Xeon processor and a 700 MHz Intel Pentium III processor. The Xeon processor showed similar performance to the Pentium IV and was thus omitted. We include results from the Pentium III processor, however, as it had no hardware prefetch mechanism — we were unable to disable hardware prefetch on our primary machine. We also conducted experiments on a Sun Ultra Spark server running a Solaris operating system, the results of which are provided in the Appendix. We found the results to be similar to those discussed below, highlighting that our techniques remain successful at reduce access costs

	Intel Pentium IV	Intel Pentium III
CPU speed	2.8 GHz	700 MHz
No. CPUs	1	4
L1/L2 size (KB)	8/512	16/1024
L1/L2 cache-line (B)	64/128	32/64
TLB entries	64	32
Main memory (MB)	2048	2048
Page size (KB)	4096	4096
Avg. mem. latency	95 ns	160 ns
Linux kernel	2.6.12	2.6.12
Hardware prefetch	yes	no

Table 4.2: Characteristics of the machines used in our experiments. The Pentium IV (highlighted) was our primary machine.

on alternative computing architectures. Some of the characteristics of the Intel based machines are summarized in Table 4.2.

Our experiments were conducted on a Linux operating system under light load. We are confident — after extensive profiling — that our implementations are of high quality. Our data structures were implemented in C and compiled using *gcc* version 4.1.1, with all optimizations enabled. We found that the bitwise hash function [Ramakrishna and Zobel, 1997] with a mask instead of a modulo, was a near-insignificant component of the total costs of hashing. Williams et al. [2001] reported the inefficiency of using the default string compare (`strcmp`) library routine provided by the Linux operating system, and showed that their own implementation achieved speed gains of up to 20%. We do the same for our implementations.

4.2 Skew data

A typical use for string data structures 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 some strings are much more common than others. Figure 4.1 and Figure 4.2 show the relationship between the time and memory requirements of our data structures, for construction and self-search. The dataset used in these experiments was TREC.

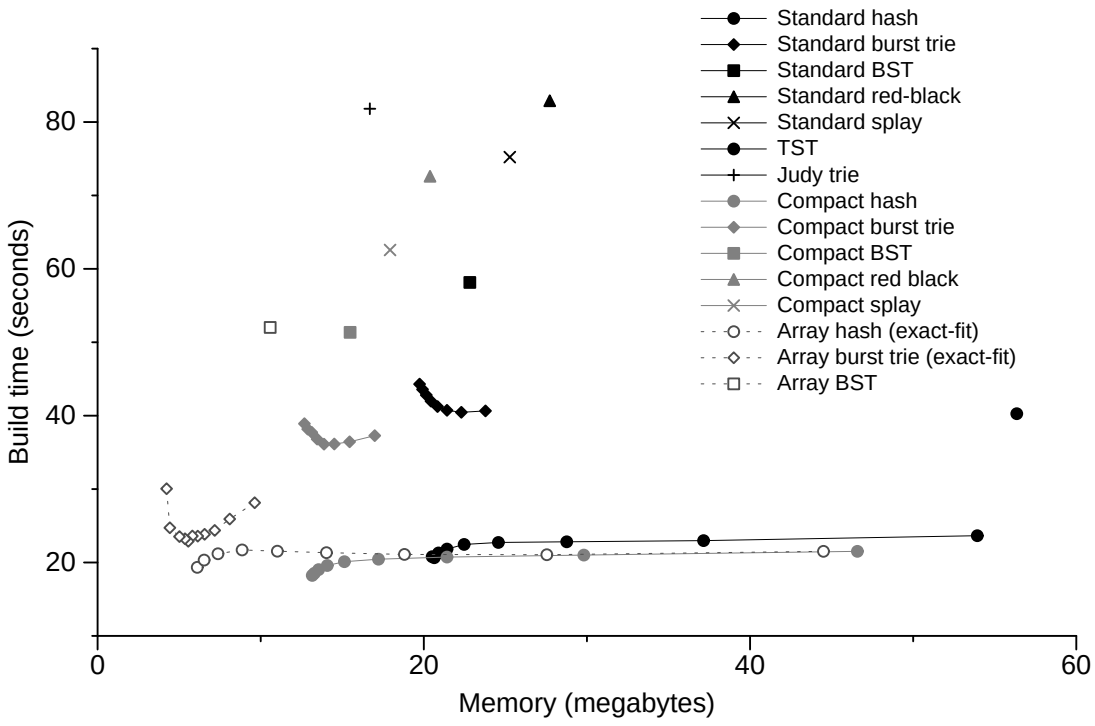


Figure 4.1: The time and space required to build the data structures using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence is extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*.*

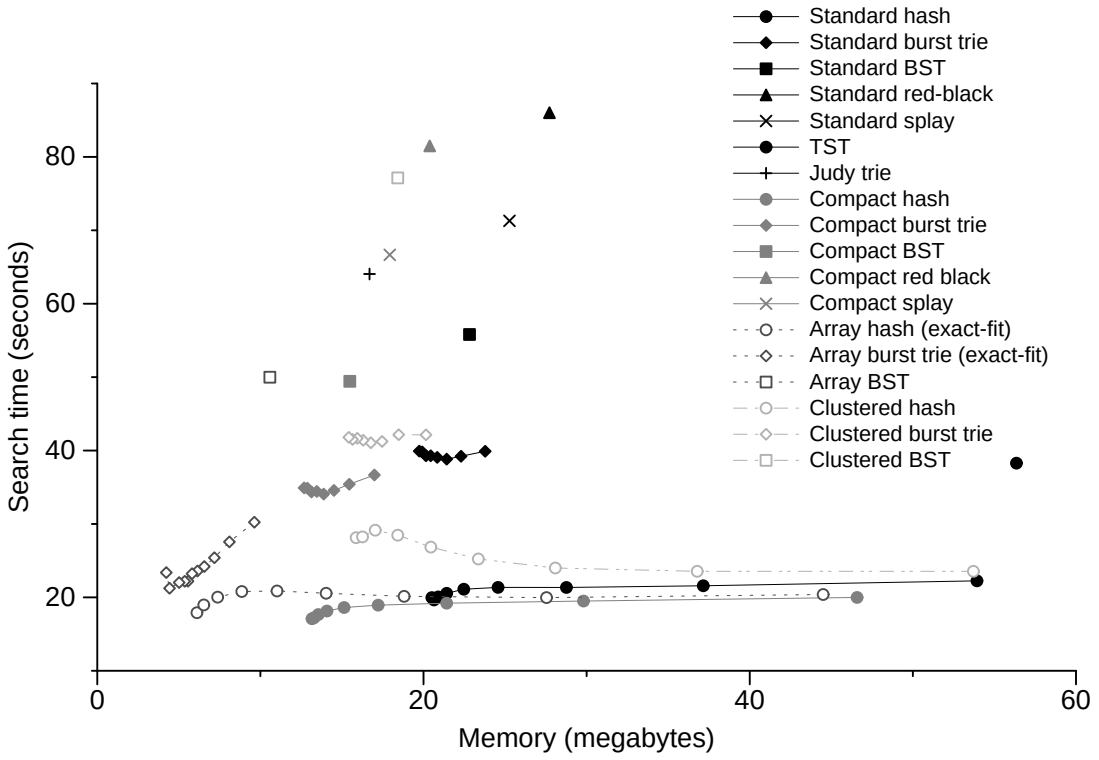


Figure 4.2: The time and space required to self-search the data structures using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence is extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*.*

4.2.1 Standard-chain data structures

The Judy data structure required the least amount of space of all standard-chained data structures, but was almost the slowest to access under skew, being only slightly faster than the red-black tree. As reported by Williams et al. [2001], the standard BST was the fastest tree to construct and self-search under skew. The red-black and splay trees were inefficient, due to the maintenance of a balanced and self-adjusting tree structure, respectively. The L2, TLB, and instruction costs for self-searching Judy and the variants of BST are shown in Figure 4.5. There is a strong correlation with the number of cache misses incurred and the overall search time. For example, the red-black tree incurred the highest L2 and TLB misses and was therefore the slowest to access. Although Judy executed fewer instructions than the standard BST, it incurred more cache misses which caused its poor performance.

As reported by Heinz et al. [2002], the standard hash table and burst trie were the fastest standard-chained data structures to build and search. Although the standard burst trie required almost twice as much time to build and self-search than the standard hash table, it was competitive with regards to space, and maintained sorted access to containers. The TST required the most space but could match the construction and self-search speeds of the standard burst trie, as result of its relatively low L2 and TLB misses (Figure 4.5).

4.2.2 Clustered-chain data structures

We compare the self-search performance of the three fastest standard-chain data structures — the hash table, burst trie, and BST — with clustering [Chilimbi, 1999]. Clustering stores the nodes of a linked list in a contiguous block of memory, to improve spatial access locality without eliminating pointers, as discussed in Chapter 3. The time and space required to self-search a clustered hash table, burst trie, and BST, are shown in Figure 4.2.

Our results show that, although the clustered data structures require less space than their standard representations — due to the elimination of the 8-byte allocation overhead per node — the clustered hash table, burst trie, and BST were, in all cases, slower to access than their standard representations. Consider the memory access patterns of traversing a clustered chain from the slot of a hash table or within a container of a burst trie. With move-to-front on access, the first node in the block may not necessarily be the first node accessed. As a result, searching a clustered list can lead to random memory accesses resulting in poor use of cache. When the clustered list spans multiple cache lines, these random accesses are likely to become more expensive. This is reflected in the self-search costs of Figure 4.2 and the cache costs of Figure 4.5. For example, as the average load factor increases, the clustered hash table incurs more cache misses than the equivalent standard hash table.

If we disable move-to-front on access in a clustered list, the cost of access increases even further, as now the list must be traversed, in order of allocation, until the desired node is found. Although

the number of random accesses is reduced, the number of pointer de-references increases, which can further hinder the effectiveness of hardware prefetchers. For clustering to be effective under skew access, it is necessary to both reduce random accesses and the number of pointer de-references, by physically moving nodes to the start of the list. However, this approach can substantially increase the computational cost of search — particularly for large lists — which can outweigh the benefits (which we confirmed through preliminary trials on the clustered hash table and burst trie).

Traversing a standard list also incurs random memory accesses. However, nodes can be allocated anywhere in memory, which gives the operating system some flexibility at improving access locality. That is, the operating system can dynamically group frequently accessed pages in memory to improve cache utilization; it cannot do this as effectively once the programmer enforces contiguous storage of nodes that are accessed via pointers.

4.2.3 Compact-chain and array-based data structures

We implement the compact-chain and dynamic array representations for the hash table, burst trie, and BST, as described in Chapter 3, and include a compact-chain representation of the red-black and splay tree. The time and space required to construct and self-search these data structures are presented in Figure 4.1 and Figure 4.2.

Hash tables

The compact-chain hash table was the fastest data structure to construct under skew access. The compact hash achieved its best construction time of 18.2 seconds and a self-search time of 17.1 seconds, using 2^{15} slots and 13.1 MB of memory; a space overhead of about 98 bits per string (only 612,219 strings are distinct in the TREC dataset, which require a total of 5.6 MB). The equivalent array hash was marginally slower to construct and self-search, requiring under 19.2 and 17.9 seconds, respectively. Despite its high cache-efficiency, the array hash required more time as a result of executing more instructions (Figure 4.5), caused by the absence of move-to-front during search, and array resizing during construction.

The array hash, however, was considerably smaller in size, requiring only 6.1 MB of memory; a space overhead of about 6 bits per string. This efficiency is achieved despite a load average of 37 strings per slot. Increasing the number of slots (reducing the load average) had no positive 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. The cache performance of the compact and array hash tables are shown in Figure 4.5. In all cases, the compact and array hash tables showed consistent and simultaneous reductions in L2 and TLB misses over the standard hash table.

The standard hash table was markedly inferior in both time and space, achieving its best construction and self-search time of 20.6 and 19.6 seconds, respectively, using 2^{16} slots and 20.6 MB of memory; a space overhead of about 195 bits per string. Given that the standard hash table was the fastest-known data structure to construct and self-search under skew access [Zobel et al., 2001], we have strong evidence that our new structures represent a significant improvement.

BSTs

Compared to the performance of the standard hash table and burst trie, the standard BST was markedly inferior. However, this was not the result of poor cache utilization. Figure 4.5 shows that the standard BST is more cache-efficient — in both L2 and TLB — than the standard burst trie. Although unbalanced, frequently accessed nodes are likely to be located near the root of the tree. Furthermore, with no structural modifications during search, frequently accessed tree paths are likely to persist longer in cache. However, the standard BST was computationally expensive to self-search — executing almost 400 instructions per search, which is almost three times more than the burst trie (Figure 4.5).

The compact BST further reduced L2 and TLB misses, which made it faster to build and self-search than the standard BST, but remained computationally expensive relative to the standard burst trie. The array BST showed even further reductions in L2 and TLB misses, and approached the cache-efficiency of the array hash table and array burst trie. However, having executed more instructions, the array BST was marginally slower to access than the compact BST. Nonetheless, both the array and compact BSTs displayed consistent improvement over the standard BST, which is currently the fastest tree structure under skew access, with size falling from 23 MB to 15 MB for the compact BST, to about 10 MB for the array BST, at no cost.

Burst tries

The array burst trie showed strong improvements in both time and space over its compact and standard chained variants. These improvements are consistent for both construction and self-search. Comparing the cache performance of the array burst trie with its chained variants (Figure 4.5), we see a sharp decline in L2 and TLB misses as the container threshold or capacity increases from 30 to 512 strings.

At its best, the array burst trie with a container threshold of 256 strings, required 21.2 seconds to self-search and 4.4 MB of space. This is about 1.6 MB smaller than the array hash, which is less space than required by the strings alone. The compact burst trie at its best, required 34.1 seconds to self-search with a container threshold of 60 strings and 13.7 MB of space; an overhead of about 104 bits per string. The standard burst trie was markedly inferior, requiring 39.1 seconds to self-search and 20.6 MB of space; an overhead of about 195 bits per string. Considering that the standard burst trie was reported to be the most efficient data structure for the task of vocabulary

accumulation [Heinz et al., 2002], our array burst trie, being up to 46% faster while imposing no space overhead per string, is a substantial advance.

Use of large containers in the array burst trie — containers storing over 256 strings for example — are preferable, because they can be both cache- and space -efficient. Large containers, however, are computationally expensive to access. As the size of containers increase from 128 to 512 strings, for example, Figure 4.5 shows a sharp increase in instructions per search, caused by the absence of move-to-front on access. The same behavior was observed during construction, with large containers being cache-efficient but incurring high numbers of instructions per search, due to the absence of move-to-front and array resizing.

The high computational cost of accessing large containers could not be entirely masked by the reductions in cache misses, and, as a consequence, the array burst trie became more expensive to build and search, relative to its smaller containers. For example, when the container size increased from 256 to 512 strings in Figure 4.1, the time required to build increased by about 17% (from 25 to 30 seconds). However, when compared to a standard burst trie with a container threshold of 512 strings, the array burst trie was up to 66% faster (or around 58 seconds) to build and self-search, due to its high reduction in cache misses. These results show that, in order to sustain a fast and scalable burst trie, it is necessary to reduce cache misses while sustaining low numbers of instructions, relative to chaining. We address this issue in greater detail in the next chapter.

4.2.4 Paging vs. exact-fit array growth

We compare the difference in time and space for building the array hash and array burst trie, using the exact-fit and 64-byte paging techniques described in Chapter 3. The results are shown in Figure 4.3. The exact-fit model — where the array is resized on every insertion — is space-efficient but can be more expensive to build (as a result of excessive copying), whereas paging permits faster construction at the expense of maintaining some unused space. The relationship with speed however, is more complex, with paging faster in some cases, but degrading relative to exact-fit, due to a more rapid increase in space consumption. The choice of array growth policy had negligible impact on the performance of search. The situations where paging would likely yield stronger improvements is with large numbers of insertions, which we consider in later experiments.

4.2.5 Effectiveness of move-to-front on arrays

Despite its high memory usage, the compact hash table performed well under skewed access, partly due to the use of move-to-front. With dynamic arrays, move-to-front on access is computationally expensive because strings must be copied. Figure 4.4 compares the self-search costs with and without move-to-front on access, and includes comparable figures for compact and standard chaining (both with move-to-front). The use of move-to-front reduces the speed of the array hash; even though the vast majority of searches terminate with the first string (so there is no string

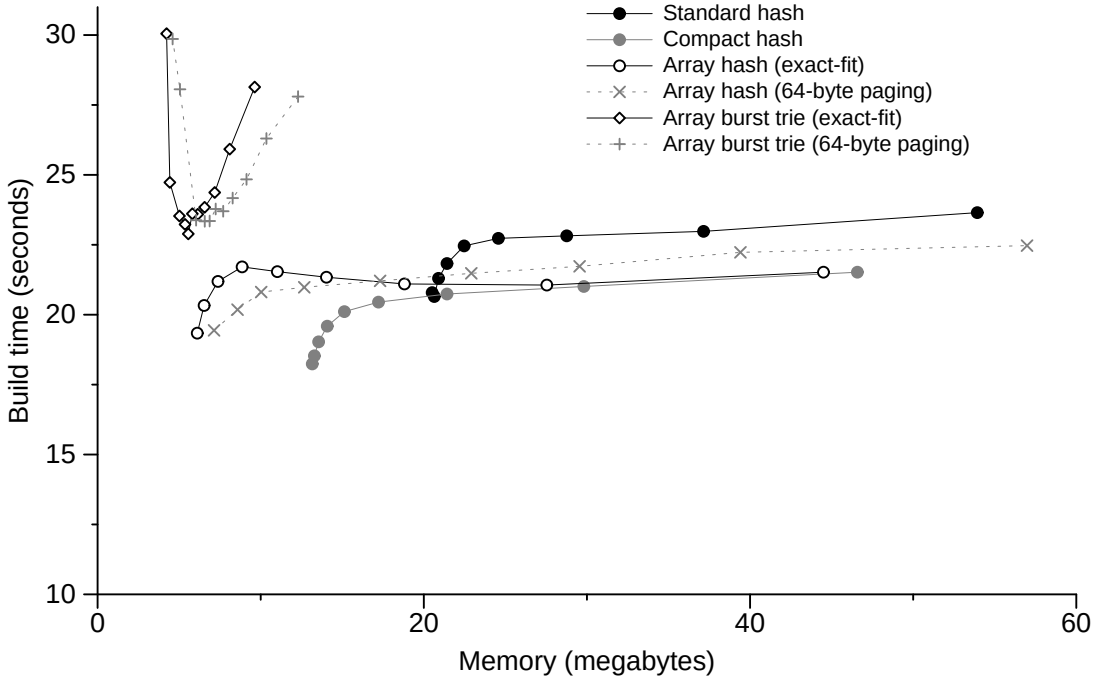


Figure 4.3: A magnified version of Figure 4.1 showing the time and space required to build the array hash and array burst trie, using the TREC *dataset with and without paging. Paging grows an array in 64-byte chunks, in contrast to exact-fit. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory.*

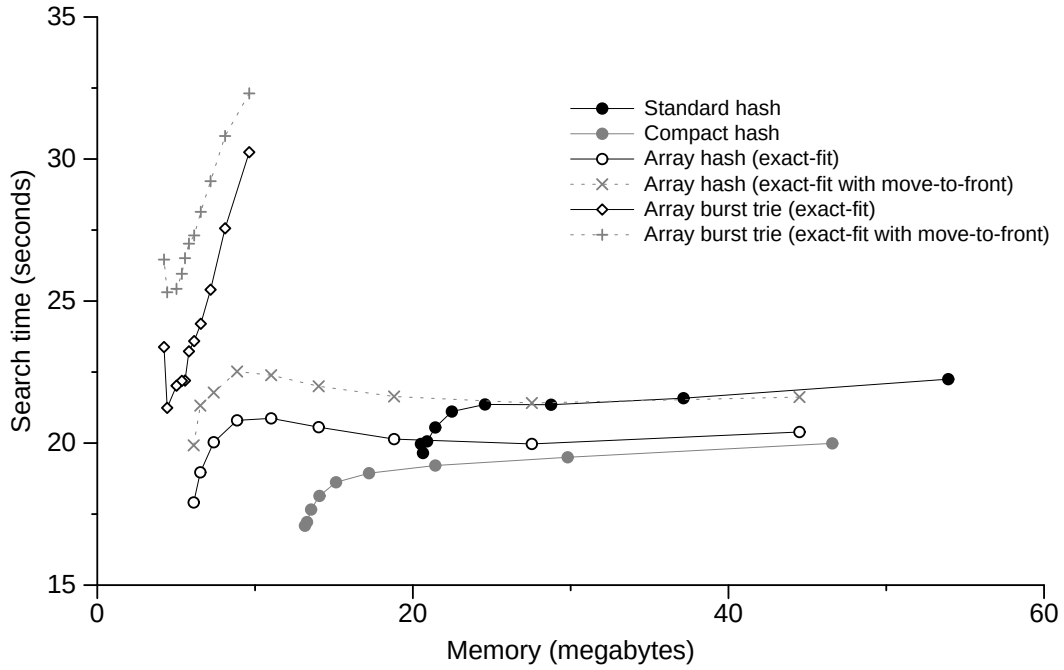
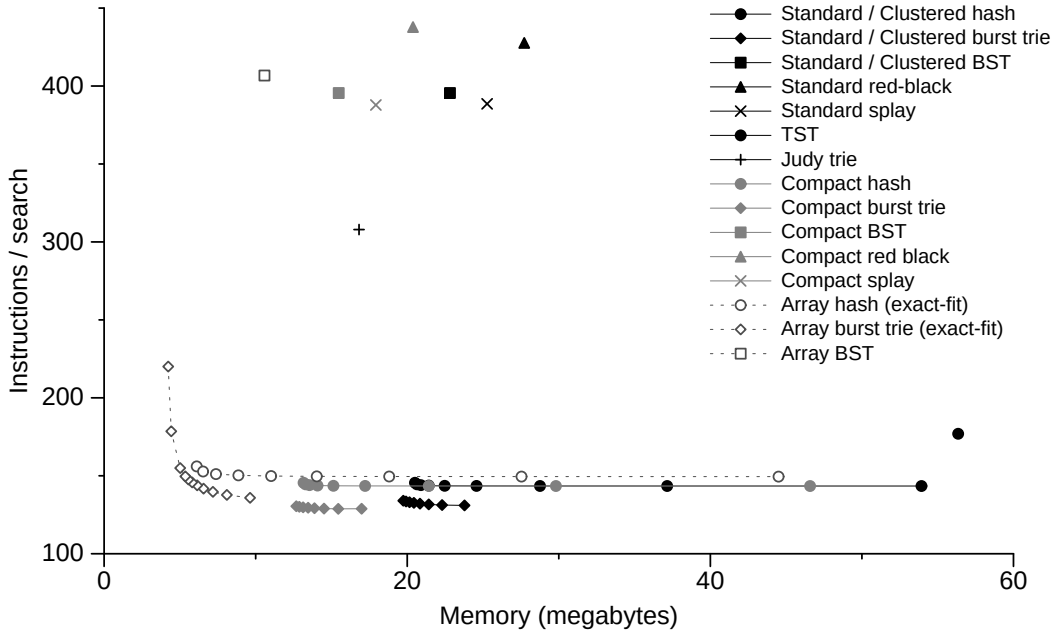
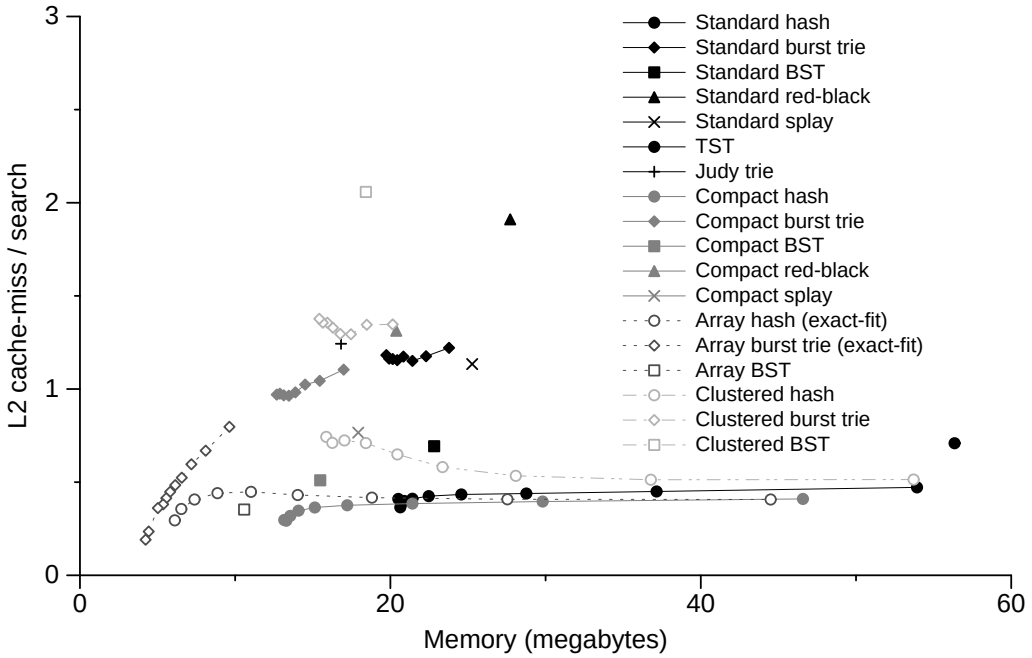


Figure 4.4: A magnified version of Figure 4.2 showing the time and space required to self-search the array hash and array burst trie, using the TREC *dataset with and without move-to-front on access. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory.*

(a) Instructions per search



(b) L2 cache-miss per search



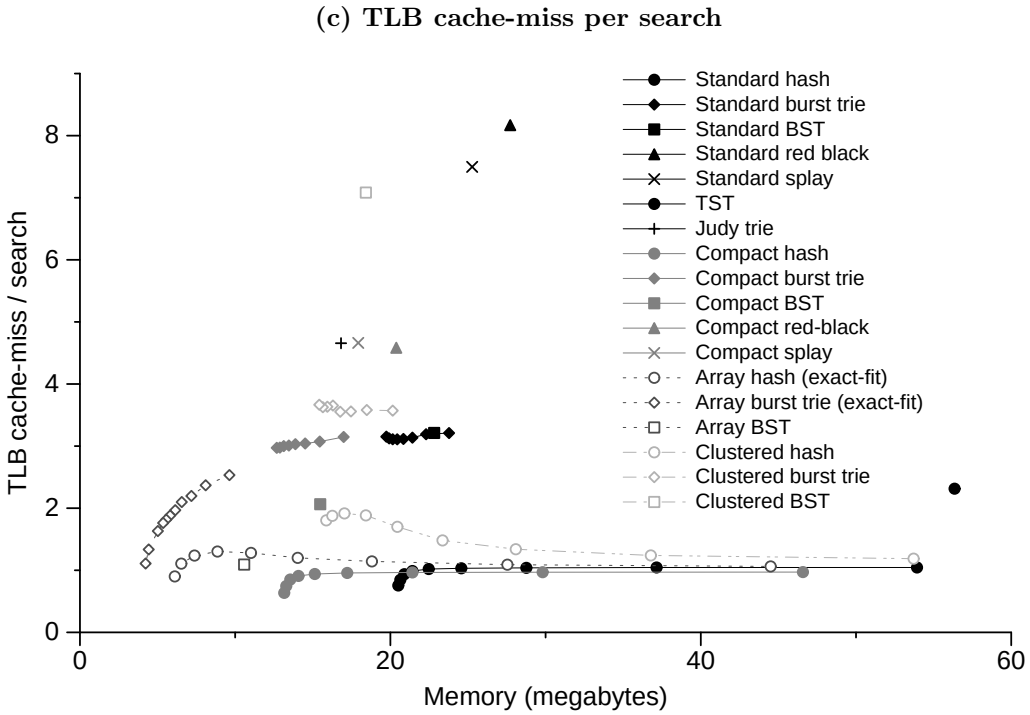


Figure 4.5: The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*.*

movement), the cases that do require a movement are costly.

Performing a move-to-front after every kth successful search might be more appropriate. Alternatively, the matching string can be interchanged with the preceding string, a technique proposed by McCabe [1965]. However, preliminary trials revealed that these techniques were not particularly effective at reducing the high computational costs of move-to-front. Hence, we believe that move-to-front is unnecessary for the array-based data structures, as the potential gains seem likely to be low.

With move-to-front disabled, however, frequently accessed strings are likely to incur more instructions and cache misses — in particular TLB misses — which is reflected in the instruction and TLB costs of the array hash (Figure 4.5). As a consequence, the array hash was slightly slower to build and self-search than the compact hash table. We also enabled move-to-front in the containers of the array burst trie and observed similar results. With move-to-front enabled, the array burst trie was consistently slower to access. In contrast to the array hash, however, the array burst trie remained faster than its chained variants, even without move-to-front, due to the use of bounded-size containers and a trie structure that removes shared prefixes — which can reduce both instruction and cache costs during search.

4.2.6 Skew search on large data structures

Our previous experiments involved data structures that were small in size, containing only 612,219 distinct strings. In this section, we repeat the previous skew search experiment but on much larger data structures that contained almost 29 million distinct strings. In this case, fewer frequently accessed nodes and strings are likely to reside in cache, which can impact performance. The data structures were built using the DISTINCT dataset, and then searched using the TREC dataset. The time and space required to search is shown in Figure 4.6.

The red-black tree and Judy were the slowest data structures to access. The splay tree, however, showed considerable improvement over the red-black tree, which demonstrates that splaying is more effective under skew access than a balanced tree structure. Both the splay tree and the red-black tree, however, were space-intensive. As observed in previous experiments, the standard BST was the fastest tree to access, despite having increased in size by a factor of more than 48 — from 22.8 MB to over 1110 MB. The TST consumed almost 2.5 GB of space and thus exhausted main memory. As a consequence, virtual memory was accessed which resulted in a total search time of almost 1200 seconds.

At its best, the array hash required about 27.3 seconds to search using 2^{17} slots and only 306.2 MB of space. This is a space overhead of less than a bit per string. The equivalent compact and standard hash tables were up to 24% slower, requiring 32.6 and 36.1 seconds, respectively. Furthermore, the standard hash table consumed 995.6 MB of memory, a space overhead of about 192 bits per string. The equivalent compact hash was more efficient, requiring 650.3 MB or about 96 bits per string. With 2^{17} slots, the average load factor was approximately 220 strings per slot.

At such a high load, the chaining hash tables were too expensive to access, as they incurred high L2 and TLB misses that masked the benefits of move-to-front. Under heavy load, the cache-efficient array hash was therefore the fastest data structure, despite having executed more instructions due to absence of move-to-front.

Given enough space, the chained hash tables — as a result of move-to-front — could rival the cache-efficiency of the array hash, while executing fewer instructions. As a result, the chaining hash tables can become slightly faster to access than the array hash. As observed in previous experiments, however, increasing the number of slots had no positive impact on the array hash.

At their best — using 2^{22} slots — the compact and standard hash tables were up to 24% faster than the array hash, requiring only 20.8 and 22.9 seconds to search, respectively. However, in order to achieve this speed (a difference of only 6.5 seconds from the array hash), the compact and standard hash table required two to three times the space of the array hash, respectively. For common string processing tasks such as vocabulary accumulation, such an increase in space consumption will often be unacceptable, as it implies that fewer strings can be managed in-memory.

The compact and array BSTs were faster to access than the standard BST, while saving a substantial amount of space. The clustered BST, however, required almost twice the time of the standard BST. The standard burst trie was faster than all variants of BST, but remained almost twice as slow as the standard hash table, as observed in previous experiments. The clustered burst trie was the slowest burst trie to access, while the compact burst trie showed consistent gains in both time and space over the standard burst trie. The array burst trie displayed the strongest improvements, being the most space-efficient data structure — imposing no space overhead per string — while approaching the performance of the array hash table. For example, the array burst trie was up to 38% faster than its chained variants, and required at best, only 8 seconds more than the array hash while maintaining sorted access to containers.

4.2.7 Performance without hardware prefetch

We repeat the previous skew search experiment on the Pentium III processor, to observe the value of eliminating pointers on a slower machine with no hardware prefetch. In this experiment, we considered the three best data structures — the hash table, burst trie, and BST — using their standard, compact, and array-based representations. Figure 4.7 shows the results.

Despite the absence of hardware prefetch, the results were consistent with previous experiments. The compact BST and array BST remained faster than the standard BST while saving a considerable amount of space. The standard burst trie was faster than all three BSTs, but remained almost twice as slow as the standard hash table. The compact burst trie showed only marginal improvements in time, whereas the array burst trie showed the strongest gains.

Similarly, the array hash table was slightly slower than the compact and standard hash tables, but saved a considerable amount of space. The lack of hardware prefetch, however, meant that large arrays would incur more cache misses when traversed. As a consequence, the array hash

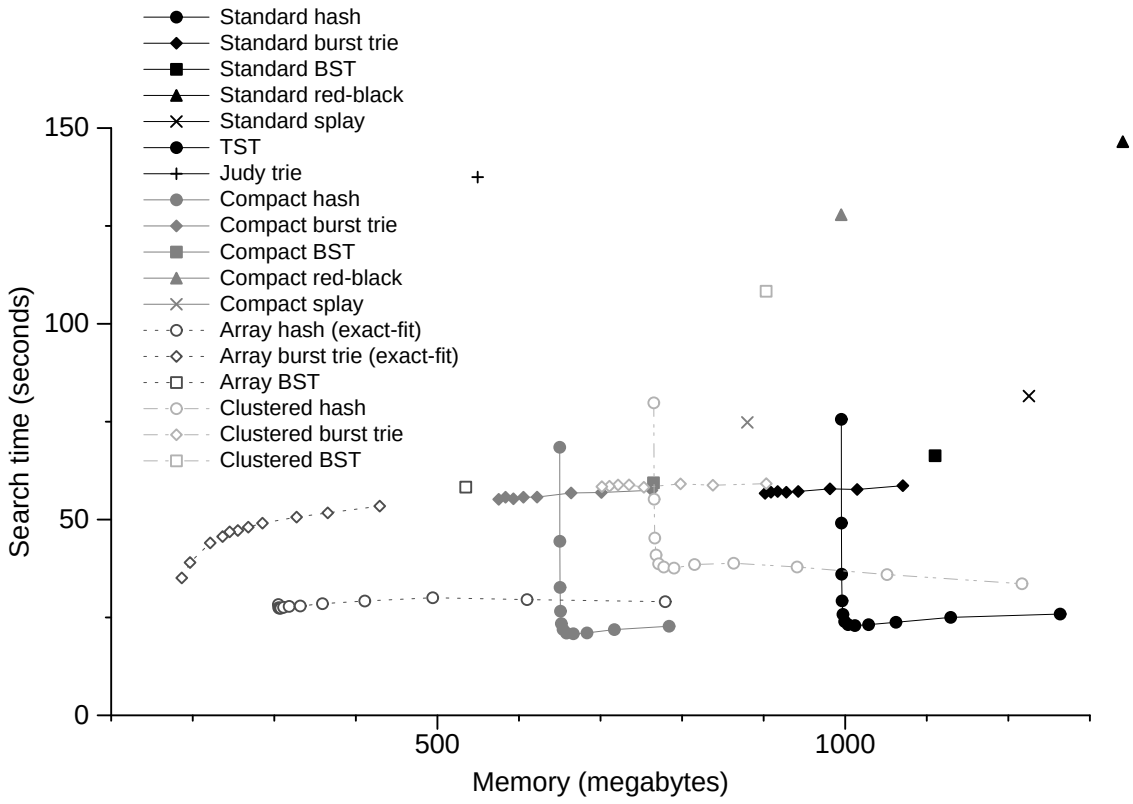


Figure 4.6: A skew search using the TREC *dataset, on data structures that were built using the* DISTINCT *dataset. The TST is not shown as it required over 2497 MB of memory and about 1194 seconds to search. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128, 256, *and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{26}*.*

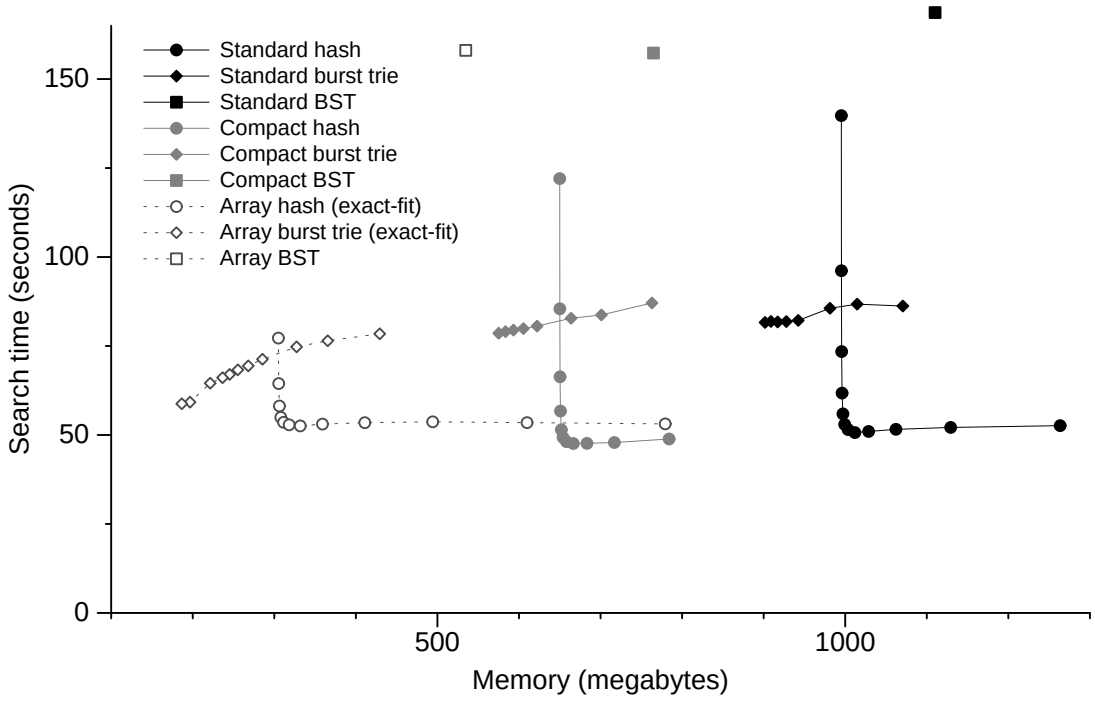


Figure 4.7: A skew search using the TREC *dataset on data structures that were constructed using the* DISTINCT *dataset on the Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{26}*.*

required more time to search when under heavy load, compared to its previous performance in Figure 4.6. The increase in time can also be attributed to the use of a slower processor and a smaller cache-line size, which can increase the cache-miss rate as discussed in Chapter 2. Nonetheless, the array hash remained competitive to the chained hash tables, which were only faster once given enough space. These results demonstrate that the compact and array-based representations of the hash table, burst trie, and BST, can yield strong gains in performance on older machines where instructions are more expensive to execute, and with no hardware prefetch.

4.3 URL data

Our next experiments used the URLS dataset, a dataset 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 and self-search results are shown in Figure 4.8 and Figure 4.9.

4.3.1 Hash tables

The hash tables were the fastest data structures to build and search, but the optimum number of slots was much larger than that required during the TREC experiments, with the best load average being less than 1. To achieve its best time of 4.3 and 4.0 seconds for construction and self-search, respectively, the standard hash table required 2^{22} slots and 94.3 MB of memory; this is a space overhead of about 300 bits per string. The compact hash was slightly faster at 4.1 and 3.8 seconds for construction and self-search, respectively, using 2^{22} slots and only 78.8 MB of memory; this is an overhead of about 204 bits per string. The array hash with exact-fit achieved its fastest time of 4.1 and 3.9 seconds for construction and self-search, respectively, using only 2^{21} slots and 63.6 MB of memory; this is a space overhead of about 109 bits per string, almost three times less than the standard hash.

As observed in previous experiments, the array hash was slightly slower to build and self-search than the compact hash, due to the lack of move-to-front during search and the resizing of arrays during construction. However, the increase in instructions was small and greatly compensated by the high reductions of L2 and TLB misses. As a result, the array hash was consistently faster to build and self-search than the standard hash table. The L2, TLB, and instructions costs incurred by the standard, compact, and array hash tables during self-search are shown in Figure 4.13. Increasing the number of slots available had little impact on the performance of the array hash, due to high reductions in L2 and TLB misses; the chaining hash tables, in contrast, could only compete in cache efficiency once given enough space.

The clustered hash table was slightly faster to self-search than the standard and compact hash, but only under heavy load and in no case was it superior to the array hash. In the previous experiments, however, the clustered hash table was found to be consistently slower than both the

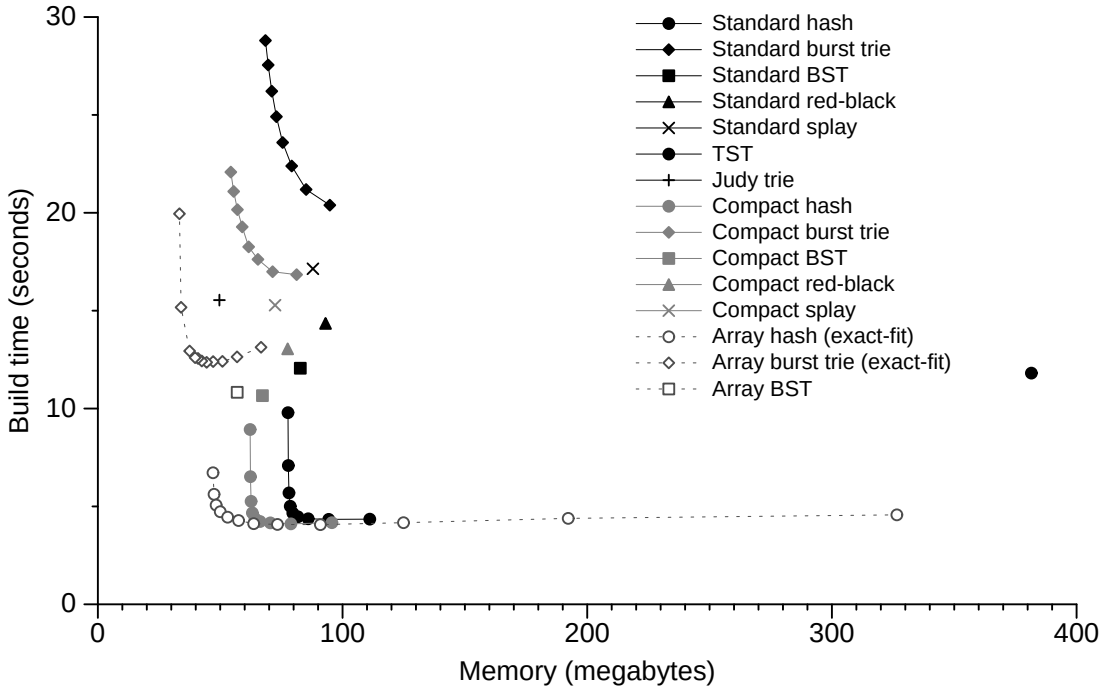


Figure 4.8: The time and space required to build the data structures using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

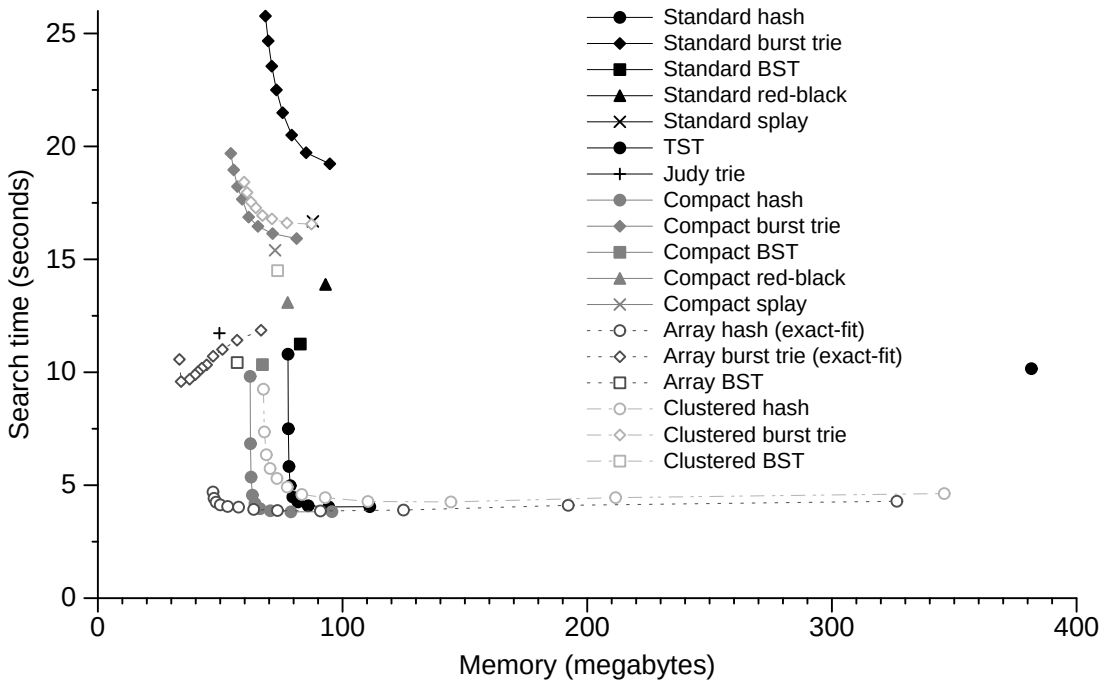


Figure 4.9: The time and space required to self-search the data structures using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128, 256, *and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

standard and compact hash tables. Hence, these results suggest that the clustered hash table may become more effective at exploiting cache when there is little to no skew in the data distribution. Without skew, traversing a clustered list will effectively resemble a linear scan, which can make good use of cache. We consider this in later experiments.

4.3.2 Burst tries

As in our experiments involving the TREC dataset, the array burst trie displayed strong improvements over its chained variants, approaching the speed of hashing while requiring the least amount of space. At its best, the array burst trie took 12.9 and 9.6 seconds to construct and self-search, respectively, while consuming only 37.4 MB; this is less space than required by the strings alone, which is the result of eliminating pointers and the pruning of shared prefixes in containers.

The compact burst trie also showed consistent improvements in both the time and space compared the standard burst trie, though not as strong as the array burst trie. At its best time, the standard burst trie required 20.3 and 19.2 seconds to construct and self-search, respectively, and about 94.2 MB of space; this is a space overhead of about 300 bits per string. Remarkably, the array burst trie was able to almost halve the time required by the standard burst trie, while imposing no space overhead per string.

The clustered burst trie was slightly faster to self-search than the compact burst trie, but only with large containers. Once the container size decreased, so did its performance relative to the compact burst trie. Nonetheless, the clustered burst trie remained faster to self-search than the standard burst trie, due to the reduction of L2 and TLB misses. The L2, TLB, and instruction costs incurred by the burst tries during self-search are shown in Figure 4.13.

As observed in the previous experiments, use of large containers in the array burst trie was both cache- and space -efficient, but also incurred high numbers of instructions. As the container threshold increased from 128 to 512 strings, for example, the number of instructions executed per search by the array burst trie increased from 1175 to 2266. The URLS dataset is not as skew at the TREC dataset, and as a consequence, the effectiveness of move-to-front is reduced. This caused the standard, clustered, and compact chains to incur high instructions costs on search, as more nodes were likely to be inspected. For example, as the container threshold increased from 128 to 512 strings, the number of instructions executed per search by the standard burst trie, increased from 1149 to 2251 — which is almost as expensive as the array burst trie. Hence, the standard and compact burst tries were slow to access due to high cache and instruction costs.

The array burst trie, however, substantially reduced the number of cache misses incurred and as a result, was up to 60% faster than its chained variants while simultaneously saving space. Nonetheless, as its container threshold increased from 128 to 512 strings, the computational cost of accessing arrays could not be entirely masked by the reduction of cache misses. As a consequence, the array burst trie became more expensive to access relative to its smaller containers, despite a further reduction in cache misses. For example, when the container size increased from 256 to

512 strings, the array burst trie was around 25% slower (or about 5 seconds) to build, but still remained greatly superior to its chained equivalents.

4.3.3 Variants of BST, the TST, and Judy

The standard, clustered, compact, and array BSTs performed surprisingly well compared to the tries. While not as compact, all three models — the array BST in particular — required less time to construct and self-search than all representations of burst tries; the array burst trie could only rival in speed during self-search.

Although strings in the burst tries are also stored in occurrence order, these strings share long prefixes, such as "http://www", which in a burst trie, implies access to a potentially larger trie index. Trie nodes are computationally efficient but accessing a large number of them before acquiring a container can result in an increase in cache misses. With a container threshold of 50, for example, 50,627 tries nodes were created, as opposed to the 6009 created with the TREC dataset. Consequentially, as seen in Figure 4.13, the number of cache misses incurred by the array burst trie was, at best, only slightly less than that incurred by the BSTs. Hence, although the computational cost of the BSTs was higher, the reduction in cache misses compensated.

The standard and compact red-black and splay trees were also faster to build and self-search than the chaining burst tries, but required more space and were slower to access than the BSTs. Judy also showed strong gains in performance relative to the previous TREC experiments, rivaling the speed of the standard and compact red-black and splay tree, while requiring the least amount of space of all standard-chained and compact-chained data structures. Judy also required less space than the array BST — due to the pruning of shared prefixes — and was as space-efficient as the array hash table. Nonetheless, Judy remained slower to build and self-search than the standard, compact, and array BST, the array hash, and array burst trie, as it incurred more cache misses.

The TST also rivaled the build and self-search time of the burst tries and BSTs, due to low instruction costs. However, the TST required almost 400 MB of space — which is almost 4 times the space required by the standard BST. Although the clustered BST remained faster to access than the compact or standard burst tries, it was ineffective at improving the cache-efficiency of the standard BST. The cache performance of these data structures is shown in Figure 4.13.

Our results show that the standard burst trie is not always the fastest data structure for vocabulary accumulation, as claimed by Heinz et al. [2002]. For long strings that share long prefixes, the standard BST can be faster, though its high computational cost and $O(N)$ worst-case does not necessarily make it a practical choice. For example, as the number of distinct URLs increase to say ten million, or with a substantial increase in the number of searches, the performance of the BST will likely deteriorate due to the excessive cost of binary search — as observed in the previous TREC experiments. The burst trie, however, is more scalable and can operate efficiently with large numbers of strings, as a result of removing shared prefixes that reduce

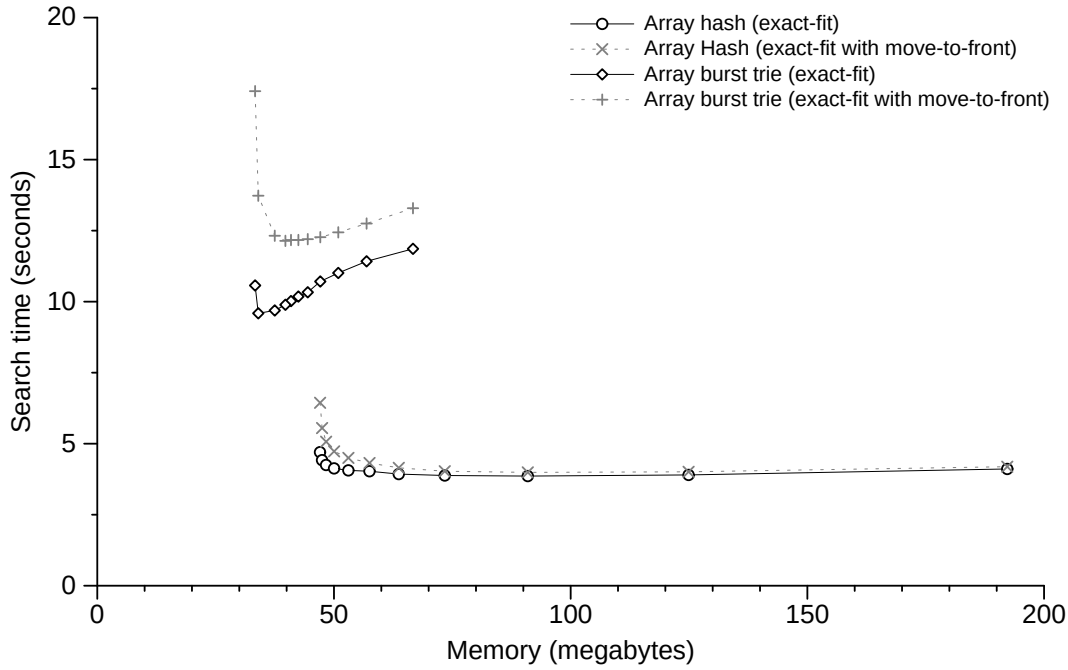


Figure 4.10: The time and space required to self-search the array hash and array burst trie with and without move-to-front on access, using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

both space and computational costs, and the use of bounded-size containers.

4.3.4 The effectiveness of move-to-front on arrays

Figure 4.10 compares the time required to self-search the array hash and array burst trie with and without move-to-front on access. There were no cases where move-to-front improved the performance of the array hash and array burst trie. As discussed in the previous TREC experiments, performing a move-to-front in an array is computationally expensive, due to string copying. Our results show that move-to-front is unnecessary for the array-based data structures, as any potential gains seem likely to be low.

4.3.5 Paging vs. exact-fit array growth

The cost of constructing the array burst trie and array hash with and without paging is shown in Figure 4.11. The use of paging led to consistent improvements for the array burst trie, which was up to 11% faster (or about 1.7 seconds) than exact-fit. Exact-fit involves growing a dynamic array on every insertion by copying it into a new larger space. Paging reduces the amount of copying to save time, but at a small cost in space. Paging was not particularly useful for the array hash, apart from the slight improvement observed under heavy load. In all, the savings offered by the paging technique in these experiments were small, making the more space-efficient exact-fit growth policy preferable.

4.3.6 Performance without hardware prefetch

We repeat the previous skew search experiment on the Pentium III processor, to observe the value of eliminating pointers on a slower machine with no hardware prefetch. We consider the three best data structures — the hash table, burst trie, and BST — using their standard, compact, and array-based representations, and measure the time and space required to self-search using the URLS dataset. Results are shown in Figure 4.12.

The relative performance between the hash table, burst trie, and BST was consistent with previous experiments. The standard, compact, and array BSTs remained faster to access than the standard and compact burst tries, while the array burst trie displayed strong gains in performance — approaching the speed of the array hash while consuming the least amount of space. The hash tables remained the fastest data structures, with the array hash being consistently faster and smaller than the standard hash table. The compact hash was slightly faster than the array hash due to move-to-front on access, but required more space. These results show that the absence of hardware prefetch and the use of a slower processor with smaller cache-lines — which can increase the cache-miss rate — has minimal impact on the relative performance between the array-based hash table, burst trie, and BST.

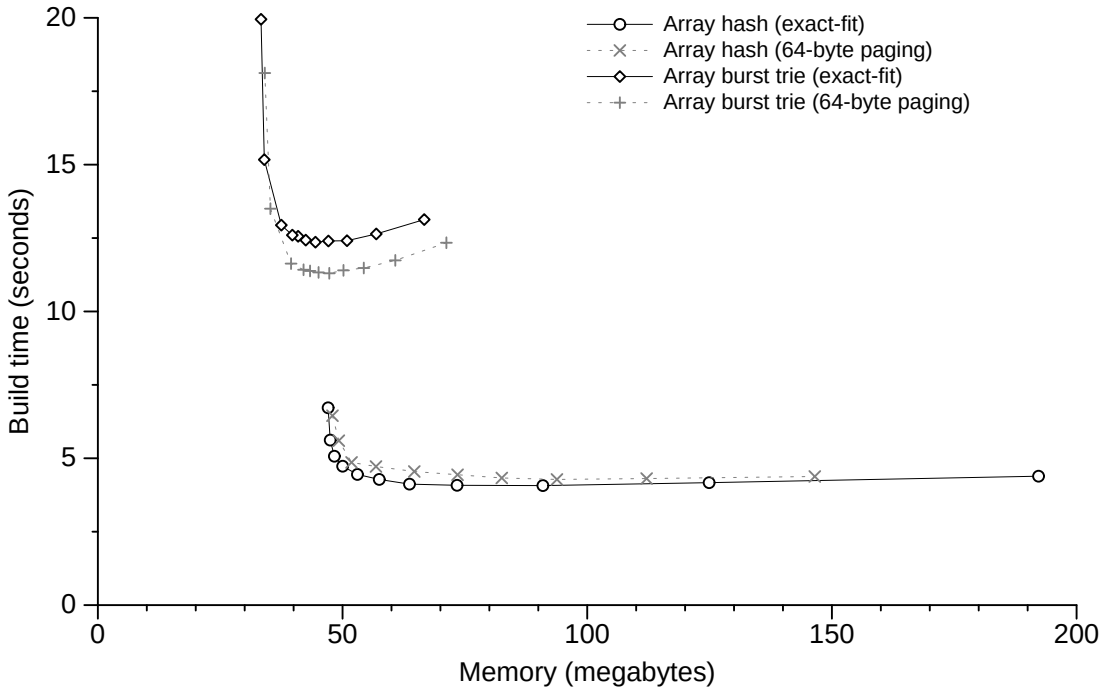


Figure 4.11: The time and space required to build the array hash and array burst trie with and without paging, using the URLS *dataset. Paging grows an array in 64-byte chunks, in contrast to exact-fit. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

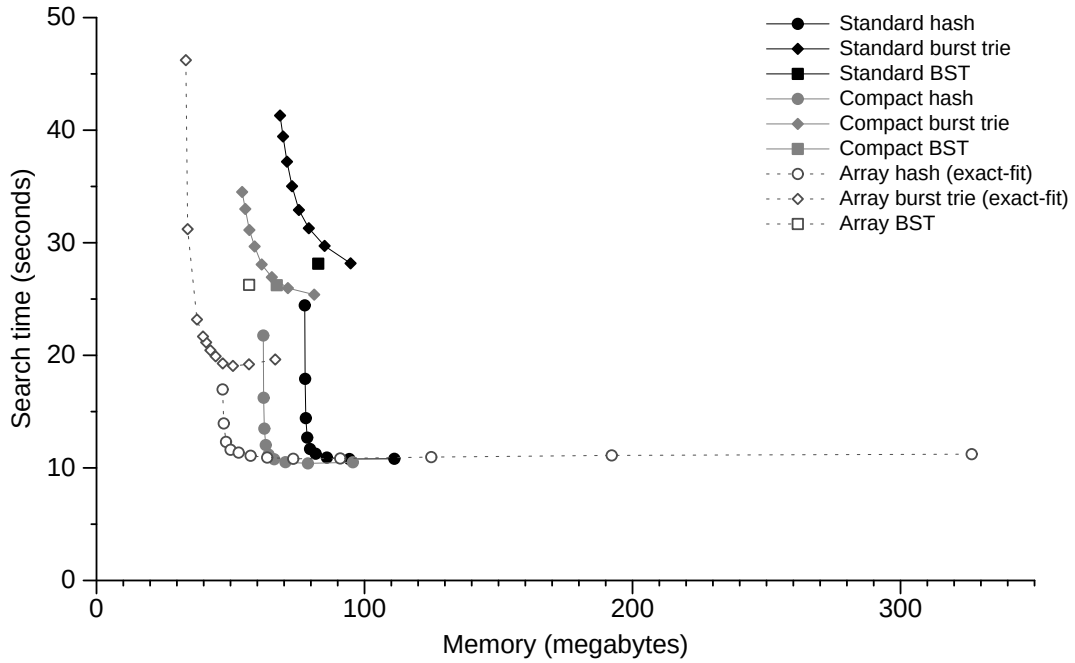
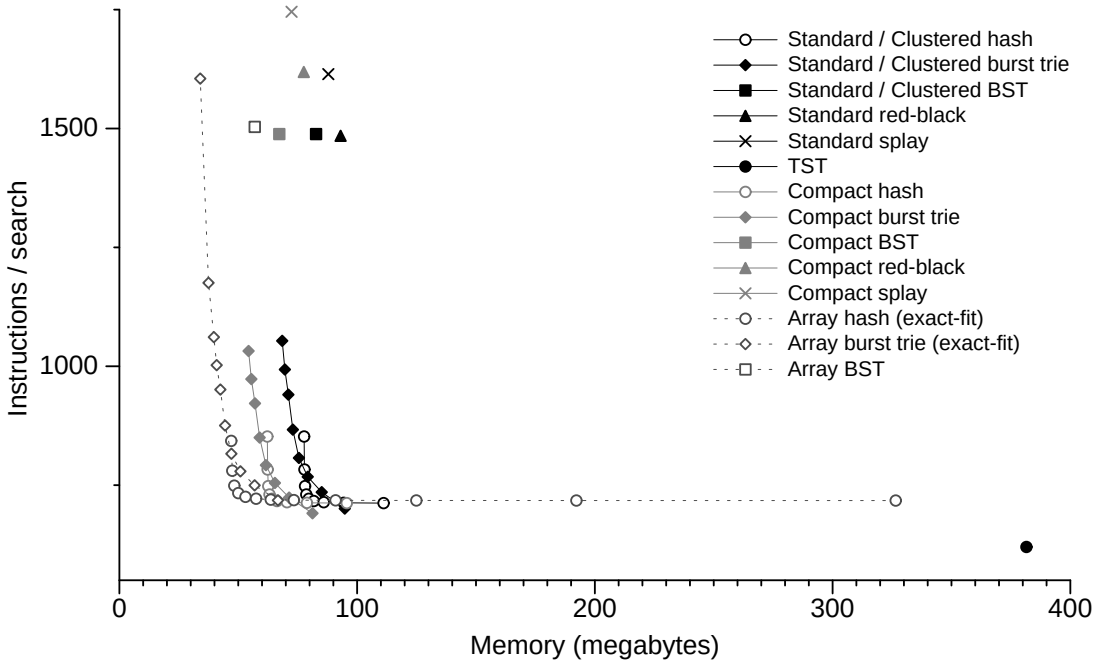
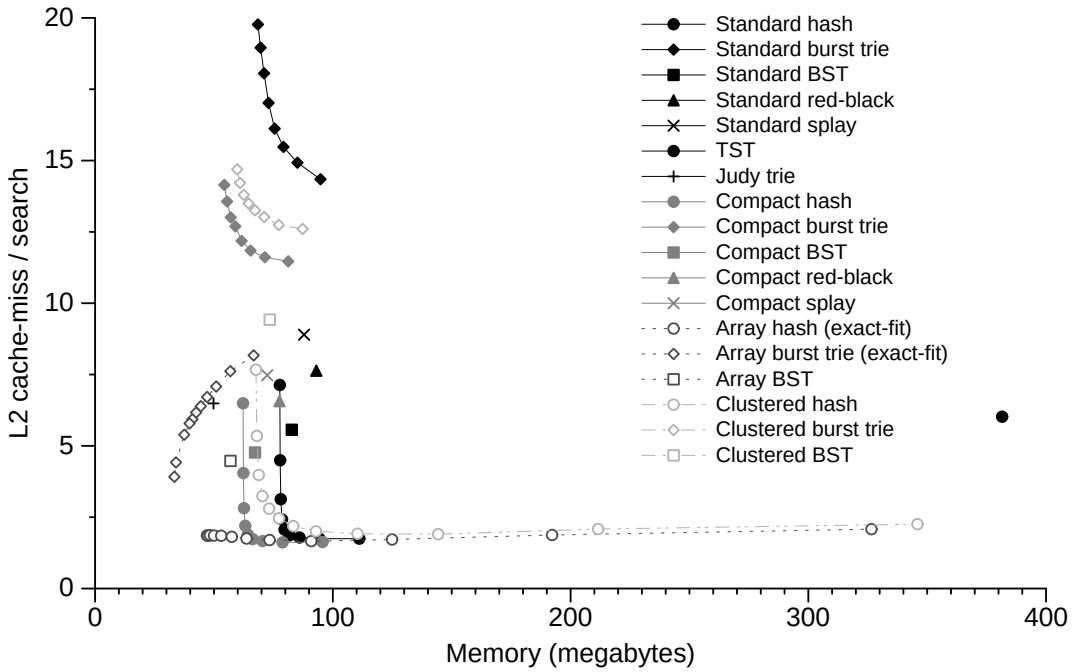


Figure 4.12: The time and space required to self-search the data structures using the URLS *dataset on the Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

(a) Instructions per search



(b) L2 cache-miss per search



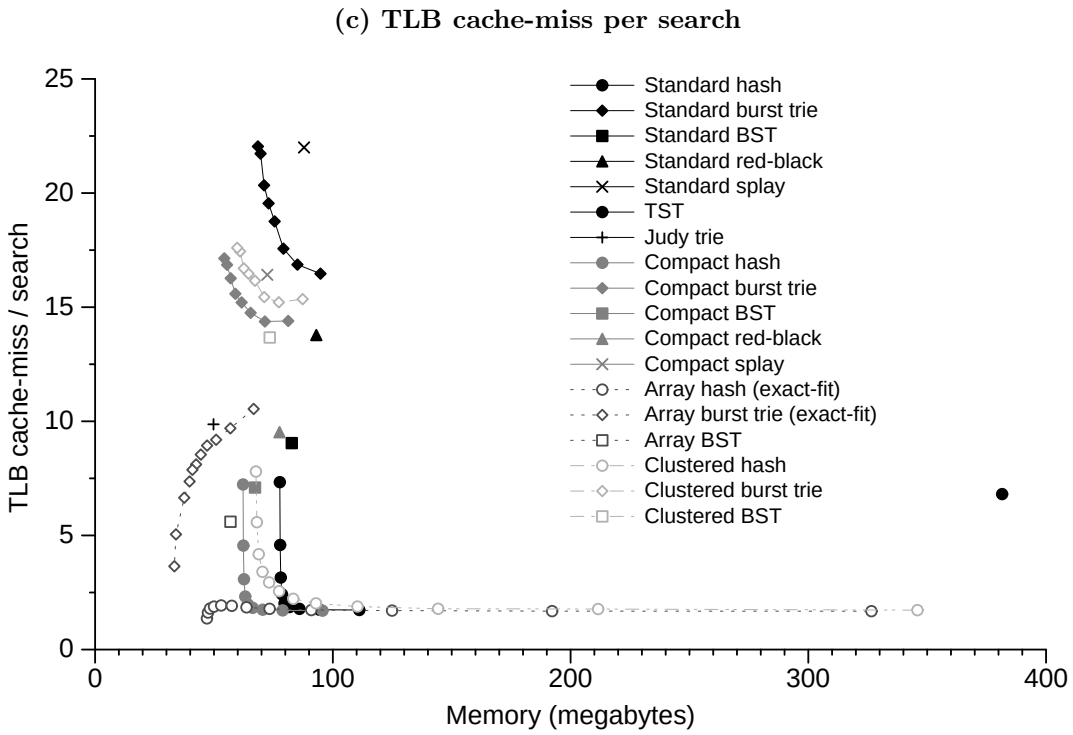


Figure 4.13: The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256 *and* 512 *strings. We omit the instruction cost for containers with a threshold of 512 strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{24}*.*

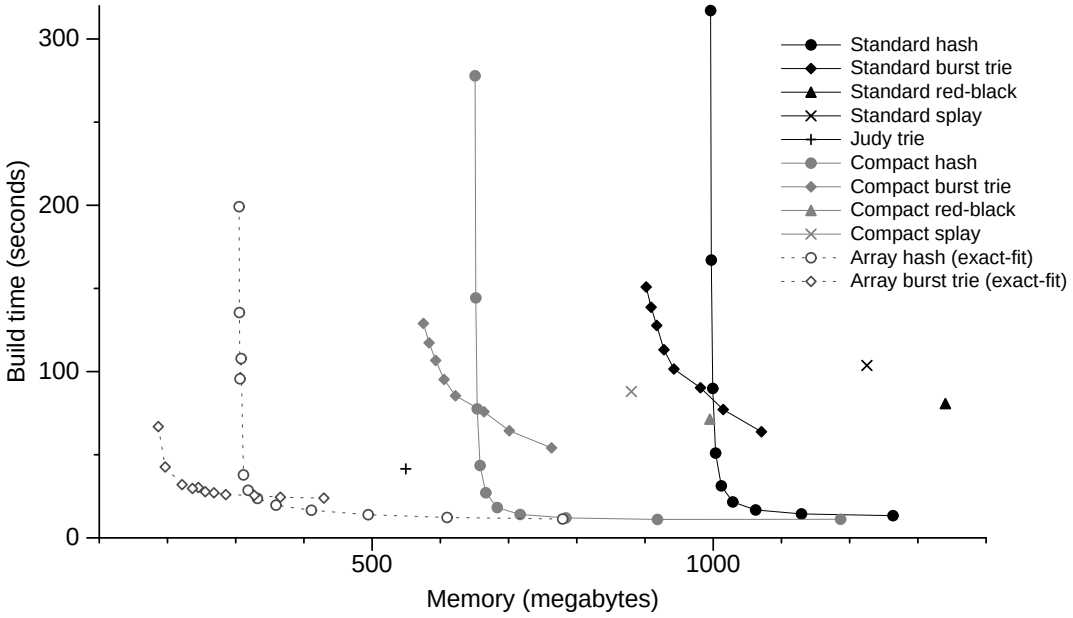


Figure 4.14: The time and space required to build the data structures using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{26}*.*

4.4 Distinct data

We then used the DISTINCT dataset for construction and self-search. This dataset contains no repeated strings, and thus every insertion requires that the data structure be fully traversed. With no string repetitions, move-to-front on access is rendered ineffective for both chains and arrays. Results for construction and self-search are shown in Figure 4.14 and Figure 4.15.

Table 4.3: Elapsed time (in seconds) required when the DISTINCT *dataset is used to construct and self-search the variants of hash table.*

	Num. of	**Array**		**Clustered**	**Compact**	**Standard**
	slots	page	exact			
Construction	2^{15}	114.5	199.1	—	2082.0	2380.6
	2^{16}	106.4	135.5	—	1055.5	1209.0
	2^{17}	99.8	95.6	—	539.7	620.2
	2^{18}	68.3	78.8	—	277.8	317.1
	2^{19}	31.1	37.8	—	144.3	167.0
	2^{20}	19.6	28.6	—	77.5	89.8
	2^{21}	17.6	23.7	—	43.5	50.9
	2^{22}	16.3	19.6	—	27.1	31.2
	2^{23}	14.4	16.6	—	18.2	21.6
	2^{24}	12.5	13.9	—	14.0	16.7
	2^{25}	12.0	12.2	—	12.0	14.4
	2^{26}	11.3	11.4	—	11.0	13.3
	2^{27}	11.8	10.9	—	11.1	13.5
Self-search	2^{15}	—	77.9	1310.7	2076.5	2370.0
	2^{16}	—	47.4	567.1	1050.8	1201.7
	2^{17}	—	32.1	294.6	535.4	612.7
	2^{18}	—	23.7	156.0	273.8	314.7
	2^{19}	—	19.0	87.2	141.2	160.1
	2^{20}	—	15.9	53.2	74.5	84.6
	2^{21}	—	13.9	35.4	41.1	46.3
	2^{22}	—	12.7	25.4	24.4	27.3
	2^{23}	—	11.9	19.5	16.0	17.8
	2^{24}	—	11.3	15.4	11.8	12.8
	2^{25}	—	10.3	12.5	9.7	10.4
	2^{26}	—	9.5	10.6	8.9	9.3
	2^{27}	—	8.9	10.7	8.6	13.2

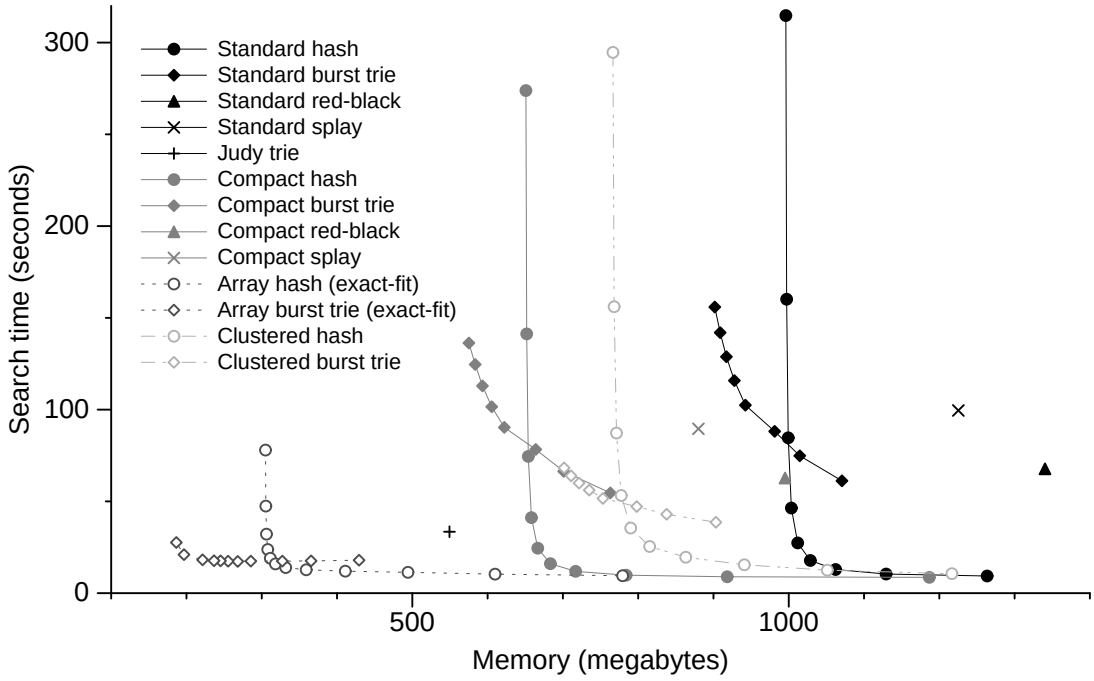


Figure 4.15: The time and space required to self-search the data structures using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{26}*.*

4.4.1 Hash tables

The difference in performance between the array and chaining methods is startling. The time required to construct and self-search the standard, clustered, compact, and array hash tables are shown in Table 4.3. This is an artificial case, but highlights the fact that random memory accesses are highly inefficient. With only 2^{15} slots, for example, the exact-fit array hash table was constructed in about 199 seconds, whereas the compact and standard chains required about 2082 and 2380 seconds, respectively. The use of paging further reduced the construction time of the array hash to only 114 seconds, a saving due to the lack of excessive copying. This speed is despite the fact that the average load factor is 878. As we anticipated in previous experiments, use of paging is effective with large numbers of insertions. As the load factor decreased, however, paging became less effective as fewer strings were inserted into each slot.

The results for self-search are similar to those for construction, with the array hash being up to 97% faster than the chained hash tables, with search time falling from over 2370 seconds to under 80 seconds, while space simultaneously falls from 995 MB to 304 MB. Once again, increasing the number of slots allows the chained hash tables to be much faster, but the array hash remains competitive at all table sizes. The chained hash tables approach the efficiency of the array hash only when given surplus slots. For example, with 2^{27} slots, the compact hash table is by a small margin the fastest method, but required over 1186 MB with the equivalently-sized standard chain at 1531 MB. The array hash, however, achieved almost the same speeds using 2^{24} slots and a total of 494 MB, a dramatic saving.

As anticipated from previous experiments, the clustered hash table displayed good performance with no skew in the data distribution, being up to 45% faster to self-search than the compact and standard hash tables. However, the clustered hash could only remain efficient under heavy load; as the average load factor decreased, so did its performance, until finally becoming slower than both the compact and standard hash tables. There were no cases where the clustered hash was superior to the array hash, which was up to 94% faster. Furthermore, with both pointers and nodes eliminated, a substantial amount of space is saved without any impact on performance. These results demonstrate that combining clustering with pointer elimination via the use of dynamic arrays, is by far more effective at exploiting cache than clustering alone.

Table 4.4: The time and space required to build and self-search a standard, clustered (search only), compact, and array BST, using the DISTINCT *dataset.*

BST	Build (sec)	Self-search (sec)	Space (megabytes)
Array	497.1	471.1	534.7
Compact	594.8	569.3	764.9
Clustered	—	721.0	880.0
Standard	758.6	740.6	1110.1

The cache performance of the standard, clustered, compact, and array hash tables during self-search are shown in Figure 4.16. These results show a strong correlation with the self-search times reported in Figure 4.15. For example, the array hash sustained low L2 and TLB costs relative to its chained representations, and was thus faster to access under heavy load. In these experiments, move-to-front on access was rendered ineffective due to the absence of duplicate strings. As a consequence, searching the array hash incurred almost the same number of instructions per search as the equivalent chains. During construction, the array hash executed more instructions due to array resizing, but of which was greatly compensated with a high reduction in cache misses. As a result, the array hash could be built up to 92% faster than the chained hash tables while under heavy load — where arrays are longer and thus more expensive to resize. For uniform access distributions, the array hash is by far the most space- and time -efficient data structure when sorted access to strings is not required.

4.4.2 The variants of BST, the TST, and Judy

The BSTs were expensive to construct and self-search but remained competitive in space, with the array BST in particular, requiring less space than the compact and standard hash tables and burst tries. The time and space required to build and self-search the standard, clustered, compact, and array BST, are shown in Table 4.4.

At best, the array BST took 497 seconds to construct and 471 seconds for self-search, using 534 MB of space. Although this is a considerable improvement over the standard BST, which required over 1110 MB of space and more than 740 seconds for both construction and self-search, the computational cost of binary search was expensive. To self-search the array BST, for example, over 34,000 instructions per search were executed. This is far more expensive than the array hash, which executed only 511 instructions per search, while using almost the same amount of space — 2^{25} slots and 609 MB. The clustered BST was not particularly effective at exploiting cache, showing only small gains in performance over the standard BST.

In previous experiments, the standard, compact, and array BSTs were competitive in speed,

due to the presence of a skew distribution. This permitted frequently access tree paths to reside longer in cache, reducing cache misses which compensated for the use of an unbalanced structure. In these experiments, however, no access skew was present, and as a result, previously cached paths were likely to be evicted from cache prior to reuse. As a result and despite the improvements offered by the array BST, the BSTs were amongst the slowest data structures to access.

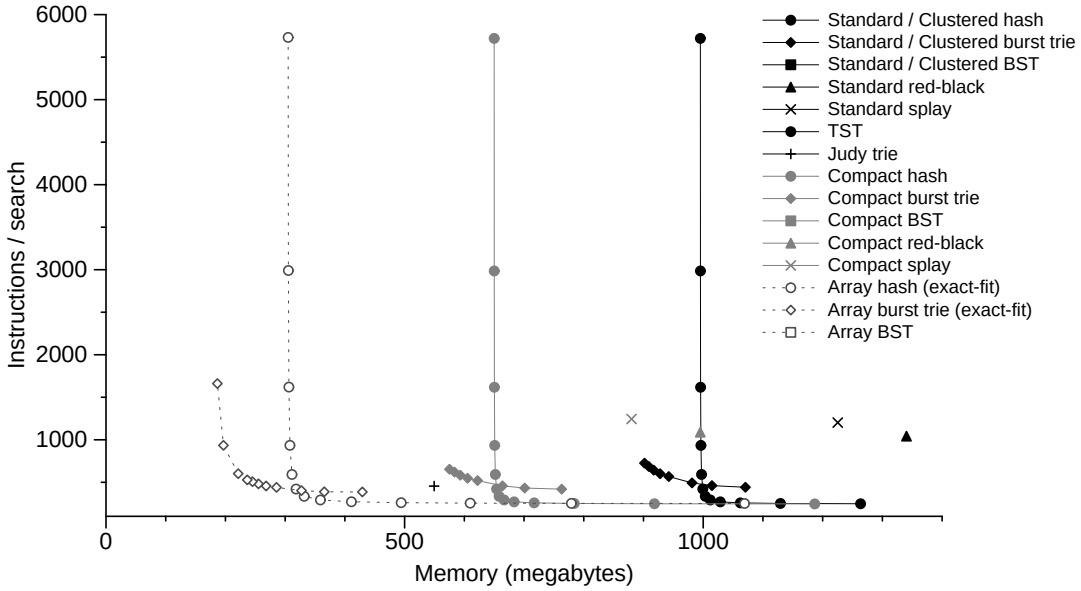
The standard splay and red-black trees, for both construction and self-search, performed relatively well compared to the BSTs. Splaying amortized the cost of access, reducing the number of nodes inspected per search, which in turn reduced the number of cache misses and instructions executed. The standard and compact splay trees, however, were slower to access than the standard and compact red-black trees, as a balanced structure is more effective when there is no access skew in the data distribution. The TST exhausted main memory, requiring almost 2.5 GB of space, and as a consequence it required over 5500 and 13,000 seconds to construct and self-search, respectively, due to the involvement of virtual memory. The Judy data structure was significantly faster than any of the trees, as a result of its efficient use of cache (Figure 4.16). Judy also required less space than the standard, clustered, and compact hash tables, but its performance remained markedly inferior in both time and space, when compared to the array hash and array burst tries.

4.4.3 Burst tries

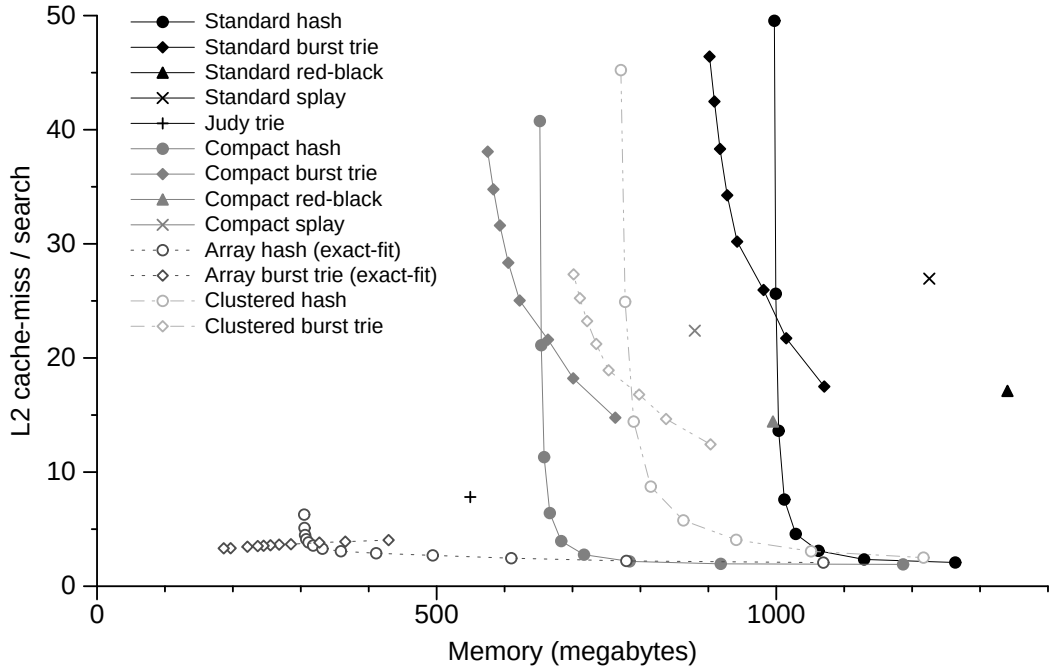
The array burst trie showed substantial improvements over its compact and standard representations, requiring the least amount of space, at best a total space usage of 186 MB — that is 119 MB less space than the original dataset — while access times were also reduced and approached those of the array hash table. With a container threshold of 128 strings, the array burst trie required only 221 MB of space to achieve a self-search time of 18.1 seconds. The array hash, in contrast, needed over 311 MB of space (or 2^{19} slots) to achieve a faster time. Paging was consistently faster than exact-fit, allowing the array burst trie to be constructed up to 24% faster (or around 15 seconds) with large containers, but at the cost of wasting up to 120 MB of space with small containers.

The cache performance of the burst tries during self-search is shown in Figure 4.16. There is a strong correlation between cache performance and the time required for self-search. For example, the array burst trie was, in all cases, faster than the compact, clustered, and standard burst trie, because it sustained low L2 and TLB misses. Figure 4.16 shows that the cache efficiency of the array burst trie improves with an increase in container size, and yet, as shown in Figure 4.14 and Figure 4.15, the array burst trie becomes slower to access relative to its smaller containers. This is caused by a high increase in instructions; a cost which is not entirely masked by the reduction of cache misses. The equivalent chained burst tries also incurred high instruction costs, but made inefficient use of cache which dominated their performance. Hence, as observed in previous experiments, in order to attain a fast and scalable burst trie, it is necessary to use containers that are both cache-efficient and conservative with the number of instructions they execute.

(a) Instructions per search



(b) L2 cache-miss per search



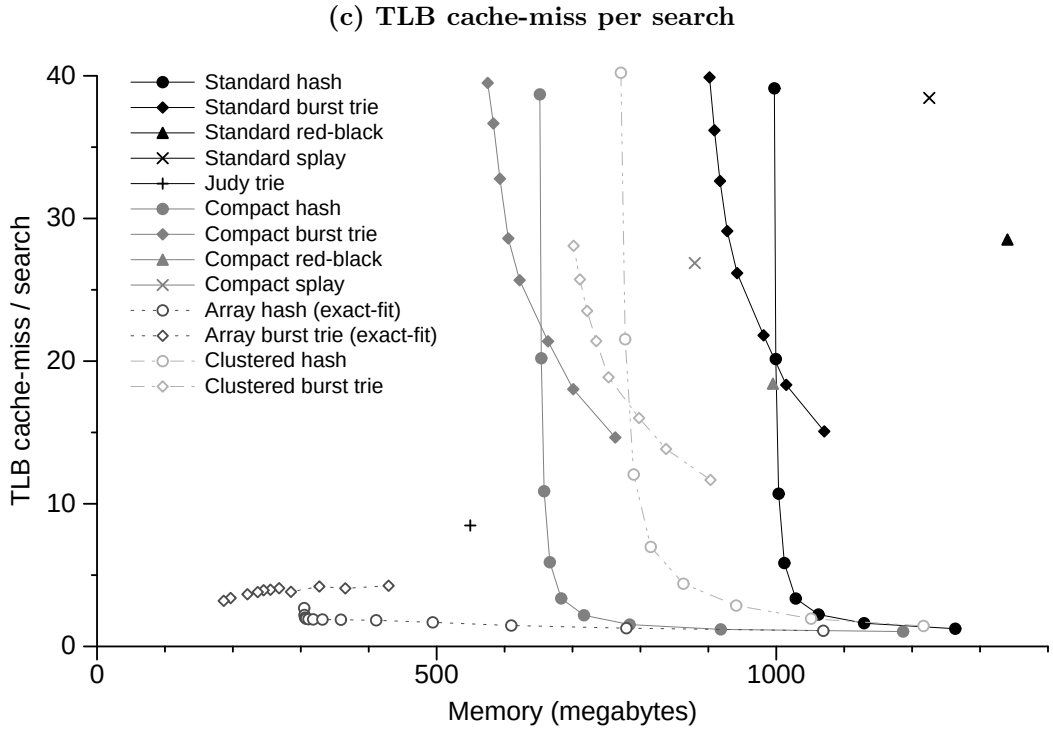


Figure 4.16: The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{26}*.*

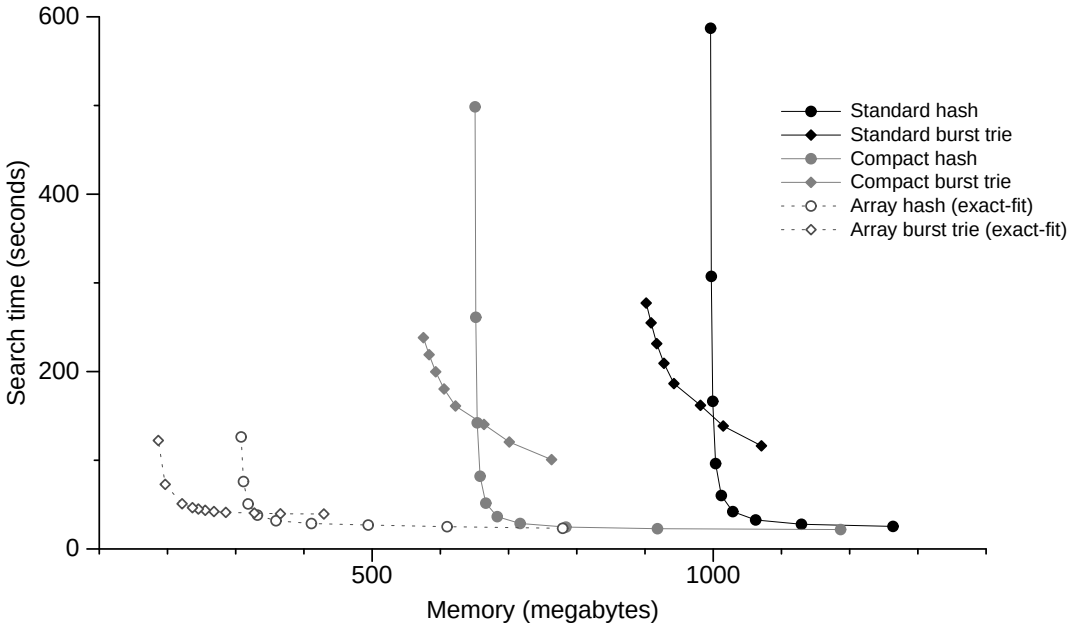


Figure 4.17: The time and space required to self-search the data structures using the DISTINCT *dataset on the Pentium III with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{18} *slots, doubling up to* 2^{26}*.*

4.4.4 Performance without hardware prefetch

We repeat the previous self-search experiment to observe the impact on performance using the Pentium III, which does not employ hardware prefetch. We considered the three best data structures — the hash table, burst trie, and BST — using their standard, compact, and array-based representations. The results for the hash tables and burst tries are shown in Figure 4.17.

The array burst trie and array hash table were substantially faster to search than their chained representations, even without the aid of a hardware prefetcher. The array burst trie displayed the strongest gains — approaching the speed of the array hash while consuming the least amount of space. With 2^{18} slots, the average load factor was about 110 strings per slot. In this case, the array hash took under 126 seconds to self-search, whereas the equivalent compact and standard hash tables required 498 and 587 seconds, respectively; that is, the array hash was up to 79% faster. Compared to its previous performance shown in Table 4.3, the array hash was only 15% slower on the Pentium III. The array BST also showed considerable improvement over its chained variants, being up to 24% faster; the standard BST required 1918 seconds to self-search, which was reduced to 1603 seconds by the compact BST, to a further 1449 seconds by the array BST, while achieving considerable reductions in space. These results demonstrate that our methods offer consistent and substantial improvements.

4.5 The adaptive trie

We downloaded a high-quality implementation of an adaptive trie [Acharya et al., 1999], to compare its performance against our set of data structures including the cache-conscious hash table, burst trie, and binary search tree. The current implementation of the adaptive trie (for small alphabet sizes) is only compatible with lower-case alphabetic characters. We therefore filtered our DISTINCT and TREC datasets to remove all strings that contain either a upper-case or non-alphabetic character, forming the FILTERED DISTINCT and FILTERED TREC datasets, respectively. The FILTERED DISTINCT dataset contains 7,892,272 unique strings. We truncated the FILTERED TREC dataset to contain only the first 50 million strings from the total of 141,693,220 strings filtered. The time (in seconds) and space (in megabytes) required to build and self-search the data structures using the FILTERED DISTINCT dataset, is shown in Figure 4.18 and Figure 4.19.

With no skew in the data distribution, the adaptive trie was faster to build and self-search than the trees and the standard and compact burst tries. Given enough space, however, the hash tables were superior. Similarly with a container threshold of less than 512 strings, the array burst trie was faster to build and self-search than the adaptive trie. The Judy trie and the TST were also slightly faster to build and self-search. Despite its competitive speed, however, the adaptive trie was the most space-intensive data structure, requiring over 900 MB of memory, due to the space overhead of maintaining adaptive trie nodes. The space required by the adaptive trie was almost twice that of the TST, and around a factor of 10 more than the array burst trie and array hash table.

Figure 4.20 shows the self-search performance of the adaptive trie using the FILTERED TREC dataset. We omit the cost of construction as it was found to be similar to that of search. As claimed by Crescenzi et al. [2003], the adaptive trie performed poorly under skew access, requiring over 17 seconds to self-search, which places it amongst the slowest data structures to access under skew, being only slightly faster than the red-black tree. Moreover, the adaptive trie required over 22 MB of memory, which is more space than required by the TST. These results demonstrate that the adaptive trie is not a practical choice for managing a large set of strings in memory, due to its high space requirements and poor skew performance.

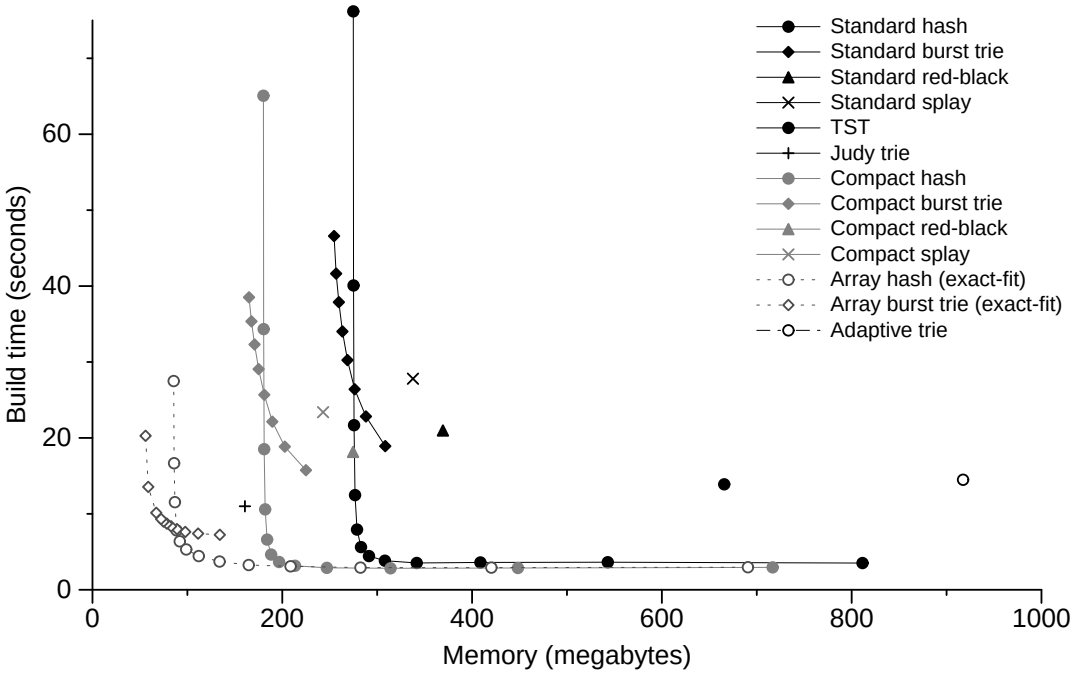


Figure 4.18: The time and space required to build the adaptive trie using the FILTERED DISTINCT *dataset, described in Section 4.5. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{16} *slots, doubling up to* 2^{24}*.*

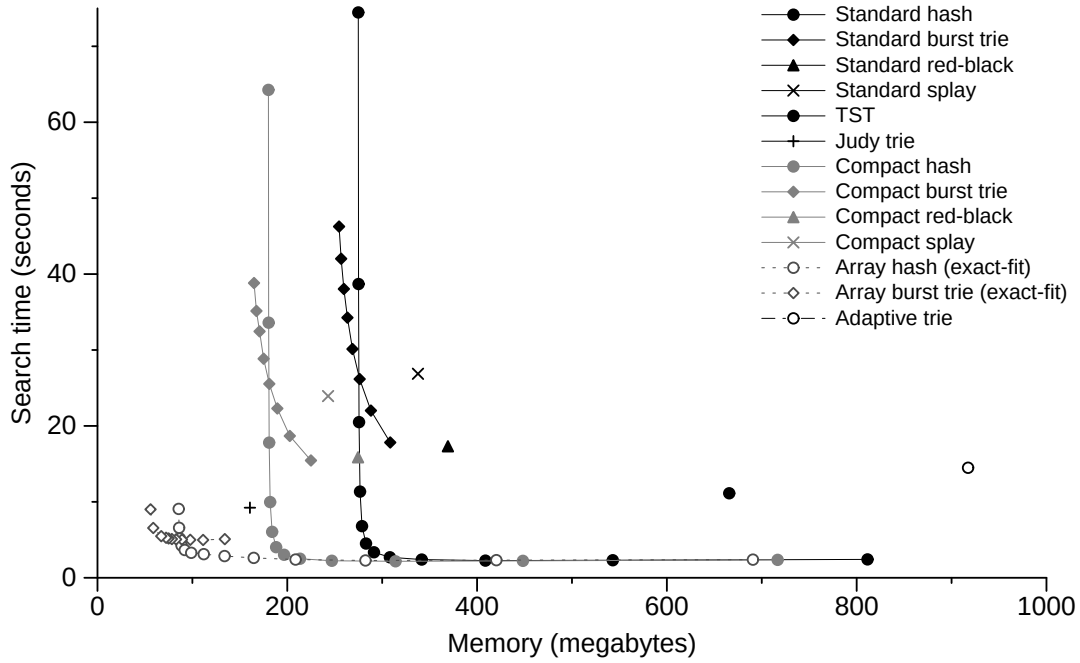


Figure 4.19: The time and space required to self-search the adaptive trie using the FILTERED DISTINCT *dataset, described in Section 4.5. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{16} *slots, doubling up to* 2^{24}*.*

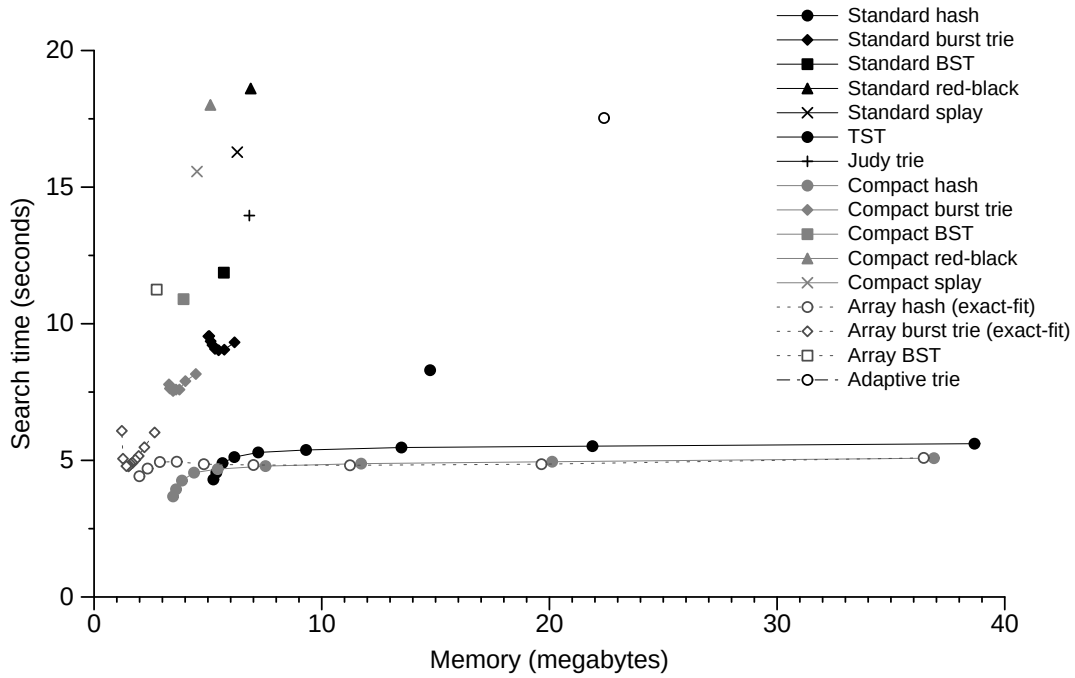


Figure 4.20: The time and space required to self-search the adaptive trie using the FILTERED TREC *dataset, described in Section 4.5. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{16} *slots, doubling up to* 2^{24}*.*

4.6 Custom memory allocation

In this experiment, we use the DISTINCT dataset to measure the time (in seconds) required to build the standard hash table using *malloc*, the general-purpose memory allocator that is provided by most Linux operating systems. We then compare the build time against two custom memory allocators: *ialloc* and *pre-alloc*. *Ialloc* is a cache-conscious heap allocator that attempts to improve the cache-efficiency of pointer-intensive data structures, by interleaving or clustering the internal fields of homogeneous nodes [Truong et al., 1998]. With *pre-alloc*, we allocate a gigabyte of memory as a contiguous array of characters prior to building the hash table, and maintain a counter (that is, a pointer) to the next available byte in the block. The pointer is then returned on an allocation request for a node and is incremented to point to the next available byte. We compare the time required to build the standard hash table using *malloc*, *ialloc*, and *pre-alloc*, in Table 4.5.

As claimed by Berger et al. [2002], custom memory allocation often results in poor program performance when compared to general-purpose allocators, such as *malloc*. We observed similar results in our experiments. *Ialloc* can improve the cache-efficiency of a single linked list, but is ineffective for a standard hash table which maintains a set of linked lists that can be accessed and grown at random. In this case, *ialloc* will interleave nodes from different slots, which will pollute cache-lines with unwanted data. As a consequence, more cache misses were incurred, resulting in a hash table that was up to 25% slower to build than the identical hash table that used *malloc*.

Pre-allocating space offered slightly better performance than *ialloc*, but was still inefficient compared to *malloc*, resulting in a hash table that was up to 25% slower to build. Although the general-purpose *malloc* allocator is not as space-efficient as *ialloc* or *pre-alloc* — due to the 8-byte allocation overhead imposed per call — our results have shown that *malloc* can lead to better use of cache on dynamic pointer-intensive data structures. Hence, the data structures that we explore in this thesis are built using the general-purpose *malloc* or *calloc* memory allocators.

Num. of slots	Malloc (sec)	Ialloc (sec)	Pre-allocation (sec)
2^{15}	2380.6	3160.3	3163.1
2^{16}	1209.0	1623.4	1603.2
2^{17}	620.2	839.1	820.9
2^{18}	317.1	435.5	426.5
2^{19}	167.0	231.2	225.7
2^{20}	89.8	124.8	119.5
2^{21}	50.9	72.8	66.4
2^{22}	31.2	44.9	40.0
2^{23}	21.6	32.5	26.1
2^{24}	16.7	24.5	19.0
2^{25}	14.4	20.9	15.1
2^{26}	13.3	19.1	13.5
2^{27}	13.5	18.5	13.4

Table 4.5: A comparison of the elapsed time in seconds required to build a standard hash table using the general-purpose memory allocator "malloc", compared to the custom cache-efficient heap allocator "ialloc" [Truong et al., 1998], and a "pre-allocator". The pre-allocator simply allocates a large contiguous block of memory prior to building the hash table: a pointer to the next available byte is returned per memory allocation request. The DISTINCT *dataset was used to build the hash tables.*

4.7 Implicit clustered chains

A clustered chain ensures that the nodes of a linked list are stored contiguously in main memory — in order of allocation — to improve spatial access locality. However, clustering does not eliminate pointers. Chilimbi [1999] suggests eliminating the next-node pointers of homogeneous nodes in a chain, to form an implicit clustered chain where nodes are accessed via arithmetic offsets. As half of the pointers are eliminated, we expect better cache and space utilization over pointer-based clustering. As discussed in Chapter 3, implicit chains cannot support move-to-front on access, but as we show, this has little impact on performance due to a further reduction in cache misses. Several researches have also improved the cache-efficiency of software by applying similar implicit clustering techniques [Ghoting et al., 2006; Badawy et al., 2004; Luk and Mowry, 1996].

In this experiment, we construct a standard hash table using the DISTINCT and TREC datasets. We then convert the standard chains that are assigned to each slot into implicit chains, to compare the time and space required to self-search the implicit hash table against the clustered, compact, standard, and array hash tables. The results are shown in Figure 4.21 and Figure 4.22.

With half the pointers eliminated, the implicit hash table is more space-efficient than the clustered hash table, and, is almost as space-efficient as compact hash table. As discussed in Chapter 3, the implicit chain requires more space than a compact chain, due to the initial 8-byte allocation overhead per slot and the extra 4-byte null pointer which serves as a list delimiter.

The implicit hash table was consistently faster to self-search than the clustered hash table using the DISTINCT dataset (Figure 4.21). However, the relative difference between the two was small, with the implicit hash being only up to 19% faster than the clustered hash table. The implicit and clustered hash tables were faster to access than the compact and standard hash tables, but only under heavy load. Once the load factor fell below 7 strings per slot (that is, using more than 2^{22} slots), both the implicit and clustered hash tables were slower to access than the compact and standard hash tables. The array hash was the fastest and most compact data structure to access, being up to 93% faster than the implicit hash table when under heavy load, while simultaneously requiring less space.

These results demonstrate the value of combining clustering with pointer elimination through the use of dynamic arrays. Traversing a large implicit chain can incur up to two cache misses per string: one to access the string pointer and another to access its string.

In a dynamic array, in contrast, a single cache-miss will prefetch an entire cache-line of strings, and with no pointers to follow, hardware prefetch can greatly reduce the number of cache misses incurred as the array is scanned. The difference in time (in seconds) required to self-search the standard, compact, clustered, implicit, and array hash tables, using the DISTINCT dataset, is shown in Table 4.6.

Figure 4.22 shows the skew self-search performance of the standard, compact, clustered, implicit, and array hash tables, using the TREC dataset. The implicit hash table was slightly faster

Num. of slots	**Array**	**Implicit**	**Clustered**	**Compact**	**Standard**
2^{15}	77.9	1063.0	1310.7	2076.5	2370.0
2^{16}	47.4	533.8	567.1	1050.8	1201.7
2^{17}	32.1	279.6	294.6	535.4	612.7
2^{18}	23.7	149.7	156.0	273.8	314.7
2^{19}	19.0	83.3	87.2	141.2	160.1
2^{20}	15.9	50.1	53.2	74.5	84.6
2^{21}	13.9	32.9	35.4	41.1	46.3
2^{22}	12.7	24.3	25.4	24.4	27.3
2^{23}	11.9	18.9	19.5	16.0	17.8
2^{24}	11.3	14.9	15.4	11.8	12.8
2^{25}	10.3	12.1	12.5	9.7	10.4
2^{26}	9.5	10.3	10.6	8.9	9.3
2^{27}	8.9	9.8	10.7	8.6	13.2

Table 4.6: Elapsed time (in seconds) required to self-search the standard, compact, clustered, implicit, and array-based hash tables, using the DISTINCT *dataset.*

to access than the clustered hash table, but not when under heavy load due to the absence of move-to-front on access. Moreover, the implicit hash table showed relatively poor performance when compared to the standard, compact, and array hash tables, being up to 45% slower to access. Although the array hash did not employ move-to-front on access, it remained substantially faster than the implicit hash, due to the elimination of both string and node pointers, which lead to a high reduction in cache misses.

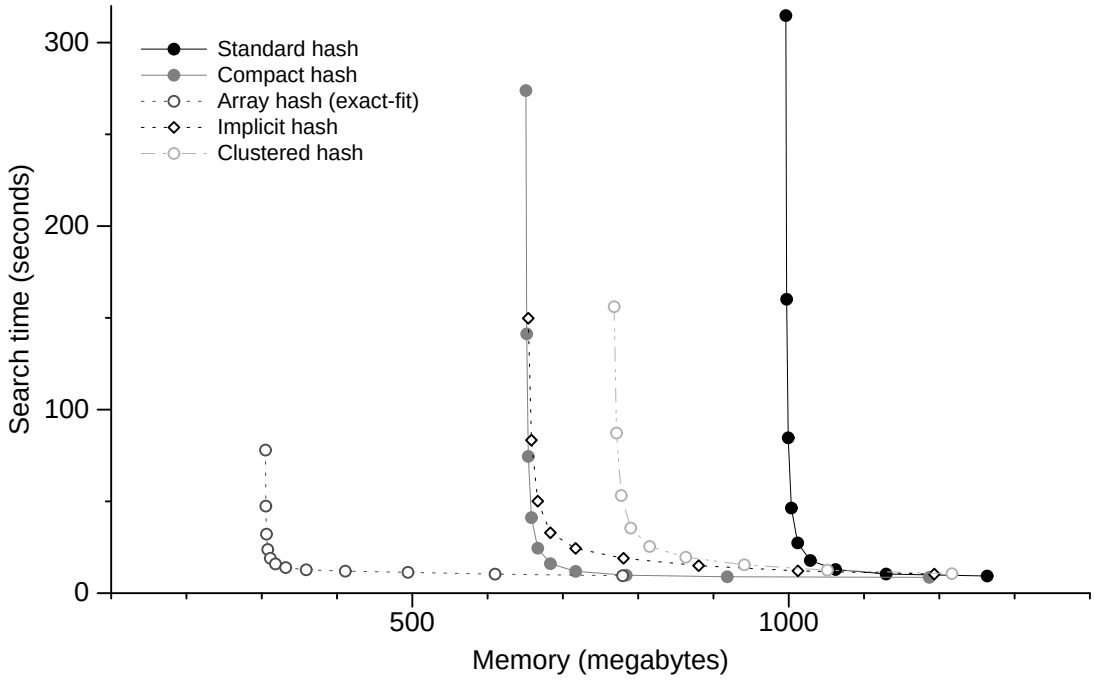


Figure 4.21: The time and space required to self-search the implicit hash table using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128, 256, *and* 512 *strings. The hash tables commenced with* 2^{16} *slots, doubling up to* 2^{24}*.*

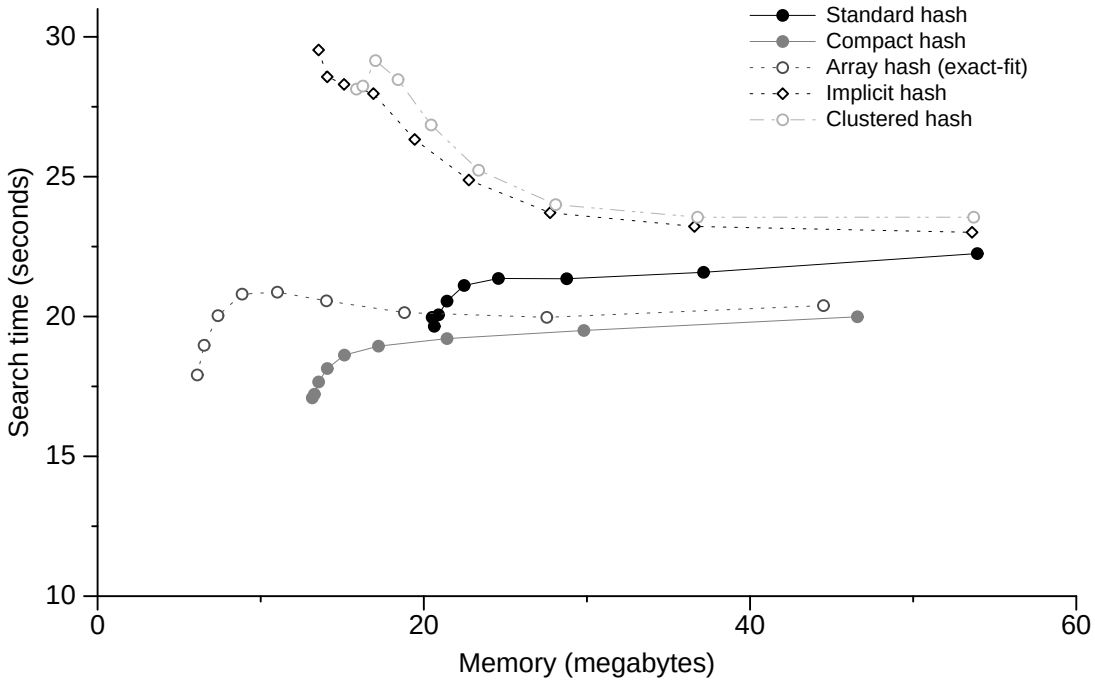


Figure 4.22: The time and space required to self-search the implicit hash table using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{16} *slots, doubling up to* 2^{24}*.*

4.8 Summary

In this chapter, we experimentally explored the performance of well-known string data structures, namely the hash table, burst trie, variants of BST, TST, the adaptive trie, and the Judy data structure. We compared the time, space, and cache performance of these data structures for the task of storing and retrieving large sets of strings in-memory. The fastest data structures were found to be the hash table, burst trie, and BST. However, our results show that the standard representation of these structures — a linked list of fixed-sized nodes consisting of a string pointer and a node pointer — is not cache- nor space -efficient. A well-known technique at improving the use of cache is clustering, which stores the nodes of a list contiguously in memory. However, the clustered representations of the hash table, burst trie, and BST, were in most cases, inferior to their standard representations and were only effective when accessed with no skew in the data distribution.

In every case, replacing the standard chain with a compact-chain — a simple alternative where the fixed-length string pointer is replaced with the variable-length string — proved faster and smaller. Eliminating the chains altogether by storing the sequence of strings in a contiguous array that is dynamically resized as strings are inserted, showed further substantial gains in performance on both current and older machine architectures. The array hash table, for example, achieved up to a 97% improvement in speed over the standard hash table. The standard hash table can often approach the speed of the array hash table, but only when given a surplus amount of space. Hence, the array hash table is a faster and scalable alternative.

Similarly, the array burst trie displayed strong and consistent improvements, being up to 89% faster than the standard burst trie. In most cases, the array burst trie approached the speed of hashing, while maintaining sorted access to containers. The array BST also displayed considerable improvements, being up to 36% faster than the standard and clustered BST. The compact BST was in most cases, marginally faster than the array BST, but requires more space.

Compared to compact chaining, dynamic arrays can yield substantial further benefits. In the best case, the space overhead of the array hash can be reduced by a factor of around 200, to less than a single bit per string, while access speed is consistently faster than under standard, clustered, and compact chaining. The array burst trie also displayed similar benefits, with space falling from around 200 bits per string, to no space overhead, while access speed remained substantially faster than the standard, clustered, and compact burst tries. The array BST also displayed similar gains, requiring only a third of the space of the standard BST, while being faster to access. These results are an illustration of the importance of considering cache in algorithm design. The standard-chain hash table, burst trie, and (in most cases) the BST, were previously shown to be the most efficient structures for managing strings, but we have greatly reduced their total space consumption while simultaneously reducing access time.

4.8.1 Recommendations for parameters

Having observed the performance of the array hash and array burst trie on different string distributions and sizes, we recommend the following parameters that should offer good — but not necessarily an optimal — performance in practice. For the array hash and assuming some skew in the data distribution, maintaining a small number of slots, typically between 2^{15} and 2^{18}, is preferable. For uniform access distributions — were typically millions of distinct strings are processed — the number of slots can be increased to reduce the average load factor. From our results, between 2^{23} and 2^{24} slots should achieve a good balance between time and space for the insertion of around 30 million distinct (plain text) strings. A further increase in slot numbers can improve speed for larger collections, but at the expense of space. For the array burst trie, our results show that a container threshold between of 128 to 256 strings should provide good performance for both skew and uniform data distributions.

Chapter 5

HAT-trie

We have explored the performance of cache-conscious or array-based data structures and found the array burst trie to be the fastest and most compact in-memory data structure, when sorted access to strings is required. The array burst trie can make excellent use of both cache and space when using large containers, that is, which store more than 128 strings. Nonetheless, large containers remain expensive to access. We learnt that in order to manage large containers efficiently, it is important to reduce both cache misses and instruction costs. In this chapter, we apply this knowledge and introduce a new dynamic string data structure, called a *HAT-trie*, that can efficiently maintain large cache-conscious containers that do not impose high instruction costs.

5.1 The HAT-trie

The array burst trie stores strings in containers that are represented as dynamic arrays. Large containers are of particular interest, as they can make excellent use of both cache and space, but are computationally expensive to access. To retain the cache-efficiency of large containers without incurring their high instruction costs, we must change the structural representation of containers from dynamic arrays to a data structure that is both cache-conscious and computationally efficient to access. It is therefore attractive to consider structuring containers as cache-conscious hash tables or array hash tables. This approach forms a novel variant of burst trie, which we call a *HAT-trie*. The advantage of using an array hash is that it can efficiently partition a large container into smaller and thus faster pieces, at little to no impact on performance.

Representing containers as a standard or compact hash table is not a viable option. Although these chaining hash tables are computationally efficient, our results have shown them to be space-intensive relative to the array hash. Furthermore, due to the use of pointers, the standard and compact hash tables can lead to poor use of cache. Figure 5.1 shows an example of the HAT-trie. We know that move-to-front on access is inefficient for arrays. Similarly, growing containers using

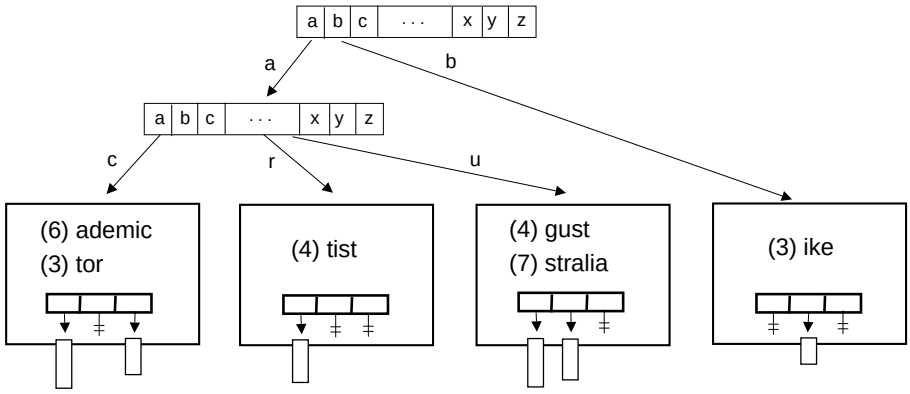


Figure 5.1: We replace the containers in a burst trie with cache-conscious hash tables to create an alternative data structure, called the HAT-trie. These hashed containers can grow much larger than their chained or dynamic array equivalents, without attracting high cache or instruction costs. As a result, the HAT-trie is a more scalable data structure. Strings are length-encoded (shown as brackets).

the 64-byte paging technique yielded only small gains in access time, and at the cost of wasting space. We therefore grow the containers of the HAT-trie using the space-efficient *exact-fit* growth technique, and we do not employ move-to-front on access.

The HAT-trie is built and searched in a top-down manner, as described for the burst trie in Chapter 3. That is, a trie node is accessed by following the pointer corresponding to the lead character of the query string. Pointer traversal will consume (or delete) the lead character of the query, and short strings that are consumed entirely are handled by setting the end-of-string flag in the respective trie node or container. When required, an empty container can be created to store the end-of-string flag. Given the situation where strings have associated data fields (which does not arise in our data), the array hash table can be easily modified to interleave data fields with strings, as discussed in Chapter 3. In addition, if a string is consumed by a trie node, the unused pointers in the trie node can store the required data fields, or alternatively, a pointer to an auxiliary data structure.

The HAT-trie begins as a single container that is populated until full. Containers do not maintain duplicates, hence a string is only inserted on search failure. To insert or search for a string in a container, the required slot is found by first hashing what remains of the string (after traversing the trie-index), using the bitwise hash function [Ramakrishna and Zobel, 1997]. Once the required slot is accessed and its associated array is fetched, the array is scanned on a per-character basis, with a mismatch prompting a skip to the next string in the array. Insertion only occurs when the string is not found or if the slot is empty. In such a case, the string is length-

Data Structure	Space overheads
Array Burst trie	$520T + 10B$
HAT-trie	$520T + (12 + 12M)B$

Table 5.1: A comparison of the space required by the array burst trie and the HAT-trie. Containers in the array burst trie reserve two bytes for housekeeping, while in the HAT-trie, containers reserve space for $M + 1$ slots (each slot requires 4 bytes). Space is also consumed by the number (T) of trie nodes, and the number (B) of containers allocated.

encoded and appended to the end of the array, by first growing (or initially creating) the array in an exact-fit manner, being the length of the string, which is cache-efficient. Deletion proceeds in a similar manner, where all strings in the array, apart from the one to delete, are copied into a new smaller array, as discussed for array burst trie in Chapter 3.

From an implementation perspective, we represent containers as an array of $M+1$ word-length pointers (where M is the number of slots used), which are empty or point to their respective slot entries; a dynamic array containing length-encoded strings. We reserve the first pointer for housekeeping information: an end-of-string flag and a counter of the number of strings in the container. For efficiency, we ensure that all pointers are word-aligned, as discussed in Chapter 3.

When a container is full, it is burst as described for the burst trie. That is, a container is burst into at most A new containers that are parented by a new trie node (A being the size of the alphabet used). The strings of the full container are then distributed amongst the new containers, according to their lead character which is then removed. Strings can be consumed during the bursting procedure, and are handled by setting the end-of-string flag in the respective container.

5.2 Disadvantages of the HAT-trie

The HAT-trie has two major disadvantages when compared to the array burst trie. First, containers must reserve space for a fixed number of slots. As a consequence, some space is wasted when containers store a few strings. It would be more space-efficient in this case to represent an under-loaded container as a single dynamic array that is changed into array hash when full. Alternatively, containers can maintain a variable number of slots, altered according to the frequency of access. This can save space but at the cost of re-hashing containers. Although not as space-efficient as the array burst trie, the HAT-trie can still make efficient use of space by maintaining large containers, as shown in later experiments.

We compare the space overhead between the array burst trie and the HAT-trie in Table 5.1. Both the array burst trie and HAT-trie maintain array-based trie nodes. That is, each trie node is structured as an array of 128 four-byte pointers that map directly to the 128 characters of the ASCII table. The first pointer of each trie node is reserved for the end-of-string flag; pointers

that map to non-printable characters are unused. Hence, the total space required by a trie node is 520 bytes, which includes the 8-byte allocation overhead imposed by the operating system. In practice, we allocate trie nodes contiguously by storing them in a single dynamic array, which is both cache- and space -efficient.

Every container in the HAT-trie stores $M + 1$ 4-byte slots, where M is the number of slots. Each slot (except the first, which is used for housekeeping) will incur a further 8 bytes of allocation overhead when assigned to a dynamic array. The space overhead of a container is therefore between $4M + 4$ and $12M + 4$ bytes. In addition, an 8-byte allocation overhead is also imposed when a container is initially created, which raises the total space overhead to at most $12M + 12$ bytes. Although these containers consume more space than those of the array burst trie, there is a compensating trade off, a reduction in the number of trie nodes and containers allocated; that is, they can be burst less often at little to no cost in performance, as shown in later experiments.

The second disadvantage is that access to a container involves two pointer traversals: the first to access the required slot, and the second to fetch the slot entry — a dynamic array. As containers and their dynamic arrays are scattered in main memory, this is likely to incur two cache misses (and potentially two TLB misses if the slots and the dynamic arrays are not stored in the same page). The array burst trie, in contrast, incurs at most, only a single L2 and TLB-miss on container access. As a result, enough slots must be allocated in the containers of the HAT-trie to compensate for the cost of traversing a second pointer. That is, the average load factor of containers must be kept low by allocating more slots, but at the cost of space. Hence, choosing the number of slots to allocate in containers requires some care, to avoid wasting large volumes of space. We discuss how we selected the number of slots used in our experiments in Section 5.5.

5.3 Expected cache costs

We theoretically analyze the cache performance of the HAT-trie, to gain a better understanding of the expected reductions in cache misses. We simplify our calculations by assuming the following:

- Use of an ideal cache, as described in Section 3.4.
- The HAT-trie is many times larger than cache.
- A uniform data distribution, that is, there is no skew.
- Strings are of uniform length and are randomly generated; each character is randomly selected from the English alphabet. There are no string duplicates.
- The hash function used is from a universal class [Sarwate, 1980] so that strings are distributed as well as possible, regardless of data distribution. The best hash function for strings that is thought to be from a universal class is the bitwise hash method [Ramakrishna and Zobel, 1997].

Notation	Meaning
N	Number of strings inserted
C	Cache-line size in bytes
P	Page size in bytes
L	Length of the strings (includes null character)
A	The size of the alphabet used
W	The size of a pointer in bytes
B	The maximum number of strings in a container
M	The number of slots per container
α	The average load factor: B/M

Table 5.2: Notation used for analyzing the expected cache costs of the HAT-trie.

The HAT-trie maintains the expected logarithmic access costs of the burst trie [Heinz et al., 2002; Knessl and Szpankowski, 2000; Szpankowski, 1991]. We assume that every trie node accessed incurs a cache-miss. Hence, the expected cache misses on traversal is up to:

$$\log_A \left(\frac{N}{B} \right) \tag{5.1}$$

However, this only captures the cache complexity of traversing trie nodes. Once a container is accessed, we assume that a further two cache misses are incurred; the first when a slot is accessed, and the second when the slot pointer is followed, to acquire its slot entry (a dynamic array). All subsequent cache misses are incurred on array access. Hence, the expected L2 cache misses on search is up to:

$$\log_A \left(\frac{N}{B} \right) + 2 + \frac{L\alpha}{C} \tag{5.2}$$

Similarly, the expected number TLB misses on search is up to:

$$\log_A \left(\frac{N}{B} \right) + 2 + \frac{L\alpha}{P} \tag{5.3}$$

The cache-efficiency of the HAT-trie can be improved by increasing the size of B, which will reduce the number of trie nodes accessed. The cache-efficiency of containers can be improved by reducing the average load factor or α. Inserting or deleting a string in the HAT-trie can attract further cache misses, due to involvement of array copying (resizing). Assuming that *exact-fit* is used, the

slot entry (the dynamic array) to be updated is scanned twice; once for search, and then again for copying. Hence, the expected L2 cache cost on update — assuming that a burst does not occur — is up to:

$$\log_A \left(\frac{N}{B}\right) + 2 + 2\left(\frac{L\alpha}{C}\right) \tag{5.4}$$

Similarly, the expected number TLB misses on update is up to:

$$\log_A \left(\frac{N}{B}\right) + 2 + 2\left(\frac{L\alpha}{P}\right) \tag{5.5}$$

5.4 Comparing expected to actual cache performance

We compare the expected cache misses for searching the HAT-trie as discussed in Section 5.3, to the actual cache misses incurred on our primary machine, a Pentium IV. Our cache analysis in Section 5.3 assumes an ideal cache, and is thus not intended to accurately predict the number of cache misses incurred when traversing the HAT-trie. Instead, our cache analysis highlights the expected cache costs as we vary parameters such as the container size B and the number of slots M.

To conduct the experiment, we generated a random dataset consisting of 10 million unique 16-byte strings, by selecting each character at random from the English alphabet. The random dataset was then used to construct the HAT-trie. We then measured the actual L2 and TLB misses incurred during search, by using a 16-byte random string that was not stored in the HAT-trie. Hence, the search is guaranteed to fail and incur the full cost of traversal. We repeat the search 200 times and flush the cache after each run. We varied the container size B by 1024, 4096, 8192, and 16,384 strings and the number of slots M by 16, 64, and 512. We then calculated the expected cache and TLB misses as M and B are varied on an ideal cache, by using equations 5.2 and 5.3 of Section 5.3. The results are compared in Table 5.3 and Table 5.4. The number N of strings inserted is 10 million, and each string has a length L of 16 bytes. The cache-line size C is 128 bytes and the alphabet size A is 26.

As we varied M and B, the expected cache costs for searching the HAT-trie accurately depicted the cache behavior observed in practice. That is, as the size of containers increased, the actual number of cache misses also increased. Similarly, increasing the number of slots used by containers from 16 to 512, led to a reduction in actual cache misses, as expected. However, the number of cache misses calculated on the ideal cache are obviously not comparable to those incurred on the

Container size (no. of strings)	Expected Cache misses			Actual L2 misses		
Slots	16	64	512	16	64	512
1024	13	7	5	18	16	16
4096	37	12	5	19	17	15
8192	68	20	6	20	18	17
16384	132	36	8	22	18	16

Table 5.3: The expected and actual cache misses for searching the HAT-trie (which contains 10 million random fixed-length strings) for a string that is not stored. The actual values are averaged over a series of 200 runs.

Container size (no. of strings)	Expected TLB misses			Actual TLB misses		
Slots	16	64	512	16	64	512
1024	5	5	5	9	8	8
4096	5	5	4	9	8	8
8192	6	5	4	10	9	9
16384	8	5	4	12	9	5

Table 5.4: The expected and actual TLB misses for searching the HAT-trie (which contains 10 million random fixed-length strings) for a string that is not stored. The actual values are averaged over a series of 200 runs.

Pentium IV, which employs a multi-layer cache hierarchy and hardware prefetch (both of which are not considered in an ideal cache). A multi-layer cache combined with hardware prefetch can reduce more cache misses than a single layer of cache. Furthermore, traversing a container involves scanning an array, which makes good use of hardware prefetch that can lead to further reductions in cache misses. As a result, the HAT-trie with a container size of 16,384 strings and 16 slots, incurred only 22 cache misses on search, instead of the 132 cache misses calculated on the ideal cache.

There are cases, however, where the number of cache misses incurred on an actual cache can exceed those calculated on an ideal cache. An example is shown in Table 5.3, with a container size of 1024 strings and 512 slots. The ideal cache model assumes exclusive access to cache, which eliminates interference caused by other programs and hence, potential cache misses. Furthermore, the ideal cache assumes an optimal cache-line eviction policy, to minimize cache misses. In an actual cache system, however, it is often impossible to ensure exclusive access and a non-optimal eviction policy, such as random or approximate LRU (least recently used), is often used [Shanley, 2004]. These factors can lead to an increase in cache misses, as observed.

Another important factor is the hash function. In the cache analysis of Section 5.3, we have assumed that the hash function can evenly distribute strings among the slots of a container. In practice, however, this is not the case. Some slots will contain more strings than others and some can be empty, particularly when only a few strings are stored in a container. Hence depending on the slot accessed, more strings can be compared during search than expected. For example, with a container size of 16,384 strings and 512 slots, we assume that each slot contains 32 strings, but in practice, 52 strings were compared, which can increase the number of cache misses incurred.

Similar results were observed when comparing the expected and actual TLB costs. That is, as the container size increased, the actual number of TLB misses also increased, and an increase in the number of slots used by containers also led to a reduction in TLB misses. However, the number of TLB misses incurred in practice was consistently higher than those incurred on an ideal cache, due to the limited number of TLB entries available, and the resource contention caused by other (mandatory) programs administrated by the operating system.

Our next experiment compares the expected and actual cache costs for inserting a string into the HAT-trie. We repeat the previous experiment by building the HAT-trie using the random dataset, and then conduct a series of searches using the random 16-byte string that does not exist in the HAT-trie. Once the search fails, however, we insert the new string into the container acquired. Hence, the insertion procedure incurs the cost of search plus the additional cache misses caused by resizing the array (the slot entry) in a container. We calculate the expected cache misses using equation 5.4 of Section 5.3, and compare it to the actual cache misses shown in Table 5.5. As assumed in Section 5.3, no bursting occurs.

The key difference between searching for a string and inserting a string, is that we expect the number of cache misses for insertion to be up to double that of search, because the array (the slot

Container size (no. of strings)	Expected Cache misses			Actual L2 misses		
Slots	16	64	512	16	64	512
1024	21	9	5	32	28	22
4096	68	20	6	35	25	19
8192	132	36	8	40	37	21
16384	260	68	12	48	40	31

Table 5.5: The expected and actual cache misses for inserting a new randomly generated string into the HAT-trie (which contains 10 million random fixed-length strings). The actual values are averaged over a series of 200 runs.

entry) is scanned twice: once for search, and another for copying it into a new larger array. The actual cache costs for inserting a string into a HAT-trie, as we varied M and B, were consistent to the expected cache costs. That is, large containers incurred more cache misses, which can be reduced by increasing the number of slots allocated. Furthermore, inserting a string into the HAT-trie incurred around double the cache misses of search, as expected. Similar results were observed for the TLB.

In the discussions that follow, we experimentally evaluate the performance of the HAT-trie against the cache-conscious hash table, burst trie, and BST, by using large string collections that we acquired from real-world sources.

5.5 Experimental design

We experimentally compare the performance of the HAT-trie against the array hash table, the array burst trie, and the array BST, by measuring the time and space required for construction and self-search. When the same dataset is used for both construction and search, the process is called a *self-search*. We also compare the performance of the HAT-trie against the compact-chain representations of the hash table, burst trie, and BST. Our measure of time represents the average elapsed (total) time derived from a series of ten runs. After each run, main memory was flooded with random data to flush system caches. Our measure of memory takes into account the overhead imposed by the operating system. In our case, this overhead was 8 bytes per memory allocation, which we found to be consistent after comparing our measure against that reported by the operating system under the `/proc/stat/` table.

The datasets used consisted of null-terminated variable-length strings that appear in occurrence order; they are, therefore, unsorted. The characteristics of the datasets are shown in Table 5.6. The DISTINCT dataset was acquired after parsing the GOV2 web crawl. Distributed as part of the

Dataset	Distinct strings	String occs	Average length	Volume (MB) of distinct	Volume (MB) total
DISTINCT	28772169	28772169	9.59	304.56	304.56
TREC	612219	177999203	5.06	5.68	1079.46
URLS	1289459	9999425	30.93	46.62	318.87

Table 5.6: Characteristics of the datasets used in our experiments.

TREC project, it does not contain duplicates. The highly skew TREC dataset is the complete set of word occurrences, with duplicates, from the first of five TREC CDs [Harman, 1995]. The URLS dataset, also extracted from TREC web data, consists of complete URLs with duplicates.

The experiments were conducted on our primary machine: a 2.8 GHz Pentium IV, with 512 KB of L2 cache, 128-byte cache-lines, a TLB capacity of 64 entries, 2 GB of RAM and a Linux operating system under light load. Further information is provided in Table 4.2. We also evaluated the performance of the HAT-trie on different machine architectures, namely a 3 GHz Intel Xeon processor and a 700 MHz Intel Pentium III processor with no hardware prefetch. We omitted the results from the Xeon processor, as they were found to be similar to our primary machine. We also conducted experiments on a Sun Ultra Spark server running a Solaris operating system, the results of which are provided in the Appendix.

We compared the L2, TLB, and instruction costs incurred by the HAT-trie during self-search, against the variants of hash table, burst trie, and BST, using PAPI [Dongarra et al., 2001], a software that obtains the actual number of cache misses and instruction executed. In the discussions that follow, we omit the results of cache-performance during construction, as they were similar to those of search.

To measure the impact of the average load factor on the array and compact hash tables, we commenced with 2^{15} slots which we doubled to 2^{27} or until a minimum execution time was observed. The container size or threshold (the number of strings a container needs to trigger a burst) for the compact burst trie was increased by intervals of 10 from 30 to 100. For the array burst trie, we extended the sequence to include 128, 256, and 512 strings. Move-to-front on access was disabled for the array-based data structures — including the HAT-trie — but enabled for the chaining hash tables and burst tries, a configuration which was found to offer the best performance.

We varied the container thresholds of the HAT-trie from 2048 strings, doubling up to 32,768 strings. To determine the number of slots to use, we conducted a preliminary experiment to compare the time and space required by the HAT-trie to self-search the DISTINCT and TREC datasets. We fixed the container size to 32,768 strings and varied the number of slots from 2^5, doubling up to 2^{13}. The results are shown in Figure 5.2.

Our goal was find the maximum number of slots that can keep the overall space consumed by the HAT-trie to within the total space required by the strings. The space required by the unique

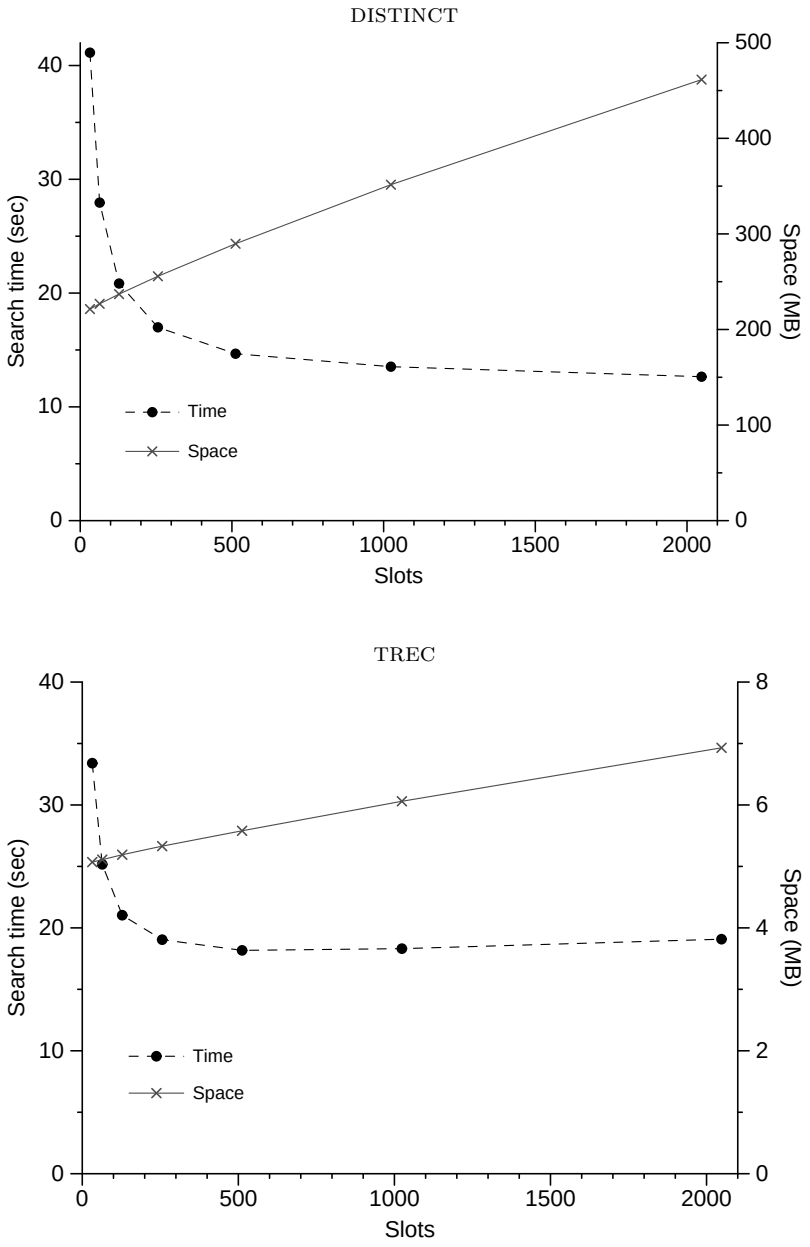


Figure 5.2: The time (in seconds) and space (in megabytes) required to self-search the HAT-trie using the DISTINCT *and* TREC *datasets. In this experiment, we fixed the container threshold to* $32,768$ *strings and varied the number of slots used from* 2^5*, doubling up to* 2^{13}*. Increasing the number of slots consumes more space.*

strings in the DISTINCT and TREC datasets is 304.5 MB and 5.6 MB, respectively. From the results in Figure 5.2, the HAT-trie consumed under 300 MB using the DISTINCT dataset and 2^9 slots (the fifth point from the left), with an access time that was almost the fastest. Similarly, using the TREC dataset and 2^9 slots, the HAT-trie required under 5.6 MB while achieving its fastest self-search time. Hence, in the experiments that follow, we allocate 2^9 slots in the containers of the HAT-trie.

5.6 Results

We now describe the results and observed trends for each collection.

5.6.1 Distinct data

The DISTINCT dataset shows two important properties: there is no skew and a large number of strings need to be managed. Such a collection is not particularly trie-friendly. Figure 5.3 and Figure 5.4 show the effects on memory and speed, as the container thresholds of the tries and the number of slots used by the hash tables are varied during construction and self-search, respectively. Small containers are split often, resulting in a large number of tries and containers. Large containers are split less frequently and are therefore more space-efficient.

The compact burst trie incurred high instruction and cache misses which resulted in its poor performance relative to the array burst trie and HAT-trie. The array burst trie also incurred high instruction costs but compensated with a high reduction in L2 and TLB misses, which led to substantial savings in access time. Nonetheless, as the size of containers increased from 128 to 512 strings, the computational cost of traversing containers grew, until it could no longer be masked by the reduction in cache misses. As a consequence, the array burst trie became slower to access relative to its smaller and less cache- and space -efficient containers. The impact on access time is more apparent during construction, due to added computational overhead of resizing arrays; array resizing is cache-efficient.

The HAT-trie can sustain containers of up to 32,768 strings efficiently, due to the reduction in both cache misses and instructions costs. As a result, the HAT-trie showed little variance in the speed of construction and self-search, as the container threshold varied. For example, with a container threshold of 2048 strings, the HAT-trie required about 21.9 seconds to build and 13.7 seconds to search, while consuming around 891 megabytes of memory. Containers that were sixteen times larger increased the build time by less than 14 seconds, and the self-search time by less than a second, while reducing space to only 290 MB; this is less space than required by the strings alone. Hence, although the HAT-trie was not as space-efficient as the array burst trie, by using large containers it can rival both the speed and space efficiency of the array hash, while maintaining sorted access to containers. At its best, the HAT-trie was up to 22% faster to access than the array burst trie while requiring around 332.1 MB of space; this is a space overhead of

under 8 bits per string.

These results are in contrast to the compact burst trie, which can only achieve its best time using smaller containers, and best space using larger containers. Either way, the compact burst trie was too expensive to access, due to its inefficient use of cache. The hash tables remained the fastest data structures but only when given enough space, which proved excessive for the compact hash table, which required at least 700 MB to surpass the speed of the HAT-trie.

The compact and array BSTs were considerably slower to access than both the tries and hash tables. The array BST required over 470 seconds to construct and self-search using 534 megabytes of memory, while the compact BST required over 570 seconds to construct and self-search, consuming 764 megabytes of memory. Figure 5.5 shows the L2, TLB, and instruction costs of the HAT-trie, array hash, and the burst tries during self-search. There is a strong correlation between cache misses and instruction costs, to the overall search time. The array hash, for example, performed poorly under heavy load because it incurred high L2 misses, TLB misses, and instruction costs. As we add more slots to reduce the average load factor, the slot entries (the dynamic arrays) became smaller and thus cheaper to access, and have improved probability of cache residency.

The HAT-trie incurred fewer L2, TLB, and instructions per search than the array burst trie, and showed little to no impact on cache performance as the container threshold varied. As the container threshold increased from 16,384 strings to 32,768 strings, however, the HAT-trie was slightly slower to build than the array burst trie, due to the high computational overhead caused by resizing containers that are under heavy load. We know from the results in Section 5.5, that reducing the average load factor of containers — by adding more slots — will only yield small improvements in access time, which will not justify the increase in space. These results show us that although the HAT-trie reduces both cache misses and instruction costs, containers that store more than 16,384 strings can still impose high computational costs on access which cannot be masked by the reduction of cache misses. Hence, the HAT-trie cannot sustain larger or unbounded container sizes without incurring a high impact on access speed.

Compared to the array burst trie, an increase in memory usage (that is, by using smaller containers) does not have a positive impact on the performance of the HAT-trie, apart from the slight improvement in build time. Although containers incur fewer instructions when they are small, maintaining a large number of tries and containers can ultimately reduce cache-efficiency, as fewer nodes are likely to remain within cache. Furthermore, with an increase in the number of trie nodes, the cache will likely be flooded with tries on search, overwriting previously cached containers. Temporal access locality is unlikely to be as good for the HAT-trie, as it is for the hash tables. Every trie node encountered during search branches to a new location in memory. Trie nodes (apart from those close to the root) are therefore less likely to be visited frequently, especially as their numbers increase. The number of conversions of virtual to physical addresses — TLB misses — is likely to increase as a result. Hence, use of large containers (around 8192 to 16,384 strings) is valuable to the HAT-trie, as it results in better use of cache and space.

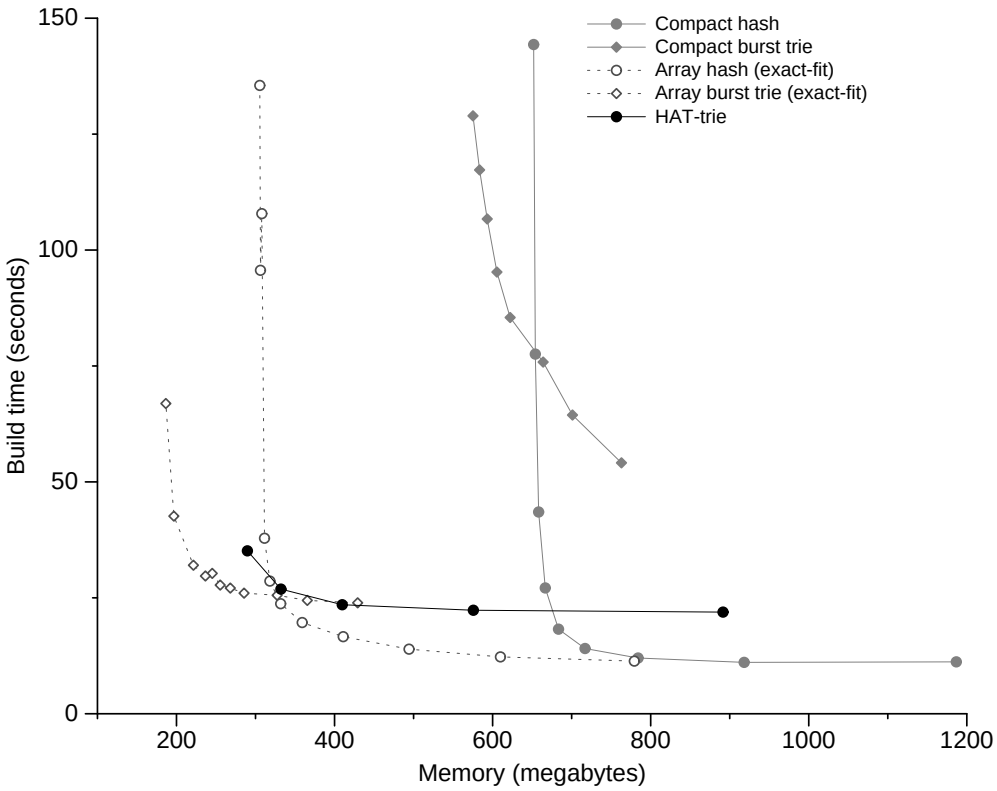


Figure 5.3: The time and space required to build the data structures using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

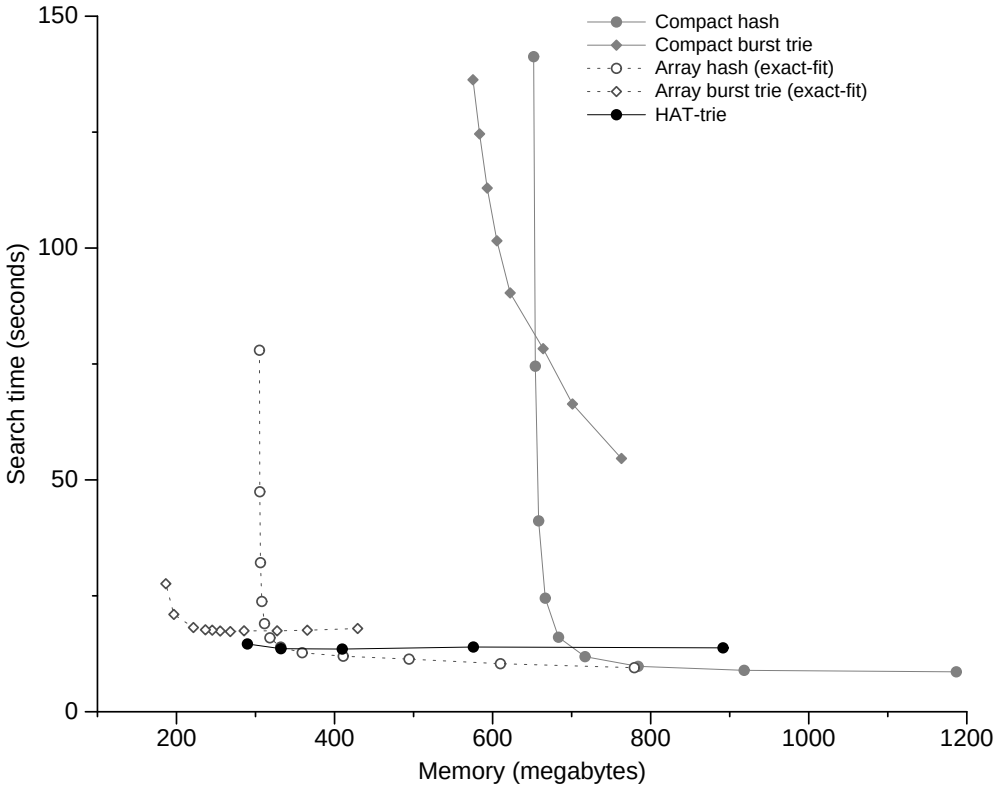
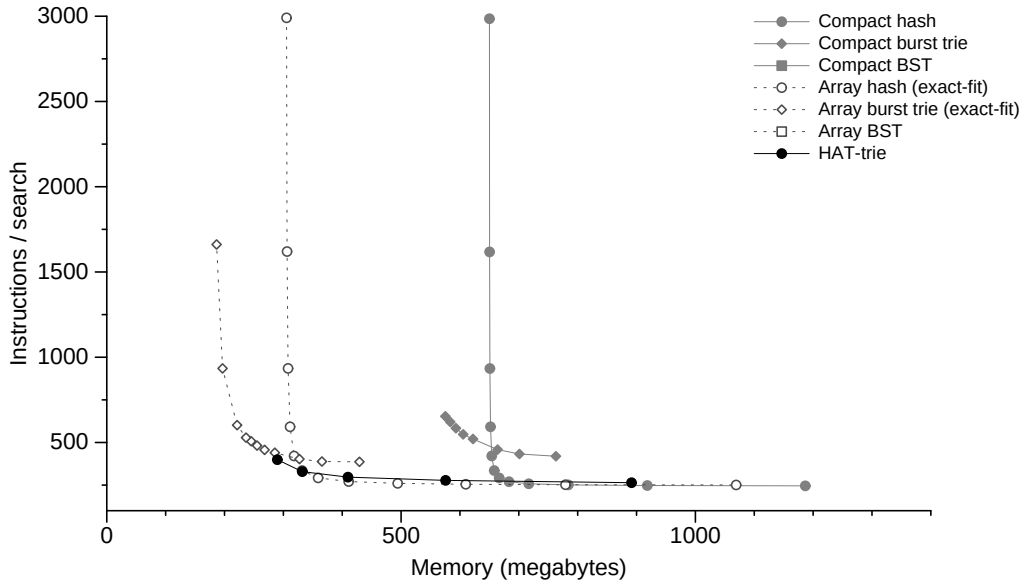
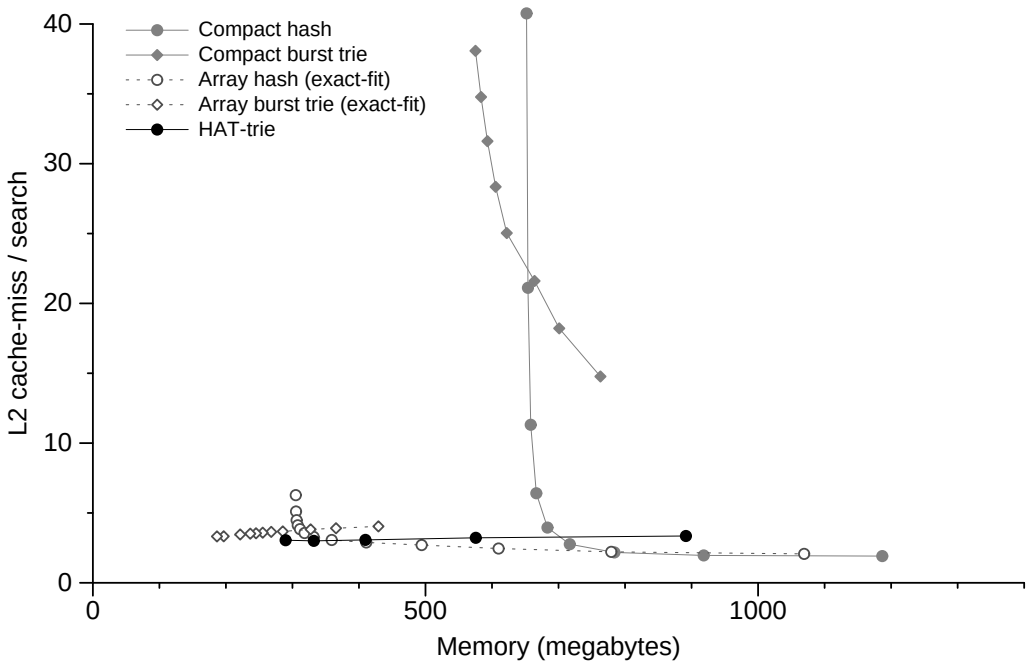


Figure 5.4: The time and space required to self-search the data structures using the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* 32,768*.*

(a) Instructions per search



(b) L2 cache-miss per search



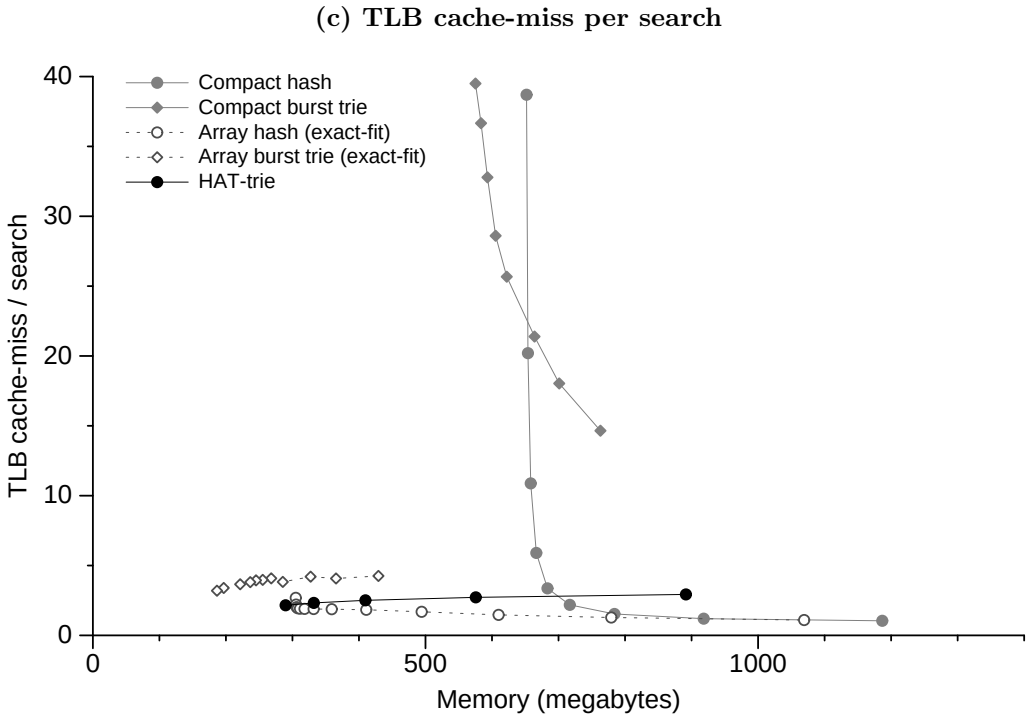


Figure 5.5: The Instructions (a), L2 cache misses (b), and TLB misses (c) incurred by the data structures when self-searching the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

5.6.2 Skewed data

Our next experiment evaluates cost where some strings are accessed much more frequently than others. We repeated the previous experiment using the TREC dataset; the results for construction and self-search are shown in Figure 5.6 and Figure 5.7, respectively.

With just over half a million distinct strings, the data structures were much smaller than those constructed previously. The array hash was at its best when under heavy load — about 19 strings per slot (or 2^{15} slots). Although the slot entries (the dynamic arrays) were large, 99% of the searches need only access the first string of each slot. Thus, the usual assumption — that load average is a primary determinant of speed — does not always hold. The compact hash was slightly faster than the array hash due to the use of move-to-front on access, but required more than double the space.

The array burst trie was the most space-efficient data structure, but it was never faster to build or self-search than the HAT-trie, which offered up to a 15% improvement in speed. Furthermore, the HAT-trie could almost match the speed of the array hash while requiring less space. The BSTs were the most expensive data structures to access, being over 30 seconds slower to build and to self-search than the HAT-trie. Figure 5.8 shows the cache and instruction costs of these data structures during self-search. The BSTs — the array BST in particular — made good use of a cache but remained expensive to access due to the high computational cost of binary search. With a container size of 32,768 strings, the HAT-trie incurred fewer L2 and TLB misses than the array and compact hash tables, but executed more instructions due to the traversal of trie nodes. With a container threshold of 512 strings, the array burst trie incurred the least amount of L2 cache misses, but was slower than the HAT-trie due to higher instruction and TLB costs.

In previous experiments involving the DISTINCT dataset, the HAT-trie became slightly slower to access — relative to its smaller containers — as the threshold increased from 8192 to 32,768 strings; the cause being the high computational cost of access which outweighed the time saved by the reduction in cache misses. In these experiments, only 612,219 strings were inserted and as a result, most containers were under-loaded and were thus computationally efficient to access. These results demonstrate that both the load factor of containers, as well as the total number of strings stored, can affect the access speed of the HAT-trie when large containers are used.

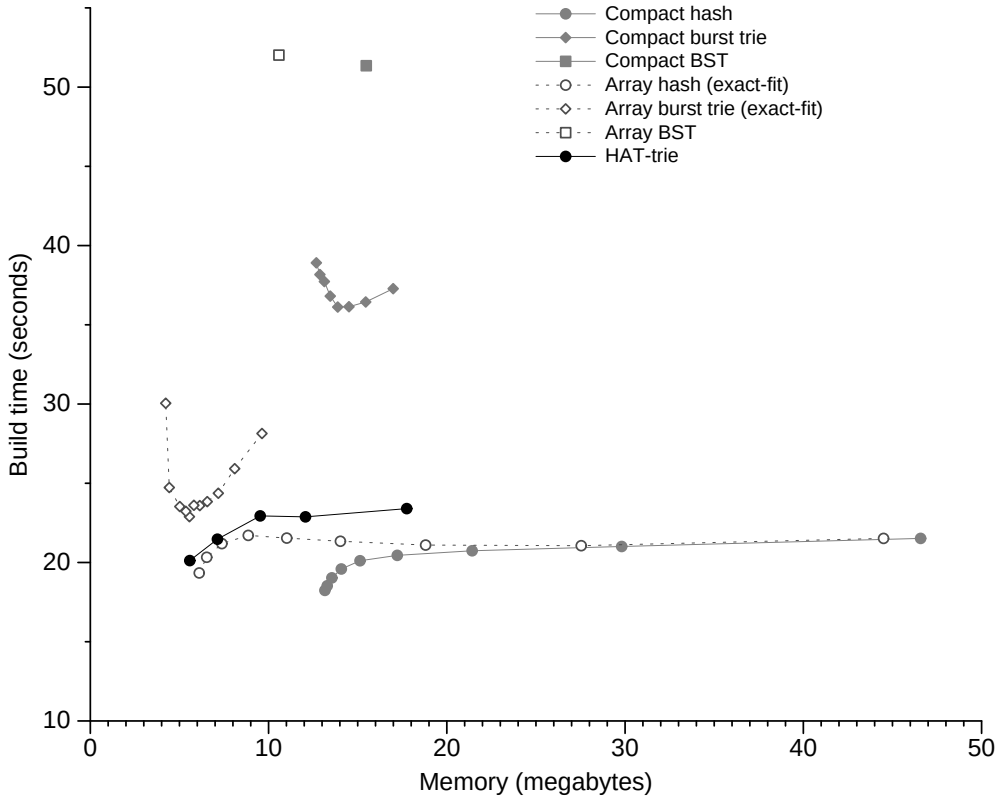


Figure 5.6: The time and space required to build the data structures using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

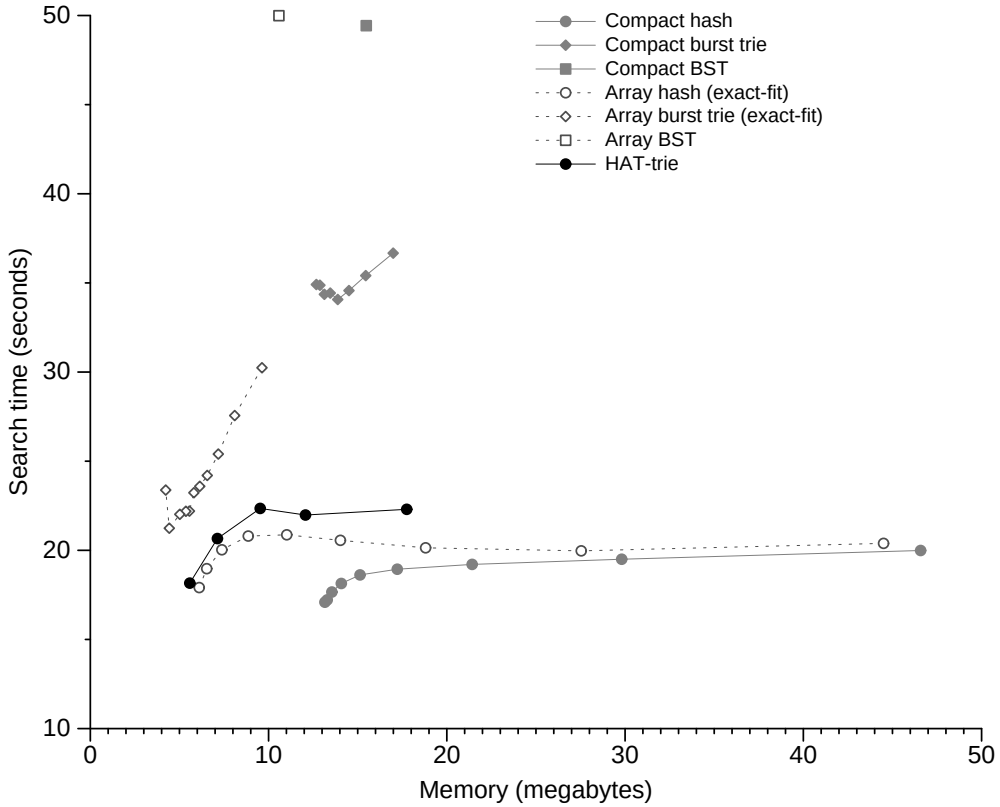
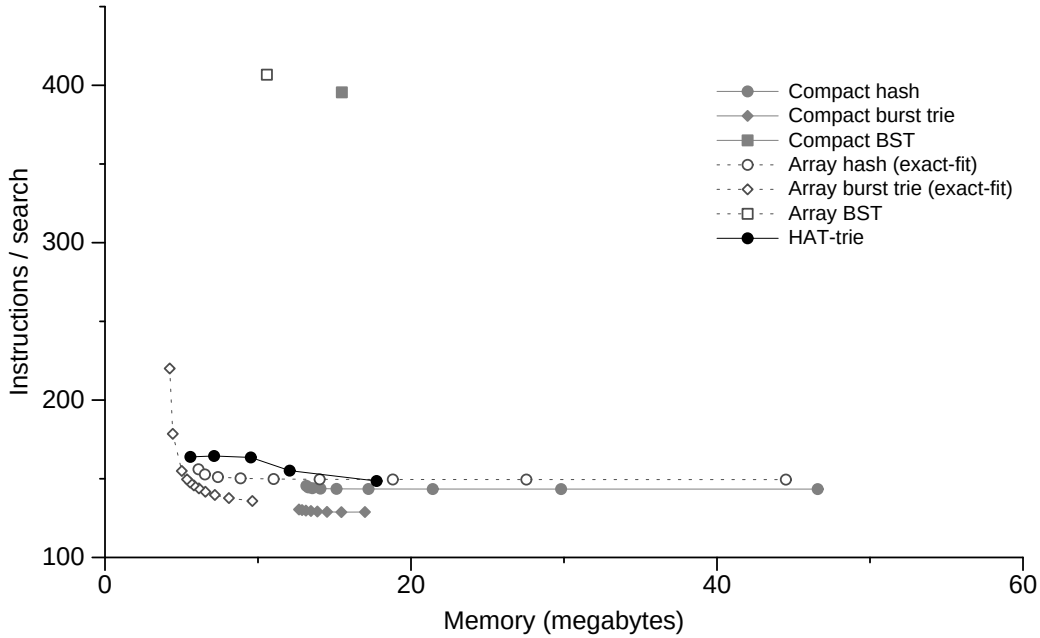
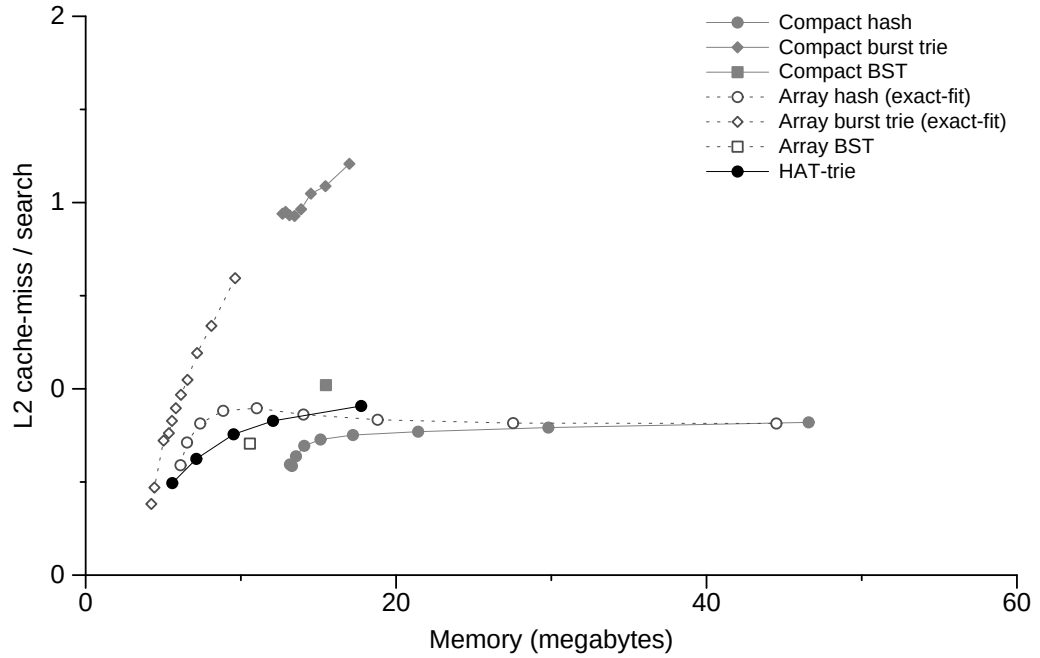


Figure 5.7: The time and space required to self-search the data structures using the TREC *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

(a) Instructions per search



(b) L2 cache-miss per search



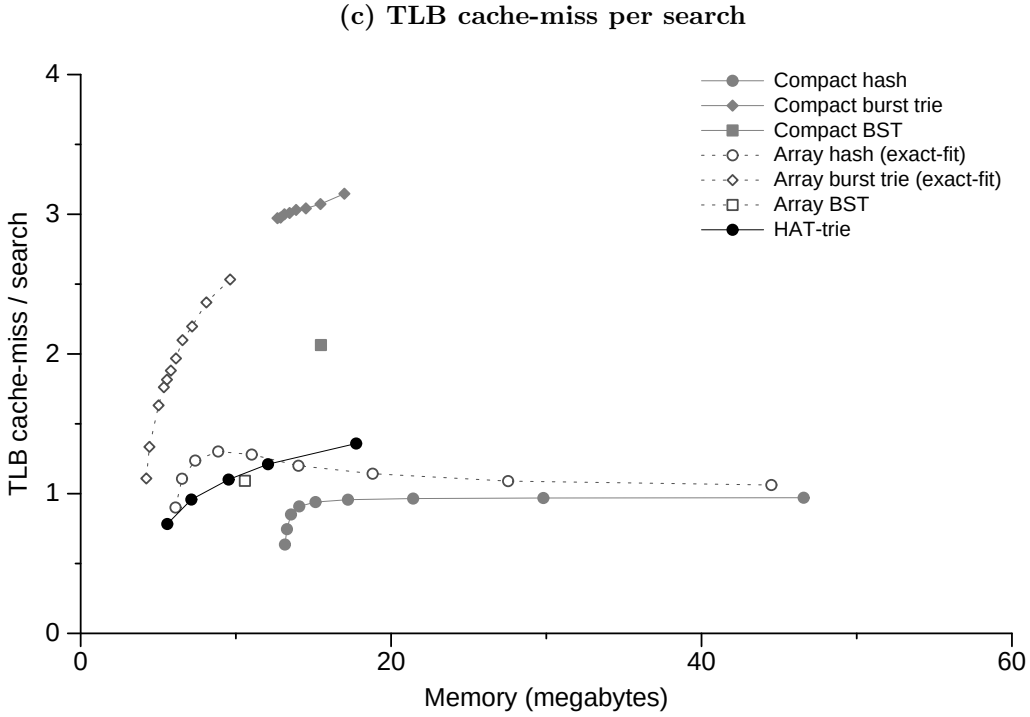


Figure 5.8: The Instructions (a), L2 cache misses (b), and TLB misses (c) incurred by the data structures when self-searching the TREC *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

5.6.3 URL data

The URLS dataset is highly skew and contains long strings, some in excess of a hundred characters. URLs typically share long prefixes; most strings start with "http://www". As a result, string comparisons can be more expensive. The results of construction and self-search are shown in Figure 5.9 and Figure 5.10, respectively.

The tries have the advantage of stripping away shared prefixes during traversal, which can save both time and space. Nonetheless, the array and compact BSTs remained faster to build than the array burst trie, and were almost as fast to self-search. The cache performance of these data structures during self-search is shown in Figure 5.11. The BSTs were efficient to access because they rivaled the cache-efficiency of the array burst trie, which compensated for the high number of instructions executed. The URLS dataset contained just over a million distinct strings. As a result, the performance of the BSTs can be attributed to their small size. Given a high increase in the number of strings stored or accessed, their performance will deteriorate relative to the array burst trie, due to the high computational cost of binary search — as observed in previous experiments involving the DISTINCT and TREC datasets.

The HAT-trie displayed strong and consistent improvements over the array burst trie, being up to 50% faster to build and self-search. Furthermore, the HAT-trie could almost match the speed of the array hash while requiring less space. The HAT-trie achieved its fastest self-search time of 4.9 seconds using a container size of 16,384 strings and only 44.2 MB. Although not as space-efficient as the array burst trie, the HAT-trie achieved its best time while consuming less space than required the strings alone. The hash tables were the fastest data structures to build and self-search, but only when given enough space.

As the container threshold varied, the HAT-trie consistently incurred fewer L2 and TLB misses than the array burst trie, which reflected the timings of Figure 5.10. However, as the container threshold increased from 8192 to 32,768 strings, the number of instructions executed also increased until their cost could no longer be masked by the reduction in cache misses. As a consequence, the HAT-trie was more expensive to access using large containers, relative to its smaller and less space- and cache -efficient containers. The impact on access time is more apparent during construction, due to the added computational cost of array resizing which can be expensive with long strings. Nonetheless, the increase in instructions remained substantially less than that of the array burst trie.

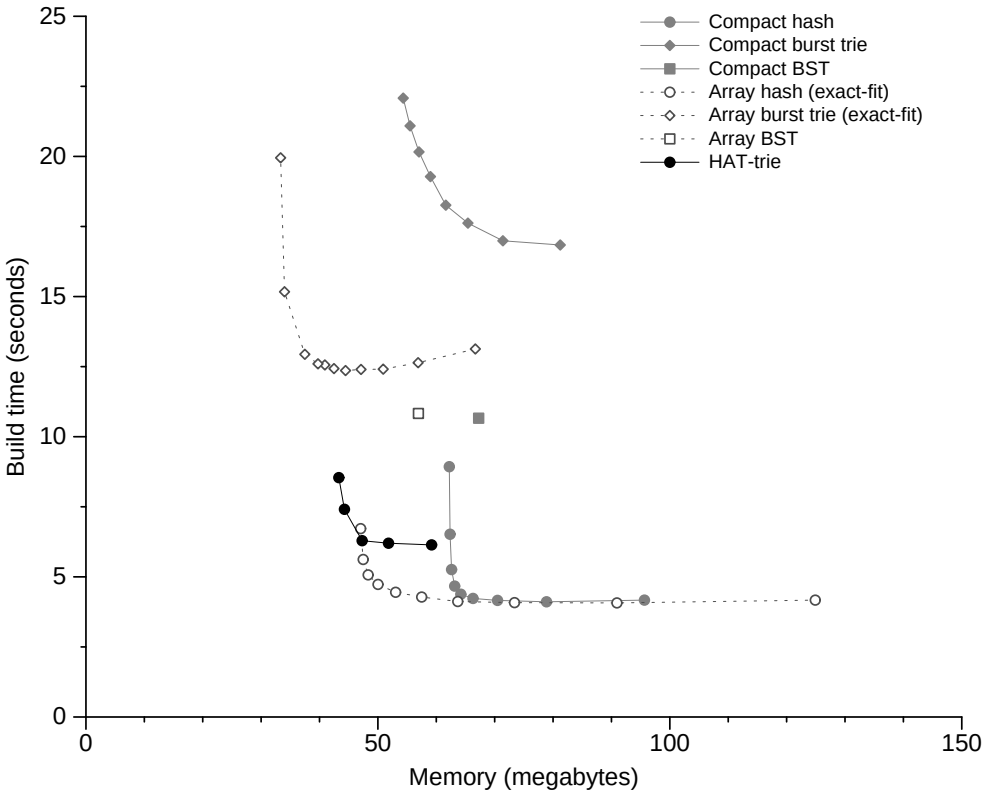


Figure 5.9: The time and space required to build the data structures using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

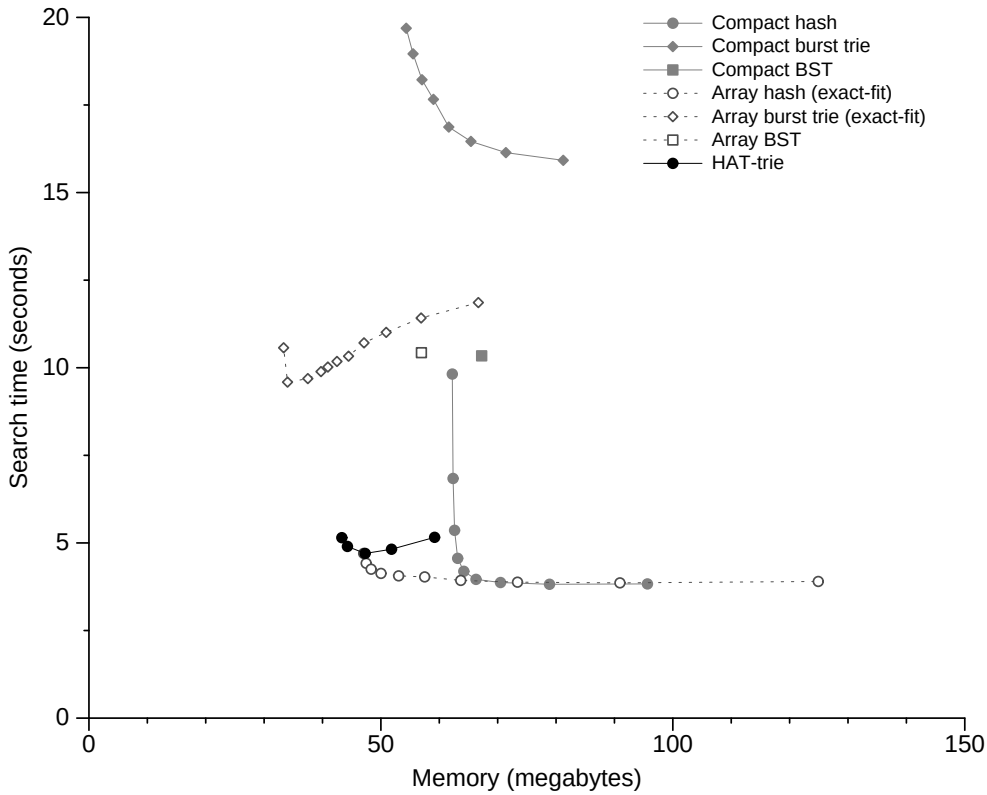
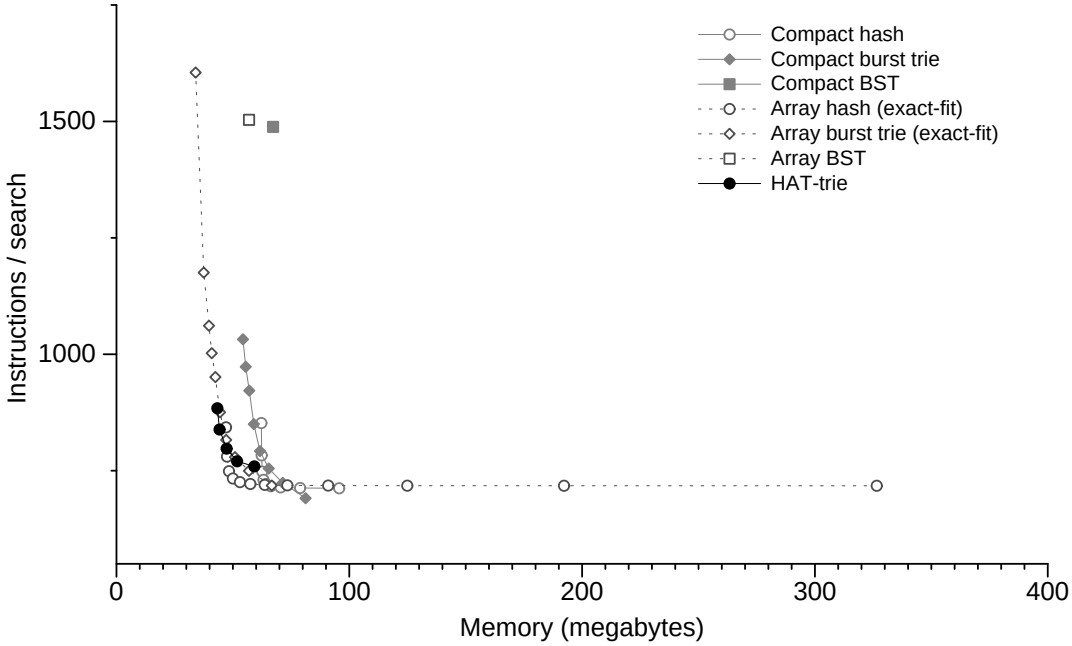
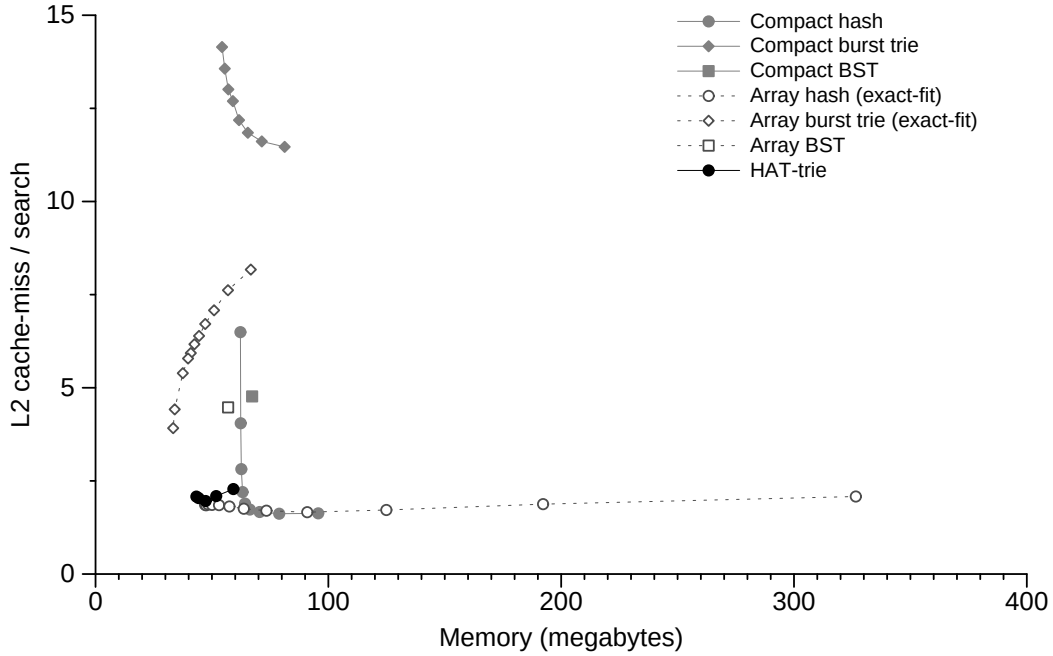


Figure 5.10: The time and space required to self-search the data structures using the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

(a) Instructions per search



(b) L2 cache-miss per search



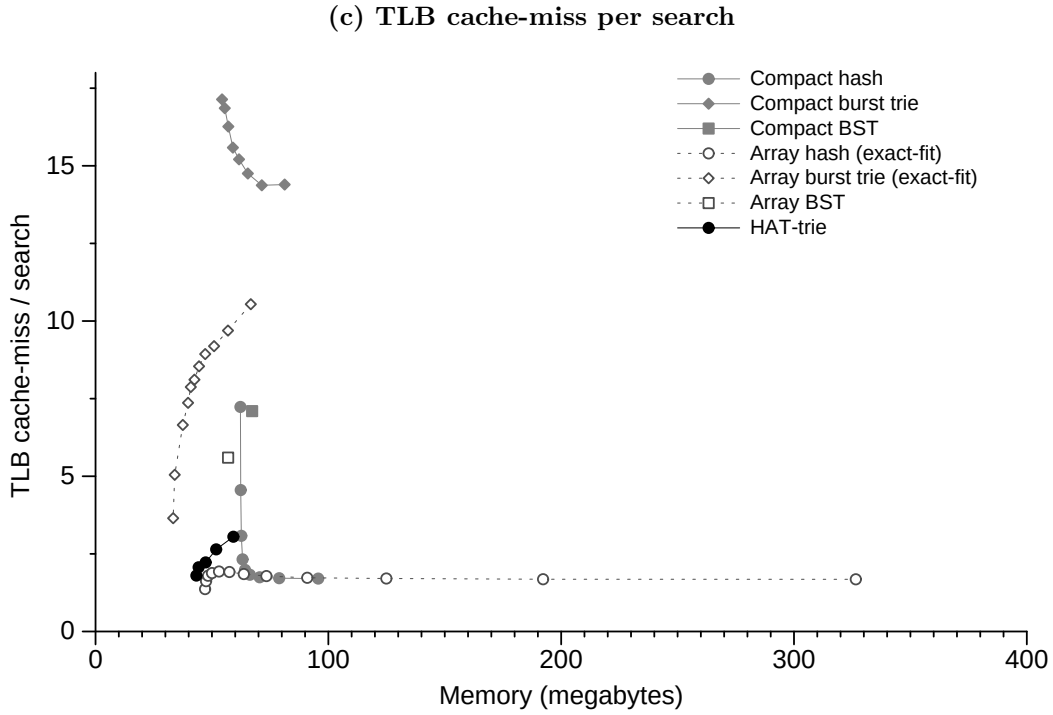


Figure 5.11: The Instructions (a), L2 cache misses (b), and TLB misses (c) incurred by the data structures when self-searching the URLS *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence was extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

5.6.4 Skew search on large data structures

Our next experiment evaluates the skew performance of the HAT-trie when grown to a much larger size. That is, we build the data structure using the DISTINCT dataset, and then measure the time required to search using the TREC dataset. The time and space required for search is shown in Figure 5.12. Unlike the previous TREC experiments, some searches fail — 1,247,782 in total — and thus incur the full cost of traversal.

The compact hash table was the fastest data structure to access, but only when given enough space. The array hash was almost as fast, but used less than half the space of the compact hash table. The array burst trie was the most space-efficient data structure, and the compact burst trie displayed poor performance due to its poor use of cache and high instruction costs. The array and compact BSTs made good use of cache, but were nonetheless expensive to access due to the high computational cost of binary search.

The HAT-trie was consistently faster to search than the array burst trie, but was not as space-efficient. In the previous TREC experiments, the HAT-trie displayed consistent improvement in speed, as the size of containers increased. Similar behavior was observed in this experiment, with the HAT-trie improving in speed as the container threshold increased from 2048 to 16,384 strings. However, unlike the previous TREC experiments, a container threshold of 32,768 strings was slower to access than smaller containers; the result of storing more strings in containers, which when kept under heavy load, will incur more cache misses and instruction costs.

Similarly, the array burst trie can not sustain containers that exceed a capacity of around 512 strings without experiencing an impact on access time — caused by high computational costs. Hence, in order to sustain low access costs, the array burst trie should keep its container size below 512 strings. However, this implies that more nodes will be allocated as the number of strings stored increase, which will consume more space and can have an impact on overall cache-efficiency; that is, fewer frequently accessed trie paths and containers can reside within cache. The HAT-trie, however, can scale more efficiently by maintaining larger containers that are burst less often. We can see the effect on space in Table 5.7, which shows the number of nodes created relative to the container threshold used by the array burst trie and HAT-trie, when built using the DISTINCT dataset.

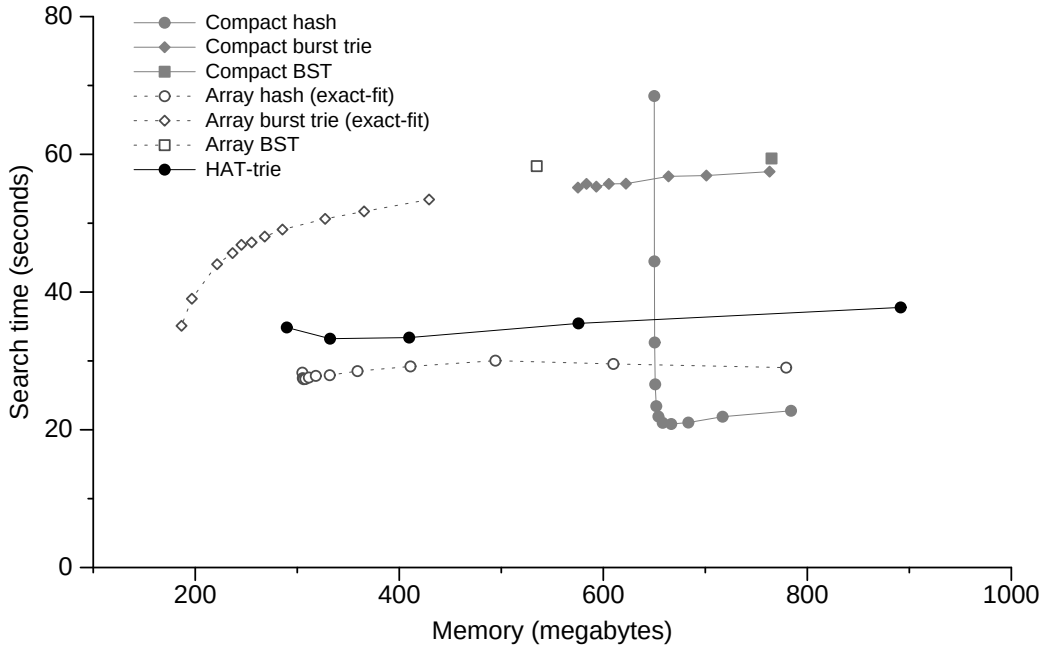


Figure 5.12: The time and space required to search for the strings in the TREC *dataset, on data structures that have been built using the* DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

	Container size	Tries	Containers	Memory (MB)
HAT-trie	2048	6384	264529	891.9
	4096	2677	130994	575.6
	8192	1283	64001	409.7
	16384	680	34780	332.1
	32768	390	19122	289.7
Array burst trie	30	471506	5551879	429.2
	40	359268	4639259	365.5
	50	292000	4058240	327.3
	60	215543	3471822	285.5
	70	184938	3151306	268.1
	80	162390	2896819	255.3
	90	144447	2685839	245.2
	100	128868	2498127	236.5
	128	101194	2128268	221.4
	256	53100	1337190	196.6
	512	27168	791975	186.5

Table 5.7: The number of trie nodes and containers allocated by the array burst trie and HAT-trie when built with the DISTINCT *dataset.*

5.6.5 Performance without hardware prefetch

We repeat the previous skew search experiment on the Pentium III processor, to explore the impact on access time when no hardware prefetching is available and where instructions are more expensive to execute. The results are shown in Figure 5.13.

The compact hash remained the fastest data structure to access but required more than double the space of the array hash, which was almost as fast. The array burst trie displayed strong improvements in performance, requiring the least amount of space while approaching the speed of the hash tables. In this experiment, the array hash was slower to access than the array burst trie when under heavy load, which is primarily due to the absence of hardware prefetch. Without hardware prefetch, the number of cache misses incurred on array access is higher.

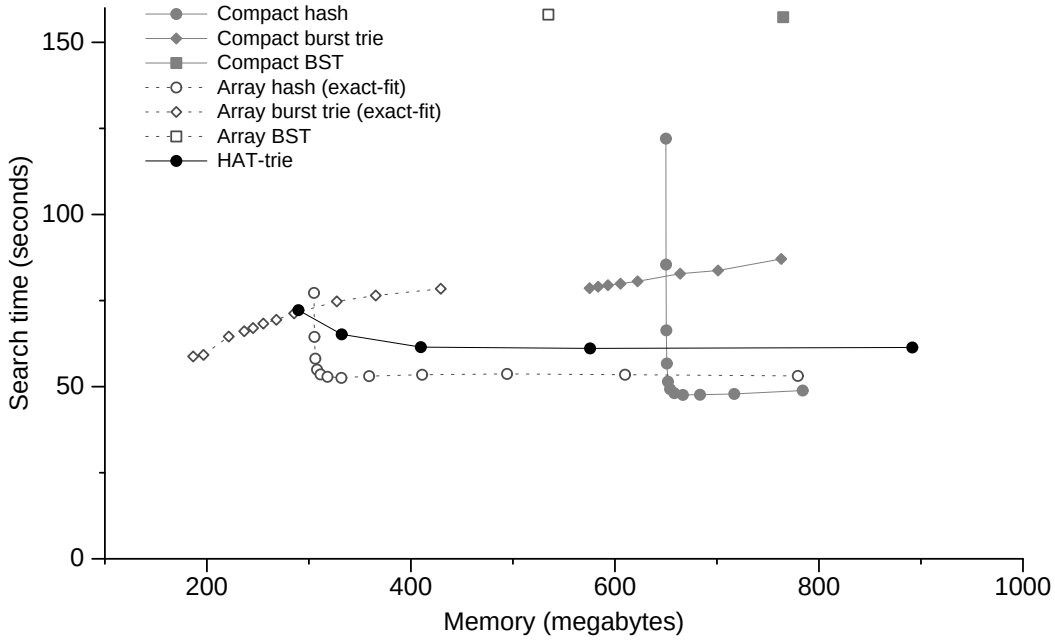


Figure 5.13: The time and space required to search for the strings in the TREC *dataset, on data structures that have been built using the* DISTINCT *dataset, on the Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and HAT-trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*, which included* 128*,* 256*, and* 512 *for the array burst trie. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

Furthermore, we know that large arrays and chains execute high numbers of instructions, which is expensive on a slower processor, such as the Pentium III. The performance of the array hash under heavy load is an important observation, as it explains the poor performance of the HAT-trie relative to the array burst trie, when using a container size beyond 8192 strings. For example, as the container size increased from 16,384 to 32,768 strings, the average load of containers doubled, which caused an impact on access time. These results suggest that the HAT-trie is better suited for use on current machine architectures that employ hardware prefetch.

5.7 Summary

The array burst trie is the fastest and most compact in-memory data structure for maintaining sorted access to strings, which is for example a key requirement in database management software. The array burst trie is space- and cache -efficient with large containers that store more than 128 strings. Nonetheless, our results have shown that large containers can still be expensive to access, due to high instruction costs.

To retain the cache-efficiency of large containers without incurring their high computational costs, we have introduced the HAT-trie, a novel variant of burst trie that changes the structural representation of containers from simple dynamic arrays, to array hash tables. By structuring containers as array hash tables, large containers are effectively broken-up into smaller and thus, faster pieces, at little to no cost in performance. We have experimentally compared the performance of the HAT-trie against the array and compact-chain hash tables, the array and compact-chain binary search trees, and the array and compact-chain burst tries. The HAT-trie was found to be up to 50% faster to build and search than the array burst trie, and in most cases could match the speed of the array hash. Although not as space-efficient as the array burst trie, in most cases the HAT-trie could achieve its best time while using less space than required by the strings alone. The hash tables remained the fastest data structures to access, but only when given enough space. Based on the results presented in this chapter, the HAT-trie is, to the best of our knowledge, the first trie-based data structure that can approach the speed- and space -efficiency of hash tables, while maintaining sorted access to strings.

5.7.1 Recommendations for parameters

We recommend assigning 2^9 slots to the containers of the HAT-trie, which achieves a good balance between time and space. Increasing the number of slots is likely to yield small gains in access time, and at a potentially high cost in space. Reducing the number of slots used, in contrast, will save space but at a cost in time. The HAT-trie was shown to operate efficiently using a container size between 8192 and 16,384 strings. Larger containers are more space-efficient but can become expensive to access. On current processors, the capacity of containers should not exceed 32,768 strings, to avoid incurring high access costs. On slower machine architectures such as a Pentium III, the container size should not exceed 8192 strings.

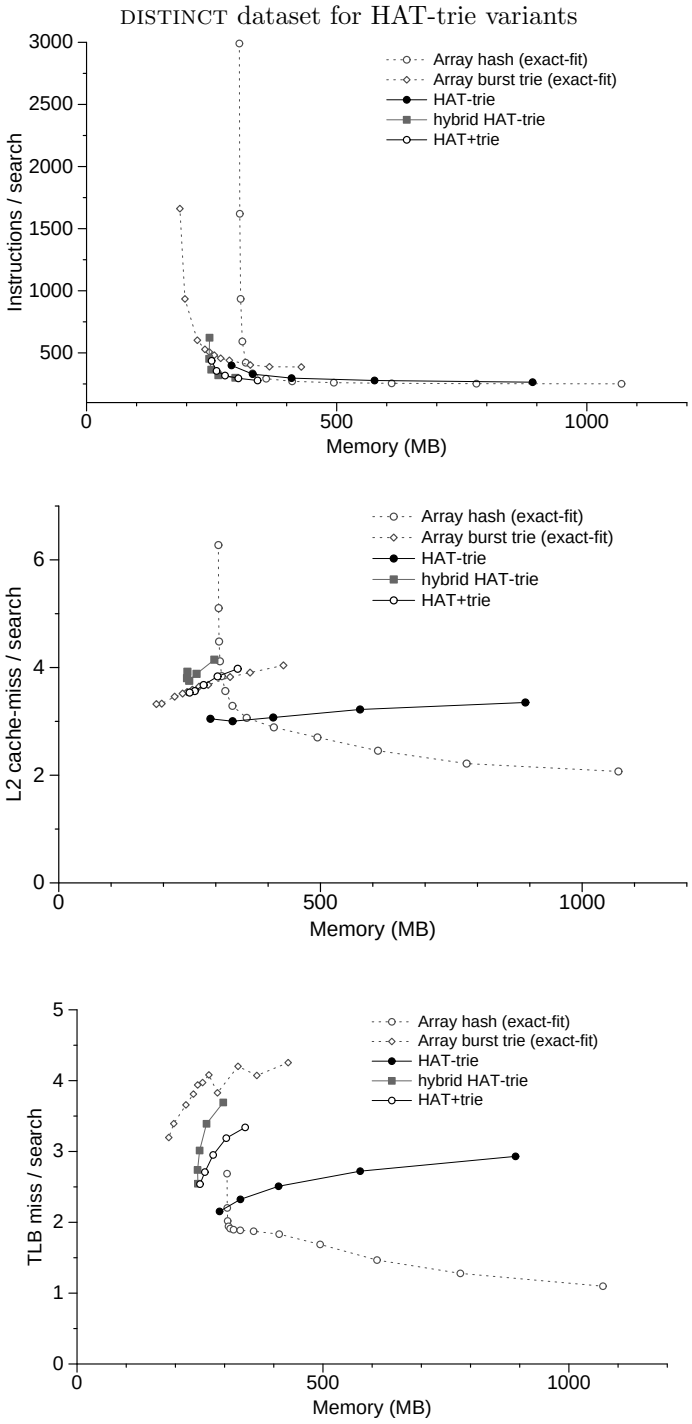


Figure 5.14: The Instructions (top), L2 cache misses (middle), and TLB misses (bottom) incurred by the variants of HAT-trie, for self-searching the DISTINCT *dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables, varied as discussed in Section 5.5. Smaller containers (or more slots) require more memory.*

Chapter 6

Disk-resident data structures for strings

We have explored data structures that can store and retrieve strings efficiently in memory, and proposed cache-conscious alternatives that are faster and more compact than the current best methods. These data structures, however, are designed to reside entirely within main memory, and are therefore oblivious to the high access costs of external memory or disk. A single disk access on a modern machine can stall the processor for tens of millions of clock cycles [Patterson and Hennessy, 2005]. Hence, in-memory data structures are ideally suited for applications that do not exhaust main memory.

However, many applications, such as databases and search engines, are built on infrastructures that require efficient access to data that is often too large to reside entirely within main memory. As a consequence, there is a need for data structures that are able to store and retrieve data efficiently on disk, but, to achieve this, access to disk must be minimized. The choice of data structures is limited, as the majority of trees and tries that are efficient in memory cannot be directly mapped to disk without incurring high costs [Hui and Martel, 1992]. Hence, the efficient storage and retrieval of data on disk is a fundamental problem in computer science.

The best-known structure for this task is the B-tree, proposed by Bayer and McCreight [1972], and its variants. In its most practical form, the B-tree is a multi-way balanced tree comprised of two types of nodes: *internal* and *leaf*. Internal nodes are used as an index or a road-map to leaf nodes, which contain the data. The B-tree has proven highly successful for disk-based data management [Arge, 2002]. A key characteristic is the use of a balanced tree structure, which guarantees a worst-case cost of $O(\log_B N)$, regardless of the distribution of data, where N is the number of keys and B is the branching factor (or node fan-out). This access bound is often significantly better than the performance of an in-memory data structure using virtual

memory [Arge, 2002].

In addition to its widespread use in standard database systems [Chong et al., 2003], the B-tree has many applications, including databases, information retrieval, and genomic databases [Ooi and Tan, 2002]. B-trees have been used to efficiently manage and retrieve large vocabularies that are associated with text databases [Bell et al., 1995]. Some file systems, such as Linux Reiser and Windows NTFS, are also based on B-trees. The B-tree is considered to be the most efficient data structure for maintaining sorted data on disk [Arge, 2002; Garcia-Molina et al., 2001; Sedgewick, 1998]. External hash tables [Chang et al., 1999; Gonnet and Larson, 1988; Kelley and Rusinkiewicz, 1988; Larson, 1988] are also efficient data structures, but cannot guarantee a bounded worst-case cost, nor can they maintain strings in sort order.

The B-trie — a disk-resident trie — has the potential to be a competitive alternative for the sorted storage of data where strings are used as keys. While the concept of the B-trie has been outlined in previous literature [Szpankowski, 2001], it has not previously been formally described, explored, or tested. In particular, there are no discussions on node splitting strategies, which are a critical element of a practical implementation. The B-trie proposed by [Szpankowski, 2001] is simply a static trie indexing a set of containers that store up to b keys.

In this chapter, we propose new algorithms for the insertion, deletion, and equality search of variable-length strings in a disk-based B-trie, for use in common string processing tasks such as vocabulary accumulation and dictionary management. Our variant of B-trie is effectively a novel application of a burst trie to disk, and is therefore composed of two types of nodes: *trie* and *container*. Unlike the B-tree, however, the B-trie is an unbalanced structure, but as we demonstrate later this has little or no impact on actual performance.

A major contribution is the development of an appropriate approach to container splitting, which allows the B-trie to reside efficiently on disk, by minimizing the number of containers created and accessed. Existing disk-resident trie structures, such as the external suffix tree [Tian et al., 2005; Kurtz, 1999], are *full-text* indexes and are therefore not suited for common string processing tasks, due to excessive space requirements and high update costs when moved onto disk [Grossi and Vitter, 2000; Tian et al., 2005].

In the discussions that follow, we address variants of B-trees that are available for disk-based string management, and continue our discussions on disk-resident suffix trees. We then propose new algorithms for the B-trie and experimentally evaluate its performance against variants of B-trees using large sets of strings with a range of characteristics. Our results demonstrate that, in the majority of cases, the B-trie is faster, more scalable, and requires less disk space than the B-trees.

6.1 B-Trees

The B-tree is a balanced multi-way disk-based tree designed to reduce the number of disk access required to manage a large set of strings. The B-tree was proposed by Bayer and McCreight [1972] to solve the problem of external data management. The B-tree employs a similar balancing scheme to that of AVL trees [Foster, 1965], which, however, cannot be efficiently sustained on disk, as changes are not restricted to a single path of the tree (from the root to the candidate node).

The B-tree is one of the most efficient disk-based data structures for external data management [Vitter, 2001; Pagh, 2003; Garcia-Molina et al., 2001; Sedgewick, 1998], as it offers four properties that are desirable for disk-based applications. First, even with large volumes of data, the height of the B-tree remains low, due to the high branching factor which minimizes the number of nodes accessed. Second, with the exception of the root node, all nodes are guaranteed to have a load factor of at least 50%. In practice, an average utilization of 69% for random keys has been observed [Yao, 1978]. Third, the tree is a balanced structure, offering a guaranteed worst-case access cost, regardless of the distribution of data. Bounds on access costs are an essential requirement for applications such as database query engines [Makawita et al., 2000]. Fourth, the tree maintains data in sort order, and can therefore support efficient range search queries.

The B-tree has been successfully applied to many tasks, including spatial and geographic databases, multimedia databases, text retrieval systems, and high-dimensional databases, which are commonly associated with data warehouses [Ooi and Tan, 2002]. There are other data structures available for disk-based string management, yet none offer all the properties described. The M-tree, for example, is a generalization of a standard binary search tree that utilizes three types of nodes: internal nodes, semi-leaves, and leaves. A comparative study of M-trees and B-trees [Arnow and Tenenbaum, 1984] demonstrated that the average search cost of M-trees often rivals that of B-trees. However, M-trees have catastrophic space requirements for large data volumes. Arnow et al. [1985] extended the concept of the M-tree to yield the P-tree. This multi-way tree reduces the space requirements of an M-tree while sustaining its favorable average-case performance. It was found to have superior average-case storage utilization and search costs (for small files) to the B-tree. However, unlike the M-tree, the P-tree is an unbalanced tree structure, to an extent that makes it impractical for large files.

Dense multi-way trees [Culik et al., 1981] are another class of balanced multi-way trees that are similar to the B-tree, but offer alternative tree construction schemes that allow for the creation of highly dense tree structures. Denser trees require fewer nodes, which can significantly reduce the space requirement of the overall tree structure. However, this is achieved at the expense of higher update and maintenance costs.

The buffer tree is a balanced multi-way tree that allocates a buffer to each node [Arge, 1995]. Nodes are only populated with data once their buffer overflows, which amortizes the cost of disk access. Hence, the buffer tree, as the name suggests, batches insertion and search requests to

improve performance. The buffer tree is primarily designed for sorting and for use in external priority queues or external range search.

There are many variants of and enhancements to the B-tree that have been developed to satisfy specific requirements. Comer [1979a] proposed the B*-tree (also known as the B*-method), developed to improve space efficiency by increasing the load factor of each node to a minimum of 67%. In this approach, when a node is full, a sibling node to the left or right is accessed. If the sibling has space, a single key is moved to prevent a split. Otherwise, the two full nodes are split into three. The space saved, however, is at the expense of a high maintenance cost.

A further advance in space efficiency is the method of partial expansion [Lomet, 1987], which uses variable-sized nodes on disk. When a node is full, its size is expanded until a threshold is met. This approach has the advantage of prolonging the split of infrequently accessed nodes, which can reduce tree height. Partial expansion can achieve the same space efficiency as the B*-tree, but at a lower cost. However, it is impractical to maintain dynamic nodes on disk, due to the high costs involved and the space wasted due to external fragmentation [Baeza-Yates and Larson, 1989]. Another similar method is the adaptive overflow technique proposed by Baeza-Yates [1990]. This method also uses variable-sized nodes, but performs unbalanced splits. Its storage utilization, which is adaptive, is not as good as the previous methods described, but it does provide better insertion costs than the B*-tree, while offering space utilization that is almost as good. Unlike partial expansion, however, nodes are grown in fixed-sized chunks, which can be more efficient to maintain on disk. As with the B*-tree and partial expansion, this technique sacrifices speed to improve space.

A simple yet effective improvement is the B^+-tree [Comer, 1979a]. When a leaf node splits, a copy of the middle key is promoted up into the internal nodes; in the B-tree, the middle key is moved out of the split node. This copy only occurs when a leaf splits. The index component of the B^+-tree remains as a B-tree, while the leaf nodes contain a complete copy of the data. This technique separates the index from the data, which has obvious advantages for applications such as databases.

Bayer and Unterauer [1977] took advantage of the independent index of a B^+-tree to develop a simple prefix B^+-tree. In this refinement, internal nodes only store the shortest distinct prefix from the strings promoted from leaf nodes. When a leaf node is split, the middle key is compared to the next larger key to determine the smallest distinguishing prefix. Once found, the prefix is then promoted up the tree and the leaf is split. For example, consider the sequence of strings "auto", "boat", "car", "zebra". On split, the middle key, "boat", is compared against "car", yielding the shortest distinct prefix of "c", which is promoted up instead of "boat". In some cases, however, no space is saved, for example when the split key is "programmer" and the next larger key is "programmers". In this case, Bayer and Unterauer [1977] suggest using a split interval or window around the middle key, to select the smallest key that can be promoted. Before the smallest key is selected, however, the keys in the split interval are filtered to determine their

smallest distinguishing prefix. For example, a split interval consisting of the strings "car", "cat", "boat", and "zebra" is filtered into "car", "ca", "b", and "z". From this example, candidate "b" offers the least characters (scanning left to right).

The goal of the prefix B$^+$-tree is to increase the string capacity of each internal node, to reduce tree height and hence the number of disk accesses. Bayer and Unterauer [1977] proposed a more complicated modification to the prefix B$^+$-tree that can further reduce height by using a prefix compression technique on strings, which is similar to front-coding [Witten et al., 1999]. However, the space saved is at the expense of access time, as internal nodes must be decompressed on access [Rosenberg and Snyder, 1981].

The string B-tree (SB-tree) is another variant proposed by Ferragina and Grossi [1999]. The primary difference between the SB-tree and its predecessors is that strings are not stored within nodes. Instead, they are stored, uncompressed, in a file on disk and nodes simply maintain pointers to them. This approach can substantially increase the fan-out of each internal node, reducing tree height while supporting unbounded length strings.

In a prefix B$^+$-tree, for example, storing long strings in nodes will reduce node fan-out and in cases where the length of a string exceeds the size of a node, overflow nodes are used which can be expensive to maintain. Hence, the SB-tree is likely to be a viable alternative for tasks involving long strings. However, maintaining a sorted array of string pointers per node is unrealistic [Arge, 2002]. During tree traversal, access to a node incurs a disk access and the subsequent binary search of its k sorted string pointers can cause a further $\log_2 k$ disk accesses. The strings on disk are typically not maintained in sort order — due to the potentially high costs involved — which can reduce the access locality of pointers within nodes. Hence, traversing a SB-tree in this manner can cause strings to be accessed randomly on disk, which is inefficient.

Ferragina and Grossi [1999] therefore represent each node as a Patricia trie, also known as a *blind trie*. Binary search is replaced by a trie traversal that incurs at most a single disk access, used to fetch a string suffix for comparison. The expected access cost for traversing a SB-tree is therefore $2 \log_B N$. In addition, the blind tries are stored succinctly on disk, to reduce their space consumption. However, this requires that nodes are decompressed on access and re-compressed on modification, which could become a performance bottleneck. Another potential disadvantage is that nodes are split unevenly, due to the complexity of splitting a blind trie.

To match the analytical cost of a conventional B-tree — where strings are stored within nodes — the SB-tree must keep nodes cached in memory, to eliminate the disk cost incurred on node access. However, this can make the SB-tree inefficient to access in situations where space is highly restrictive. To minimize the random access caused by traversing string pointers, Ferragina and Grossi [1999] suggest sorting the entire dataset on disk, and to build the SB-tree from the bottom-up, that is, to bulk-load [Ferragina and Grossi, 1999; Kärkkäinen and Rao, 2003; Arge, 2002]. This will significantly reduce the construction costs of the SB-tree and improve the access locality of its string pointers. However, sorting the entire dataset beforehand may not be a viable solution when

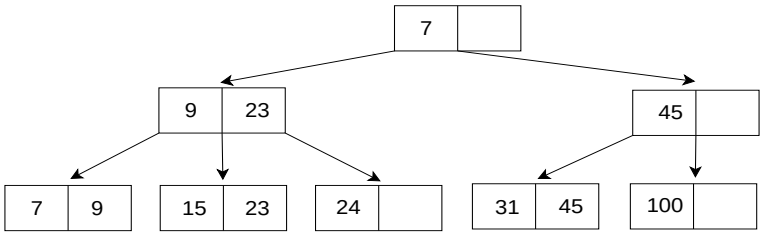


Figure 6.1: A conventional B^+-tree. In this example, integers are used as keys for simplicity.

the dataset is large, or when strings are not known in advance. In such cases, the SB-tree can be constructed from the top-down, which can be expensive. First, new strings are appended to the file on disk, which will reduce the access locality of string pointers. Second, the SB-tree requires nodes (from the same level in the tree) to be stored contiguously and in lexicographic order [Ferragina and Grossi, 1999; Kärkkäinen and Rao, 2003; Na and Park, 2004]. Hence, once a node splits, it will be necessary to move a potentially large number of nodes on disk to maintain this invariant, which can become a performance bottleneck for large datasets. To support top-down construction efficiently, the SB-tree must therefore buffer its internal and leaf nodes in-memory, to minimize access to disk.

Rose [2000] experimentally compared the performance of the SB-tree against the Berkeley B^+-tree [Oracle, 2007], a high-performance open-source B^+-tree implementation. The SB-tree was found to be consistently faster than Berkeley for long strings that contained thousands of characters. Berkeley performed poorly due to the use overflow nodes and its subsequent increase in height. The SB-tree, in contrast, required no overflow nodes and thus remained efficient and compact. With short strings, however, such as those commonly seen in plain-text documents, the SB-tree was shown to be consistently slower. Rose [2000] noted the cause as being the computational overhead of compressing and decompressing nodes and the bottleneck of requiring up to two disk accesses per node: the first to fetch the node, and the second to fetch one of its strings. Although these factors were also present with long strings, the elimination of overflow nodes and the high fan-out compensated. Hence, the SB-tree is an efficient data structure for long strings, but has relatively poor performance otherwise [Rose, 2000], as we show in later experiments.

Ferragina and Grossi [1996] compared the performance of the SB-tree to a suffix array [Manber and Myers, 1990], for the task of finding all occurrences of an arbitrary pattern P in datasets of up to 128 megabytes. The SB-tree was shown to be more efficient than a suffix array, and has subsequently been successful in applications that involve pattern matching [Ferragina and Luccio, 1998; Farach et al., 1998; Crauser and Ferragina, 1999].

B-trees have also been modified to make better use of memory and cache. A cache-conscious B^+-tree stores the child nodes of any given node sequentially [Rao and Ross, 2000]. This forms a clustered index where only the address of a node's first child is required, in order to access the

remaining child nodes. Access locality is improved as a result, but update costs are considerably higher due to the overhead of maintain clustered indexes. Furthermore, the cache-conscious B^+-tree is an in-memory data structure that operates solely on fixed-length keys. As a consequence, it is not a viable choice for managing variable-length strings on disk.

The persistently cached B-tree [Kato, 2003] is another innovation where performance is improved by exploiting unused areas within nodes. This is accomplished through a replication technique known as persistent caching, where part of one node is copied into the free space of another, thereby effectively loading two nodes from one disk access. This approach can reduce search costs using fixed-length keys, but update costs can be high due to the non-trivial task of maintaining data coherency amongst nodes.

Cache-oblivious data structures are designed to perform well on all levels of the memory hierarchy (including disk) without prior knowledge of the size and characteristics of each level [Frigo et al., 1999; Kumar, 2003]. Brodal and Fagerberg [2006], for example, theoretically investigated a static cache-oblivious string dictionary. Similarly, a dynamic cache-oblivious B-tree [Bender et al., 2000] has been described, but with no analysis of actual performance. The cache-oblivious dynamic dictionary [Bender et al., 2004] has been compared to a conventional B-tree, but on a simulated memory hierarchy. These assume a uniform distribution in data and operations, which is typically not observed in practice [Bender et al., 2002].

Bender et al. [2006] theoretically investigated a dynamic cache-oblivious string B-tree, which has been claimed to handle unbounded-length strings efficiently. However, the authors present no experimental evidence and derive expected performance from experiments involving a cache-oblivious B-tree [Bender et al., 2000], using uniformly distributed integers. Indeed, a dynamic cache-oblivious string B-tree has yet to be implemented [Bender et al., 2006].

Despite its success, the B-tree has disadvantages. One problem is the complexity involved with processing nodes. Splitting a node generally involves numerous steps that typically incur expensive performance penalties such as un-localized disk access. Another problem is that strings within leaf nodes may not share common prefixes, even though they are lexicographically adjacent. For applications that require prefix searches, a B-tree can be inefficient. The B-tree is potentially inefficient under skewed access, as frequently accessed leaves cannot be brought closer to the root of the tree, making the B-tree less attractive for applications such as search engines that typically process many repeated searches.

Researchers have addressed the problem of skew access on disk by proposing several theoretical data structures that are self-adjusting. Sherk [1989] proposed a generalization of splaying to K-ary trees, forming a self-adjusting B-tree called a k-splay tree. Unlike the B-tree, the k-splay tree can become severely unbalanced and, as a consequence, can be expensive to access and maintain on disk. Martel [1991] introduced another a self-adjusting data structure called the *k-forest*. The k-forest is simply an ordered set of B^+-trees, where the first tree is of height of 1, the second tree is of height 2, and so forth, up to a height of h. A search proceeds by accessing the trees, in sequence,

until a match is found. Once found, the key is moved to the first tree, and if there is no space, a key from the first tree is selected and demoted into the second tree. The demotion process can propagate through the trees, until finally a new tree of height $h+1$ is created. Frequently accessed items will therefore be located in trees of smaller height, which will improve access costs. However, the cost of moving and demoting keys per search can become a performance bottleneck on disk, and there is no benefit under uniform access distributions. Furthermore, the cost of unsuccessful search is high, as it involves accessing all trees in the k-forest [Hui and Martel, 1992].

Ciriani et al. [2002; 2007] proposed (in theory) a self-adjusting disk-resident skip list. The basic concept of a skip list involves building an index upon a totally ordered set of n atomic items, such as integers. Hence, the disk-resident skip list is built from the bottom-up using keys that are known and sorted in advance. Although the skip list can be updated from the top-down, it is often inefficient to do so — particularly with strings — due to cost of updating its multi-layer index [Pugh, 1990; Williams et al., 2001].

The disk-resident skip list supports unbounded-length strings in a manner similar to the SB-tree. That is, strings are kept on disk and are accessed via string pointers. Hence, up to two disk reads can be incurred on string access: one to fetch the node that contains the string pointer, and another to fetch the string for comparison. Although its expected performance is studied in theory, the external skip list has yet to be experimentally compared against other external string data structures, such as the B$^+$-tree.

6.1.1 B$^+$-tree implementation

We sought to develop a high performance disk-based B$^+$-tree to act as a baseline for comparison with our B-trie. We implemented a standard B$^+$-tree — where internal nodes store full-length strings — and a prefix B$^+$-tree. Other B-tree variants, such the cache-conscious B$^+$-tree and the persistently cached B-tree, are not suitable, due to their high update costs and lack of support for variable-length strings. Similarly, the SB-tree is also not a suitable candidate. We maintain variable but bounded-length strings that are typically no more than a few tens of characters in length. The SB-tree is known to be inefficient with such strings [Rose, 2000]. Furthermore, to the best of our knowledge, there is currently no implementation support for a dynamic SB-tree [Ferragina and Grossi, 1996; Rose, 2000; Na and Park, 2004; Kärkkäinen and Rao, 2003; Ferragina and Grossi, 1999; Fan et al., 2001] that can operate efficiently when the number of strings to be stored or indexed is not known in advance. A static SB-tree is available (discussed in Section 6.4.8), but requires strings to be sorted in advance and, once built, it can not accommodate new strings. Therefore, it is reasonable to assume that a good implementation of a standard or prefix B$^+$-tree, on balance, is competitive with even recent innovations.

We have followed a conventional B$^+$-tree model [Comer, 1979a]. All nodes are of fixed size, which in our case is 8,192 bytes. This is a typical disk-block size and has been shown to provide good performance [Deschler and Rundensteiner, 2001; Gray and Graefe, 1997]. In contrast to the

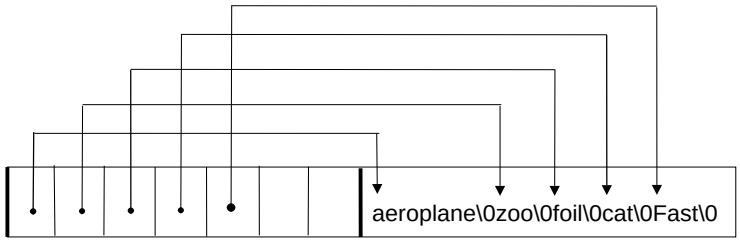


Figure 6.2: Null-terminated strings are appended to existing strings in a node. Pointers to these strings are kept in sort order. This approach permits rapid insertion and search.

SB-tree, strings are stored within nodes. To prevent the situation where a single string consumes an entire node, we enforce a string limit of 1000 characters. (A production implementation would have to cater for longer strings by using overflow nodes, but such strings do not arise in our data.) Duplicates are maintained through the use of a 4-byte counter (an accumulator), which is stored before each string. Only leaf nodes maintain accumulators.

Hansen [1981] describes several schemes for node organization, including Middle Gap, Binary Search, Partitioned, and Square Root structures, which are designed to fully utilize the space of a node at the cost of some search performance. For instance, with the Middle Gap scheme, strings are sorted and partitioned into several groups separated by unused space. When a new string is inserted into a group, the existing strings must be moved to maintain sort order. We chose to sacrifice some space efficiency within our nodes to obtain a node-organization scheme that offers fast insertion and search. Our node organization is somewhat similar to the unordered node structure mentioned by Hansen [1981]. Strings are stored in a sequential (occurrence order) manner; new strings and their accumulators are simply appended. This offers rapid insertion, but at the expense of a linear search on node access, which is inefficient. To improve this model, we assigned a string pointer to each string. These pointers are maintained in a single array that is stored before the list of strings, and are kept in ascending lexicographic order. This allows for binary search, which is known to be an efficient search method for accessing nodes [Ben-Asher et al., 1999]. Figure 6.2 shows an example.

To reduce wasted space within nodes, we apply two techniques. First is a string-pointer allocation strategy where each node begins with 128 empty pointers. Once all pointers are exhausted, room is made by appending space to the existing pointer-array. The strings that follow the array must be shifted to the right to accommodate the new string. Second, all buffered nodes are given an additional kilobyte of space when brought into memory. This oversize region helps ensure that 100% of the node is utilized prior to splitting.

We use a bottom-up splitting approach, where a split first occurs in a leaf node and then propagates up [Sedgewick, 1998]. A top-down technique has been proposed by Guibas and Sedgewick [1978], which performs the splitting during tree traversal. This approach results in a slight de-

crease in space efficiency, as full nodes can be unnecessarily split. The main components of the B^+-tree are as follows:

Internal nodes: These are 8192 byte disk blocks that serve as a road-map or an index to other nodes. They contain an array of string pointers, an array of node pointers, a string counter and a free-space counter. Strings in these nodes are only copies from those promoted from leaf nodes.

Leaf nodes: Structurally identical to internal nodes, except for the array of node pointers, which is absent.

A stack: Used to record the path taken to reach a leaf node. This avoids the use of parent pointers within nodes, which are expensive to maintain [Arge, 2002].

In a production system, we must store data that is associated with each string. We maintain string accumulators that can easily be changed to represent pointers to data objects for example. Larger data fields can be associated with each string in the leaf nodes, however, this will leave less room for the strings themselves, forcing the creation of more nodes. We do not investigate the impact on performance when large data fields are associated with strings. However, in such cases, we found that it is generally more efficient to associate a single pointer with every string, which is only traversed when the additional data is required.

The B^+-tree algorithm we implemented for string insertion, deletion, and search complies with the standard descriptions of the B^+-tree [Bayer and McCreight, 1972; Comer, 1979a; Jannink, 1995]. However, certain additional rules were adhered to:

1. A node is split once its free space is exhausted. All strings smaller or equal to the middle string are retained in the original node.
2. On modification, a node is immediately synchronized (written) to disk, to ensure data integrity.
3. The most significant bit (MSB) of each node-pointer determines the type of node it refers to. If its MSB is set, then it points to a leaf node. A pointer to a node is represented as an unsigned 32-bit integer which stores the block number of a node (in a file) on disk.

We implement what is known as *lazy deletion* [Jannink, 1995]. Deletion proceeds by first searching for the required string, and assuming it is found, it is removed from the acquired leaf node. The leaf node is then internally re-organized, to update its string pointers and to eliminate internal fragmentation. The computational cost of node re-organization is small and bound by the size of the node. Once the leaf node becomes empty, it is flagged as having been deleted by placing its file address into an address pool, to be reused by new nodes. The parent node is then modified to

have its corresponding leaf-node pointer (and its string) deleted. Once the internal node becomes empty, it is deleted in the same manner.

Lazy deletion is a simple and time-efficient way to delete entries in large B^+-trees. Many database system implementations have used lazy deletion [Gray and Reuter, 1992; Garcia-Molina et al., 2001]. Johnson and Shasha [1989; 1993] showed that with a mix of insertions and lazy deletions — assuming that deletions do not outnumber insertions — nodes can retain acceptable percentages of entries.

Lazy deletion will waste space when deletions outnumber insertions. In such cases, Jannink [1995] described an alternative deletion algorithm that can shrink a B^+-tree gracefully, to conserve space. When a key is deleted from a node, and the node is deemed as being under-loaded, we access its immediate neighbors to check whether they can transfer some of their keys without becoming under-loaded themselves. Otherwise, the under-loaded node must be merged with one of its neighbors, and have its parent node updated. Such an algorithm, however, can become expensive to apply on large B^+-trees, due to the potentially high number of disk accesses involved.

6.2 Trie-based data structures

Tries have two properties that cannot be easily imposed on data structures based on binary search. First, strings are clustered by shared prefix and second, there is an absence of — or great reduction in the number of — string comparisons. In addition, trie-based data structures, such as our B-trie, are implicit cost-adaptive data structures. Trie nodes can be rapidly traversed, allowing frequently accessed containers to be acquired at minimal cost, even though they are not physically moved closer to the root. This is a key distinction to self-adjusting tree structures, such as the splay tree [Sleator and Tarjan, 1985], the k-splay tree [Sherk, 1989], the k-forest [Martel, 1991] and the external skip list [Ciriani et al., 2007]. These data structures achieve cost-adaption through structural modifications that are often expensive to apply.

A highly effective solution to the space problem is the burst trie [Heinz et al., 2002]. The burst trie stores strings within bounded-sized containers. Once a container is full, it is *burst* which involves creating a new parent trie node that is represented as an array of pointers, one pointer for each letter of the alphabet A. The strings within the original container are then distributed into at most A new containers, in accordance to the lead character, which is then removed. By storing strings within containers and creating only a single trie node per burst, the burst trie is able to reduce the space required by the array trie by as much as 80%, with little to no impact in access speed.

The burst trie is currently one of the fastest in-memory data structures for strings, but it cannot be directly mapped to disk because of the way it represents and manages containers. Bursting a container on disk implies creating up to A new fixed-sized containers that can be randomly accessed, which is expensive. Similarly, other variants of trie, such as the TST, are also unsuitable

for disk due to their high space requirements and poor access locality. To our knowledge, there has yet to be a proposal in literature for a trie-based data structure, such as the burst trie, the can reside efficiently on disk to support common string processing tasks. Such a data structure would inherit the clever advantages offered by tries, such as the removal of common prefixes and implicit cost-adaption, which is of value considering that B^+-trees are known for their poor performance under skew access.

Suffix trees are well-known trie-based data structures that can reside on disk [Clark and Munro, 1996]. However, these data structures maintain *full-text* indexes that store every distinct suffix in a text collection [Grossi and Vitter, 2000; Kärkkäinen and Rao, 2003]. Conventional data structures — such as the B^+-tree and our B-trie — are *word-level* indexes that store only distinct words in a text collection. Full-text indexes are more powerful than word-level indexes, as they can efficiently support complex search operations such as finding all occurrences of a pattern in a text collection, whereas word-level indexes are typically restricted to exact-match word searches. Full-text indexes are used in pattern matching applications, such as molecular biology, data compression, and data mining.

Full-text indexes are space-intensive and can typically require 4 to 20 times the space of the text they index [Kurtz, 1999]. Their high space consumption is a major restriction on their application to common string processing tasks, such as vocabulary accumulation and text indexing [Grossi and Vitter, 2000; Ferragina and Manzini, 2005]. Grossi and Vitter [2000] introduced a compressed suffix tree (and suffix array) that could, in theory, approach the space-efficiency of inverted lists. In practice, however, inverted lists are substantially more space- and time-efficient [Zobel and Moffat, 2006; Zobel et al., 1998], but cannot support pattern matching.

As a result of their high space consumption, suffix trees (and suffix arrays) can rapidly exhaust main memory for large text collections. Researchers have addressed this problem by proposing new suffix-tree construction algorithms that can substantially reduce both the time and space required to construct and maintain a suffix tree on disk [Tian et al., 2005; Cheung et al., 2005; Ko and Aluru, 2006]. Relative to the performance of word-level indexes, however, current suffix-tree construction algorithms remain substantially more expensive, and thus are not suitable replacements for word-level indexes, such as the B^+-tree, for applications that do not require pattern matching.

Chowdhury et al. [2007] proposed, in theory, the application of a word-level trie for external storage, called a DiskTrie. The DiskTrie is a static variant of Patricia trie (an LPC-trie) that is designed for use in small external flash memory devices. The Patricia trie and its variants are commonly used to represent external tries because their size is not dependant on the length of the keys, but rather on the number of keys inserted, which makes them well suited for situations where space usage is highly restrictive. However, as claimed by Heinz et al. [2002], the Patricia trie (and its variants) are expensive structures to access when maintaining a large set of strings, and are not practical solutions for common string processing tasks, where typically both access time and space are important. It is therefore attractive to explore viable methods of applying the

space- and time -efficient burst trie on disk.

Hence, in the discussions that follow, we propose a novel variant of B-trie which is designed to efficiently maintain a word-level index on disk, for common string-processing tasks such as dictionary management, text indexing, document processing, and vocabulary accumulation.

6.3 The B-trie

The B-trie is an unbalanced multi-way disk-based trie structure, designed to sort and cluster strings that share common prefixes. It borrows the design of the burst trie [Heinz et al., 2002] to maintain a space-efficient trie, by storing strings within containers that are structurally similar to the nodes described for the B^+-tree: fixed-sized disk blocks represented as arrays. Once a container becomes full, a splitting strategy is required that, in contrast to bursting, throttles the number of containers and trie nodes created. The concept of a *B-trie* has been suggested by Szpankowski [2001] and is briefly discussed elsewhere [Knessl and Szpankowski, 2002; 2000; Mahmoud, 1992]. However, information about the data structure, such as algorithms to insert, delete, search, and how to split nodes efficiently on disk, is scarce.

We propose that containers undergo a new splitting procedure called a *B-trie split*. In this approach, the set of strings in each container is divided on the basis of the first character that follows the trie path that leads to the container. (Each node in the path consumes one character.) When all the strings in a container have the same first character, this character can be removed from each string; subsequent splitting of this container will force the creation of a new parent trie. We label these containers as *pure*. When the set of strings in a container have distinct first characters, several paths in the parent trie lead to it. We label these containers as *hybrid*; subsequent splitting of this container will not create a new parent trie.

When a container is split, a character is first selected as a split-point and the strings are then distributed according to their lead character. That is, strings with a lead character smaller than or equal to the split-point remain in the original container, while others are moved into the new container. During a split, the trie property is temporarily violated, but only for the leading character in each string. Once the split propagates into the parent trie node, the trie property is restored, but, in some cases, with multiple pointers to the same container, forming a *hybrid* container. Figure 6.3, Figure 6.4, and Figure 6.5 illustrate examples of this splitting procedure, which we explain in more detail later.

In either case (hybrid or pure), each container is a cluster of strings with a shared prefix, a property with clear advantages for tasks such as range search. In addition, the B-trie offers other advantages. One is that the cost of traversing a chain of trie nodes is, in comparison to the traversal of internal B-tree nodes, significantly lower; identification of a container involves no more than following a few pointers. Another is that short strings — which are the commonest strings in applications such as vocabulary management — are likely to be found without accessing

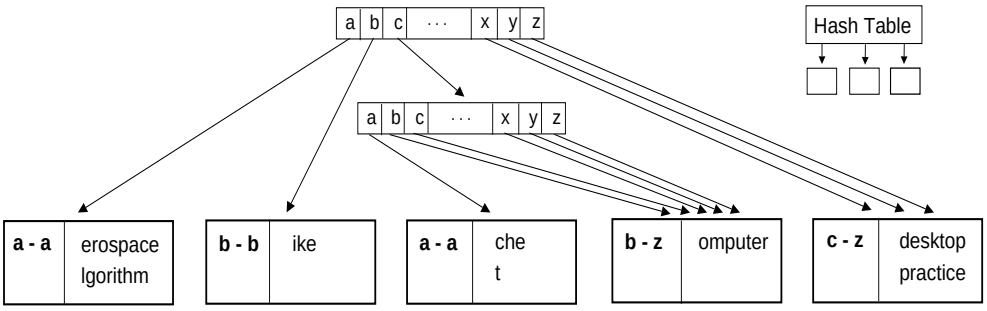


Figure 6.3: The words "cat","algorithm","computer", "practice", "cache", "bike", "desktop" and "aerospace" were inserted into the B-trie, creating three pure containers (first three from the left) along with two hybrids. The hash table stores strings that are consumed. "c" for example, would be consumed by the root trie and "a", would be consumed by the first pure container.

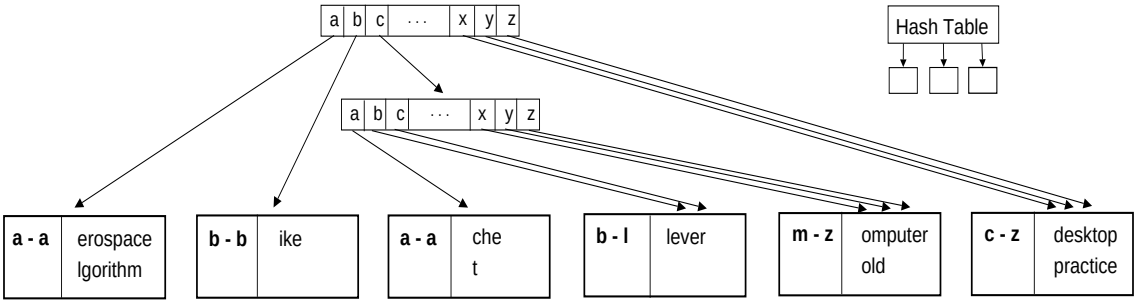


Figure 6.4: The strings "cold" and "clever" were inserted into the B-trie in Figure 6.3. The second hybrid container (from the right of Figure 6.3) split, creating two new hybrids.

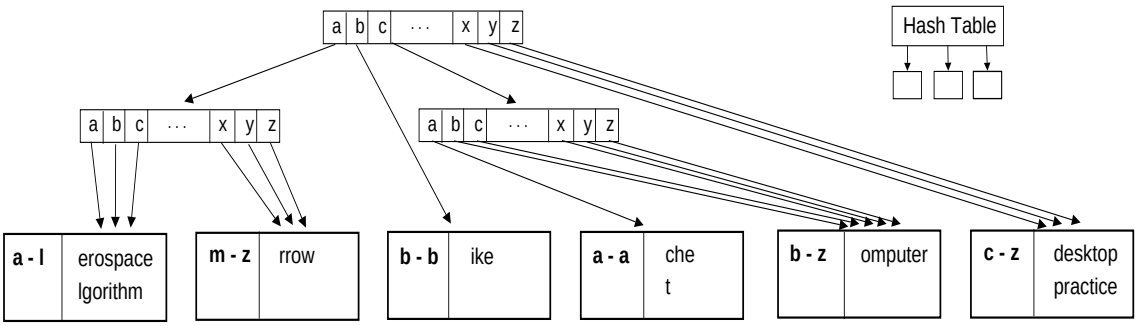


Figure 6.5: The word 'arrow" was inserted into Figure 6.3. The left-most pure container split into two hybrids and a new parent trie.

a container and can be conveniently managed in memory. This splitting process is, however, a major contribution, as it solves the problem of efficiently maintaining a trie structure on disk for common string processing tasks.

A potential drawback compared to a B$^+$-tree is that splitting a container cannot guarantee that the two new containers are equally loaded. In most cases, the load is likely to be approximately equal (as we observed in our experiments described later). In some cases, however, it is highly skew. For example, if every string but one begins with the same character, then one of the new containers will contain one string only, while the other contains the rest. However, our B-trie splitting algorithm ensures that there are no empty containers. Furthermore, the B-trie maintains the expected logarithmic access costs of the burst trie [Heinz et al., 2002; Knessl and Szpankowski, 2000; Szpankowski, 1991; Clement et al., 2001]. Hence, as we demonstrate later, an occasional uneven split has little to no impact on performance, even for our URLS dataset, where the strings exhibit a skew that is likely to provoke a surplus of uneven splits.

Another drawback is the applicability of bulk-loading. To bulk-load a data structure implies populating leaf nodes without consulting an index. This is accomplished by using sorted data; the index is constructed independently as the leaf nodes are sequentially populated. Bulk-loading is an efficient way of constructing B$^+$-trees. However, the B-trie cannot be efficiently bulk-loaded because its index — which can consume strings — is not independent from the data stored in containers. We now describe algorithms for maintaining a B-trie. The main components of our B-trie are as follows:

Containers: Structurally — apart from the added character-range field — containers are identical to the leaf nodes used by our implementation of a B$^+$-tree. However, the lead character of each string in the container must be within its character range.

Trie nodes: A trie node is an array of pointers, one pointer per character in the ASCII table, 128 pointers in total. The leading character of a string is used as an offset and is discarded once a new trie node or a pure container is acquired. Recall that a pure container contains strings that begin with the same lead character (which is removed). A pointer in a trie can be empty (null) or point to either a container or a trie. As discussed by Heinz et al. [2002], the number of trie nodes — and hence the space they require — can be kept small due to the use of containers. As a result, the space saved by employing more space-efficient trie structures, such as the Patricia trie or the TST, was found to be small and did not justify tolerating higher access costs. Hence, for the burst trie, the use of an array trie was preferable. This is also the case for our B-trie, making the use of an array trie — which is fast and can be directly mapped to disk — preferable over more space-efficient but slower alternatives.

An auxiliary data structure: Access to a trie node or to a pure container will delete the lead character from a string during search. It is therefore possible for a string to be consumed entirely (deleted) prior to reaching or searching a container. When this occurs, an auxiliary data structure is used to store such short strings. We use our cache-conscious hash table for this purpose. When a string is inserted into the hash table, it is immediately copied into a heap file on disk to allow for re-construction. Alternatively, consumed strings can be handled by setting a string-exhaust flag within the respective trie node or pure container, as described for the burst trie [Heinz et al., 2002]. This approach will eliminate the small performance overhead of accessing a hash table whenever a string is consumed during search, but can require the B-trie to maintain empty pure containers on disk, in order to maintain their string-exhaust flags, which is inefficient.

The principal structures can be formally defined as follows. A node N is a set of pointers p, one for each character c in the alphabet A; that is, $N = \{p_c | c \in A\}$. A pointer is a directed arc from a node N to another node N' or to a container B; a B-trie is then a directed acyclic graph with a single root in which all routes (traversals of the graph) terminate at a container. Each pointer is labeled with a character; some pointers are null, but all nodes have at least one non-null pointer. A complete or terminated route R is represented as a chain

$$N_1 \rightarrow_{c_1} N_2 \rightarrow_{c_2} \cdots \rightarrow_{c_{m-1}} N_m \rightarrow_{c_m} B$$

in which each arc $\rightarrow_c$ corresponds to a labeled pointer p_c. The sequence $s(R)$ of arcs in R is a representation of the string $c_1 \ldots c_m$. There are two types of containers: *hybrid* and *pure*. Pure containers are those that have a range of a single character. Hybrid containers are those that have a range comprised of two or more distinct characters. All strings in a container share some prefix h, and thus the prefix need not be stored. That is, a pure container is a set of strings

$$B^P(h) = \{t | s = h \cdot t \in V \text{ for any string } t\}$$

where V is the complete set (or vocabulary) of strings stored in the B-trie, h is a string, and "$\cdot$" is the string concatenation operator. A hybrid container is a set of strings

$$B^H(h, l, u) = \{c \cdot t | s = h \cdot c \cdot t \in V \text{ and } c \in [l, u]\}$$

where l and u are characters. The algorithms described later in this section enforce the following properties:

1. There is only a single route to each pure container.
2. There is only a single route from the root to any trie node.
3. For a route R leading to a pure container $B^P(h)$, the sequence $s(R) = h$.
4. For a route R leading to a hybrid container $B^H(h, l, u)$, the sequence $s(R) = h \cdot c$ where $c \in [l, u]$.
5. In a hybrid container, $l \neq u$.
6. For a hybrid container $B^H(h, l, u)$ where $h = c_1 \dots c_{m-1}$, there is a set $\mathcal{R}$ of routes

$$\mathcal{R} = \{N_1 \rightarrow_{c_1} \cdots \rightarrow_{c_{m-1}} N_m \rightarrow_c B^H(h, l, u) | c \in [l, u]\}$$

and no other routes terminate at $B^H(h, l, u)$.

Before proceeding with the algorithms, we give a brief overview of how to maintain a B-trie. The B-trie is initialized with one empty hybrid container with a parent trie. When a hybrid container splits, it creates one new sibling container (the original container is re-used). This action grows the B-trie horizontally. An example is shown in Figure 6.4. Eventually, splitting a hybrid will lead to the creation of a pure container. As strings are distributed into the pure container, their leading character is removed. This may cause a single string to be consumed entirely, in which case the string is reconstructed (from the path taken to reach the container) and stored in the hash table. When a pure container is split, a new trie node is created and assigned as its parent, after which the pure container is transformed into a hybrid and the split proceeds as a hybrid. When a pure container splits, the B-trie is grown both vertically and horizontally. An example is shown in Figure 6.5. We adhered to the following design principles to allow for a fairer comparison to the B$^+$-tree:

1. Containers are structured and managed in much the same manner as the leaf nodes used by our implementation of a B$^+$-tree, as discussed in Section 6.1.1. That is, containers are initialized with 128 string pointers and are given an additional kilo-byte of free space when read into memory. No string duplicates are maintained. Instead, strings that are stored in containers or the hash table are proceeded by a 4-byte accumulator.

2. A pointer refers to an internal node if its most significant bit is set. A pointer to a node is represented as an unsigned 32-bit integer which stores the block number of a node (in a file) on disk.

3. On modification, a trie or container is immediately synchronized (written) to disk, to ensure data integrity.

6.3.1 B-trie initialization

When there are no trie nodes or containers on disk, a new empty hybrid container and a parent trie is created. The hash table is re-populated with strings found in its heap file.

6.3.2 To search for a string

Equality search takes a query string Q as input and may return a pointer to a container B, its parent trie node P (if any) and Q', which represents the characters of Q that were not consumed during traversal. That is, searching for a string Q involves traversing the B-trie to determine whether the string Q was consumed and thus stored in the hash table, or to find a pure container $B^P(h)$ such that $t \in B^P(h)$ and $Q = h \cdot t$, or to find a hybrid container $B^H(h \cdot c, l, u)$ such that $c \cdot t \in B^H(h \cdot c, l, u)$ and $Q = h \cdot c \cdot t$.

A search proceeds as follows. The leading character of Q is used as an offset into the trie nodes, beginning from the root. Prior to accessing a child node that is either a trie node or a pure container, the lead character is deleted. If instead, an empty pointer is encountered or, if Q' is empty (that is, the query string is completely consumed during traversal), the search concludes by consulting the hash table for Q. When a container B is acquired, a binary search for Q' concludes the search.

6.3.3 To insert a string

Insertion takes a string Q, performs an equality search as described above, and on search failure, inserts what remains of the query string (that is, Q') into the acquired container B. That is, insertion of a string Q is the task of finding a pure container $B^P(h)$ such that $Q = h \cdot t$, and adding t to $B^P(h)$, or finding a hybrid container $B^H(h \cdot c, l, u)$ such that $Q = h \cdot c \cdot t$, and adding $c \cdot t$ to $B^H(h \cdot c, l, u)$. If the container is now full, it must be split. Otherwise, the insertion process concludes.

In the event where the query string was consumed during search (that is, Q' is empty), Q is stored in the hash table and the insertion is complete. If a null pointer was encountered during search, a new container is created to store Q'. The new container has a character range that engulfs all neighboring null pointers in the parent trie P that span from the original null pointer encountered during search (up until a non-null pointer is accessed). This action will determine

whether the new container is hybrid or pure. In the latter case, care must be taken to discard the container prior to writing it out to disk and to clear (null) its parent pointer, if it consumes Q' entirely. In this case, Q is stored in the hash table to complete the insertion process.

6.3.4 To delete a string

The B-trie employs a lazy deletion scheme, similar to that described for the B$^+$-tree. Deletion proceeds as follows. We search for the required string Q, as described above. If Q is consumed during traversal, we clear the end-of-string flag in the acquired trie node or pure container, and delete Q from the hash table to complete the deletion process. If, instead, a null pointer is encountered during search, then we delete Q from the hash table (if it exists), to complete the deletion process.

Otherwise, either a hybrid or pure container is acquired which is binary searched for the string suffix Q'. If the suffix is found, it is removed and the container is internally re-organized to avoid space wastage due to internal fragmentation. The computational cost of re-organizing the container is small and bound by the size of the container. Once the container is empty, all of its incoming pointers from the parent trie node P are nulled, and its file address is placed in an address pool for reuse. Once P has had all of its pointers nulled, then P is also deleted by having its parent pointer nulled, and by placing its file address in an address pool for reuse. Lazy deletion of trie nodes can propagate up to the root.

Alternatively, we can apply a more space-efficient deletion scheme, as described for B$^+$-trees [Jannink, 1995]. That is, once a container becomes empty, we check its immediate neighbors to determine whether we can transfer some strings. We can only initiate a transfer if the neighbor is a hybrid container and will not become empty on split. (A pure container can be used, but, on split, the lead character of its strings must be restored which can complicate matters.) If these two conditions are satisfied, the neighboring hybrid node is split and its strings are distributed between itself and the empty container. The parent trie node is then updated accordingly.

Otherwise, if no immediate neighbors are hybrids or if a split will cause a neighbor to become empty, then the empty container cannot be merged and must be deleted. One way to delete the container is to apply lazy deletion as described, and simply flag the container as having been deleted. However, to conserve space, the container should be deleted from disk, but which can involve shifting a potentially large number of containers (and updating their respective trie nodes), which is likely to be expensive for a large B-trie. As with the B$^+$-tree, this option should only be applied when deletions outnumber insertions, or when space is highly restrictive.

6.3.5 Splitting a container

Splitting takes place when the insertion algorithm deems a container as full, due to insufficient space. Splitting a hybrid container $B^H(h, l, u)$ produces two containers, both of which can be either

pure or hybrid. This action grows the B-trie horizontally. An example is shown in Figure 6.4. The basis of the split is the first character of the strings in the container. A pair of characters d and d' need to be chosen such that d' is the character that lexicographically follows d, with $d, d' \in [l, u]$, and — as nearly as possible — among the strings in $B^H(h, l, u)$, roughly half begin with a character in the range $[l, d]$ while the remainder begin with a character in the range $[d', u]$. If the range cannot be neatly divided, one container or the other will be under-loaded but no empty containers are maintained.

That is, a split-point must be found that can achieve good distribution across two new containers. To determine a split point, we count the number of occurrences of each leading character in the original container. Then, in lexicographic order, we simulate moving these counters (representing the clusters of strings to be moved) to a new location, computing a simple distribution ratio (strings moved divided by the strings remaining). Once the ratio exceeds 0.75 — a threshold found though preliminary trials — a suitable split point has been found and the strings can be distributed accordingly. Achieving this threshold may not always be possible. In such cases, the second last counter to be moved (its representing character) is used as d, which can cause the creation of an empty container, which is not maintained. Hence, the pointers in the parent trie P between the range of $[l, d]$ are nulled.

After splitting a hybrid container, if $l \neq d$ then the left-hand new container will be a hybrid container $B^H(h, l, d)$; otherwise it will be a pure container $B^P(h \cdot l)$. Being a pure container, the leading character of each string that is stored in the container is removed. This can result in the consumption of a string, which must therefore be reconstructed (from the path taken to reach the container) and stored in the hash table. Similarly, if $d' \neq u$ then the right-hand new container will be a hybrid container $B^H(h, d', u)$; otherwise it will be a pure container $B^P(h \cdot u)$. The parent trie node P of the two new child containers must now have its pointers of range $[l, u]$ re-assigned accordingly.

The splitting procedure for a pure container $B^P(h \cdot u)$ is almost identical to that of a hybrid. The difference is that a new parent trie node is created which is assigned to the pure container. The old parent becomes the grandparent. All pointers in the new parent are assigned to the pure container, which changes the container into a hybrid. The split then proceeds as described for a hybrid container, and so, when a pure container splits, the B-trie is grown both vertically and horizontally. An example is shown in Figure 6.5. The splitting process terminates only when both children are not full, in which case, the new containers and their parent trie are written to disk. Otherwise, the process continues recursively by splitting the full child container. The non-full child is written to disk and discarded from memory. These novel elements of pure and hybrid containers are how the trie properties are maintained, while giving a B-tree-like organization of data on disk.

Dataset	Distinct strings	String occs	Average length	Volume (MB) of distinct	Volume (MB) total
TREC	1401774	752495240	5.06	7.68	4508.68
URLS	1265018	9987034	30.92	44.20	308.89
GENOME	262084	31623000	9.00	2.62	316.23
RANDOM	75000000	75000000	16.00	1290.00	1290.00
287721	287721	287721	7.16	2.34	2.34
909855	909855	909855	7.78	7.99	7.99
2877217	2877217	2877217	8.18	26.41	26.41
9098559	9098559	9098559	8.88	89.97	89.97
28772169	28772169	28772169	9.58	304.56	304.56

Table 6.1: Characteristics of the datasets used in our experiments. Our DISTINCT *dataset containing 28772169 strings was scaled down geometrically to create four distinct subsets of 9098559, 2877217, 909855, and 287721 strings.*

6.4 Experiments and results

We experimentally evaluate the performance of the B-trie for the task of storing and retrieving variable-length strings on disk. In this context, we compare the B-trie against a standard and prefix B^+-tree, as well as the Berkeley B^+-tree [Oracle, 2007], by measuring their memory and disk space consumption, insertion time, and search time. A standard B^+-tree stores full-length strings in internal nodes, in contrast to a prefix B^+-tree [Bayer and Unterauer, 1977], which only stores the shortest distinct prefix of strings that are promoted from leaf nodes. We also explore front-coding [Witten et al., 1999], as it has the potential to significantly increase the string capacity of nodes in a B^+-tree.

Other variants of B^+-tree, such as the SB-tree [Ferragina and Grossi, 1999] and the cache-oblivious string B-tree [Bender et al., 2006], are not suitable candidates for common string processing tasks. As discussed previously, the SB-tree operates poorly with short strings (which are less than 500 characters in length, for example) [Rose, 2000]. In addition, it is not well suited for tasks where the number of strings are not known in advance. For example, in order to be constructed and accessed efficiently, the SB-tree requires that strings are sorted beforehand, to permit bulk-loading and to improve the access locality amongst its string pointers [Ferragina and Grossi, 1999; Rose, 2000; Kärkkäinen and Rao, 2003]. Nonetheless, we consider a high-quality but static implementation of a SB-tree and compare its performance against our B-trie and the Berkeley B^+-tree. Similarly, the cache-oblivious string B-tree is currently a theoretical construct. It has yet to be implemented and there is currently no experimental evidence that supports its

performance against conventional disk-resident B$^+$-trees for strings [Bender et al., 2006; Brodal and Fagerberg, 2006; Ladner et al., 2002].

As test data, we used the string sets shown in Table 6.1 that were extracted from documents made available through TREC [Harman, 1995] and its GOV2 test collection. They are composed of null-terminated variable-length strings (up to a 1000 characters in length), in occurrence order — that is, they are unsorted. The TREC dataset is a set of word occurrences, with duplicates, extracted from the five TREC CDs [Harman, 1995]. This dataset is highly skew, containing a relatively small set of distinct strings. The URLS dataset, extracted from TREC web data, is composed of non-distinct complete URLs. We parsed the GOV2 test collection and acquired a dataset containing about 29 million distinct strings. We scaled it down geometrically, creating four DISTINCT subsets shown in Table 6.1. These DISTINCT datasets contain only unique strings; repeat occurrences were discarded. The GENOME dataset, extracted from GenBank, consists of fixed-length n-gram sequences with duplicates. Unlike the skew distributions observed in plain text however, these strings have a more uniform distribution. Finally, the RANDOM dataset, which was generated from a memory-less source, consists of fixed-length strings where each character is selected at random from the English alphabet. The RANDOM dataset contains no duplicate strings.

Our test machine was a Pentium IV 2.8 GHz processor, with 2 GB of RAM and a Linux operating system on light load using kernel 2.6.12. Time was measured in seconds, and we report the average *elapsed* time (or total time) required to complete a task, which we derive over a sequence of six runs. After each run, we unmount the hard drives to flush disk caches and flood main memory with random data. These steps are taken to ensure that the performance of the current run is not influenced by data cached from previous runs. Our hard drives were formatted using the *reiser* file system, a well-known Linux format. We tested *reiser* and found it to offer faster disk-writes and consume less space than the *ext2* and *ext3* formats, which are found by default on most Linux distributions. The relative performance of the B$^+$-trees and B-trie, however, remained the same regardless of file format.

Research on splay trees [Williams et al., 2001] reported the inefficiency of using the string-compare system call provided by the Linux operating system. String comparisons are a vital component of most string-based data structures. Williams et al. [2001] used their own implementation of string-compare and achieved speed gains of up to 20%. We do the same for our implementations. To further reduce resource contention on library calls, we implemented high-quality versions of `strlen`, `strcpy`, and `memcpy` (string length, string copy, and memory copy respectively). The data structures were written in C and compiled using `gcc 4.1.1`, with all optimizations enabled. We are confident — after extensive profiling — that our B$^+$-tree and B-trie implementations are of high quality, and as we discussed previously, we set the node size of the B-trie and B$^+$-trees to 8,192 bytes, which is known to offer good performance [Deschler and Rundensteiner, 2001; Gray and Graefe, 1997]. We consider the height of the B-trie or B$^+$-trees as the number of nodes accessed prior to reaching a container or leaf node, respectively.

6.4.1 The use of memory as cache

Our experimental evaluations of the B-trie and B$^+$-trees involve the use of an index buffer. An index buffer stores the internal nodes of a B$^+$-tree or the trie nodes of a B-trie in memory, to eliminate disk access on index traversal. Traversing a B-trie or B$^+$-tree will therefore incur only a single disk access to fetch the required leaf node or container from disk. The index buffer can grow to accommodate new nodes, however, as the index component of these data structures is typically only a tiny fraction of the size of the data used, the amount of memory required is small. The use of an index buffer is therefore a cheap and effective technique for reducing disk accesses without compromising data integrity — nodes that are modified in memory are immediately synchronized to disk, that is, the index buffer is effectively a *write-through* cache. Data structures that use an index buffer are labeled as *buffered*.

However, the use of an index buffer can cause unfair comparisons between the B$^+$-tree, which is balanced, and the B-trie, which is an unbalanced structure. That is, the B-trie is likely to benefit more from the buffer than the B$^+$-tree. Therefore, we also evaluate the performance of these data structures without the aid of a buffer. In this case, all nodes are accessed from disk and we do not explicitly buffer nodes for future reuse. Data structures that do not use an index buffer are labeled as *unbuffered*. The operating system, however, can also maintain its own private file buffers [Silberschatz et al., 2004], which we address by ensuring that every node is accessed by issuing a blocking system call to disk, and, by evaluating the performance of these data structures when their size exceeds the capacity of main memory.

Although not maintaining an index buffer is uncommon, it will show the worst-case performance of these data structures and allow for fairer comparisons. For instance, the cost of traversing an unbalanced trie will no longer be masked by the buffering of trie nodes in memory. An alternative buffering technique is the use of a shared buffer, which allocates a fixed-sized block of memory that stores both internal and leaf nodes. Once the buffer becomes full, however, a replacement algorithm is required to select and evict a node from memory, which is non-trivial.

Shared buffers are typically implemented using a *write-back* policy, where a modified node is only written to disk once it is evicted from the buffer. A shared buffer can be effective at reducing access costs, particularly under skew access. However, for this reason, they are unsuitable for use in experimental analysis, because they can lead to biased comparisons. The B-trie, for example, can become more compact than a B$^+$-tree, and is therefore likely to reside longer in the buffer prior to having its nodes evicted. In addition, the use of a large shared buffer will effectively treat a disk-resident data structure as an in-memory structure, masking almost all update and search costs. The Berkeley B$^+$-tree also employs a shared write-back buffer, but fortunately, its default size is small — only 256 KB — which we found to have little to no impact on performance for large datasets.

Dataset	**Build**		**Search**		**Disk-space**
	Buffered	*Unbuffered*	*Buffered*	*Unbuffered*	*(MB)*
					B-trie
287721	**3.6**	**4.6**	**1.4**	**2.5**	6.3
909855	**12.1**	**15.9**	**4.6**	**8.8**	19.9
2877217	**40.6**	**54.0**	**14.8**	**29.8**	**62.4**
9098559	**130.0**	**160.1**	**47.4**	**94.7**	**201.6**
28772169	**405.9**	**605.9**	**150.8**	**343.4**	**646.2**
					Standard B$^+$-tree
287721	3.6	5.0	1.5	3.6	6.0
909855	12.4	18.5	4.8	11.6	19.4
2877217	42.0	62.1	15.5	36.9	63.5
9098559	138.7	181.2	51.0	116.1	211.8
28772169	428.8	651.2	171.7	378.7	697.9
					Prefix B$^+$-tree
287721	3.6	4.8	1.5	3.6	**6.0**
909855	12.4	18.2	4.8	11.5	**19.3**
2877217	41.2	61.7	15.8	36.8	63.7
9098559	135.7	199.1	49.9	118.2	210.8
28772169	431.6	659.3	172.0	364.4	697.9
					Berkeley B$^+$-tree
287721	—	5.7	—	2.5	12.3
909855	—	20.7	—	9.1	40.7
2877217	—	71.0	—	32.0	132.6
9098559	—	237.1	—	110.6	436.8
28772169	—	867.9	—	720.7	1435.3

Table 6.2: A comparison of construction and self-search costs between the variants of B$^+$-trees and the B-trie, with and without an index buffer, using the DISTINCT *datasets of Table 6.1. The elapsed time is shown in seconds and the space in megabytes. The best measures of time and space are highlighted.*

6.4.2 Distinct strings

We measure the cost of construction by individually inserting strings, in occurrence order, into the B^+-trees and the B-trie. While this is a slow way to construct an index, it shows the per-string cost of maintaining the index during update. Table 6.2 shows the relationship between time and the number of distinct strings used for insertion and *self-search*. A self-search is the process of retrieving all strings that were stored by a data structure during construction. This process is useful in evaluating the performance of the data structures during search for known strings.

We first consider the performance of these data structures without an index buffer. The B-trie showed consistent improvement over the variants of B^+-tree, being up to 9% faster. The prefix B^+-tree is faster to build and search than the standard B^+-tree, as expected due to the storage of shorter strings within internal nodes. However, the prefix B^+-tree was not competitive in space. A higher fan-out per node will reduce the height of the tree, but as a result, more leaf nodes will be created.

The Berkeley B^+-tree showed relatively poor performance, requiring more time and space than our unbuffered B^+-trees and the B-trie. For example, with our largest DISTINCT dataset, the Berkeley B^+-tree was up to 52% slower to access than our unbuffered B-trie, while simultaneously requiring around 55% more space. We note, however, that the comparison of space is somewhat biased, due to the fact that Berkeley B^+-tree maintains a higher space overhead per node, in order to support more advance access routines such as concurrency control (which is beyond the scope of our work). As a result, the Berkeley B^+-tree created more internal and leaf nodes — and a subsequent increase in tree height — which resulted in its poor performance, as shown in Table 6.2.

The B-trie cannot match the space efficiency of our standard and prefix B^+-trees until enough trie nodes are created to increase the storage capacity of its containers (by stripping away common prefixes). This requires that a sufficient number of distinct strings are inserted to improve the space utilization within containers, which, in turn, reduce the number of splits that occur. From our results in Table 6.2, we observe that the B-trie needs around two million distinct strings to surpass the space efficiency of our B^+-trees, and improves thereafter, reaching up to a 7% reduction in space relative to the standard and prefix B^+-trees (with simultaneous improvements in access times).

The hash table had little influence on overall performance as only a tiny fraction of strings were hashed: 32,150 words of 28,772,169. A string can only be hashed if it is consumed by a trie node or by a pure container. The number of consumable strings is therefore bound by the number of trie nodes or pure containers. Thus, the hash table cannot grow large relative to the overall size of the B-trie, as shown in Table 6.3. Furthermore, the hash table is accessed only after a query string is consumed by the B-trie.

Despite using an unbalanced index that is accessed from disk, the B-trie remains efficient. Traversing a trie node is computationally inexpensive, requiring only a character as an offset.

Dataset	Trie nodes	Containers	Tree Height	No. strings compared *(Millions)*	No. strings stored in hash table	Index buffer *(MB)*	Total memory *(MB)*
							B-trie
287721	203	767	3.0	2.2	288	0.10	0.63
909855	580	2398	3.3	7.1	922	0.29	0.83
2877217	1851	7503	3.8	22.4	3324	0.94	1.53
9098559	6664	24189	4.4	71.0	10236	3.41	4.14
28772169	20915	77549	5.2	224.7	32150	10.70	11.87
							Standard B$^+$-tree
	Internal	**Leaves**					
287721	3	735	2	4.9	—	0.02	0.02
909855	8	2363	2	17.2	—	0.06	0.06
2877217	20	7740	2	59.5	—	0.16	0.16
9098559	71	25788	2	203.7	—	0.58	0.58
28772169	285	84917	2	692.9	—	2.33	2.33
							Prefix B$^+$-tree
287721	3	732	2	4.8	—	0.02	0.02
909855	5	2362	2	16.8	—	0.04	0.04
2877217	18	7767	2	58.2	—	0.14	0.14
9098559	70	25670	2	199.6	—	0.57	0.57
28772169	236	84964	2	680.2	—	1.93	1.93
							Berkeley B$^+$-tree
287721	6	1506	2	—	—	—	—
909855	20	4958	2	—	—	—	—
2877217	72	16124	2	—	—	—	—
9098559	218	53111	2	—	—	—	—
28772169	762	174448	3	—	—	—	—

Table 6.3: A comparison of structure size (height), memory consumption, and the number of string comparisons (hash table inclusive) performed by the B-trie and B$^+$-trees, when self-searching the DISTINCT *datasets of Table 6.1.*

Hence, a long chain of trie nodes can be traversed rapidly, allowing frequently accessed containers to be fetched at low cost. The B$^+$-tree, in contrast, must binary search every node that is accessed. Traversing a B$^+$-tree is therefore computationally expensive — an expense that is not entirely obscured by the costs of disk access.

Furthermore, trie nodes are 512 bytes long, making them sixteen times smaller than our B$^+$-tree nodes. Hence, access to a single block from disk will prefetch up to 16 trie nodes, which can improve both spatial access locality and the use of hardware disk buffers. Moreover, only 4 bytes are accessed from each trie node, unlike the binary search of a B$^+$-tree node, where typically most of the node is accessed. Hence, once brought into memory, tries are more cache-conscious due to their contiguous allocation.

The total binary search cost for the B$^+$-trees is the log of the number of stored strings. In contrast, the total binary search cost for the B-trie is constant, as it is limited to a single binary search. If the query string is consumed by the trie structure, then the cost of binary search is removed altogether. Furthermore, by removing lead characters during traversal, the single binary search only involves string suffixes. This leads to a reduction in the number of instructions executed, which contributes to the reduction in overall access time. This is reflected in Table 6.3, with the total number of string comparisons being significantly less for the B-trie, than the standard or prefix B$^+$-trees.

The major advantage of the B-trie is, however, the reduction in disk costs. To access a container, it is first read from disk, which — like the cost of a binary search — is avoided entirely if the query string is consumed before a container is accessed. Hence, the larger the B-trie, the greater the chance of avoiding a disk access during search. This demonstrates the implicit cost-adaptivity of the B-trie, which we expected to yield strong gains under skew.

Despite our efforts at limiting the number of trie nodes created, the space consumed by the B-trie's index exceeded that of the B$^+$-trees. However, because a trie-index removes common prefixes, fewer and more capacious containers are created, which compensates by reducing the overall disk space required, allowing the B-trie to be more compact overall than the B$^+$-trees. Having a larger index implies that more memory is used when we enable an index buffer. The amount of memory in question, however, remains small, requiring only around 9 MB more than the B$^+$-trees, for indexing over 304 MB of strings.

With the index buffer enabled, the B-trie and B$^+$-trees showed considerable improvements in performance. At the cost of a few megabytes of memory, the buffered B-trie can be constructed up to 22% faster and searched up to 56% faster than its unbuffered version. For example, with our largest DISTINCT dataset, the buffered B-trie required about 406 seconds to construct and 151 seconds to self-search, which is about 200 seconds and 193 seconds faster than the equivalent unbuffered B-trie, respectively. Similar behavior was observed for the standard and prefix B$^+$-trees, which were up to 34% faster to build and up to 55% faster to self-search, but remained slower to build and search than the buffered B-trie.

	Front-coded standard B$^+$-tree				
Dataset	**Construction** *(sec)*	**Self-search** *(sec)*	**Internal nodes**	**Leaf nodes**	**Disk-space** *(MB)*
287721	32.2	21.1	1	538	4.4
909855	105.1	70.2	5	1679	13.7
2877217	354.1	238.3	17	5391	44.3
9098559	1271.3	728.5	47	17142	140.8
28772169	3343.4	2338.3	152	55109	452.6

Table 6.4: Construction and self-search costs when front-coding is applied to the leaf nodes of the buffered standard B$^+$-tree, using the DISTINCT *datasets of Table 6.1. When front-coded, leaf nodes can store more strings prior to splitting which reduces the number of nodes maintained, but at a substantial cost in access time — being up to 93% slower than the uncompressed standard B$^+$-tree. Elapsed times are in seconds and space in megabytes.*

By buffering all of its trie nodes in memory, the B-trie is at a further advantage over the B$^+$-trees, as the computational cost required to reach a leaf node in a buffered B$^+$-tree will greatly exceed that of traversing a long chain of trie nodes. As a result, the average height of the B-trie can grow large at no consequence, apart from an increase in buffer space.

Although our results show that the B-trie is fast with or without an index buffer, we anticipate that the performance of the unbuffered B-trie will progressively deteriorate, relative to the unbuffered B$^+$-trees, as its average trie height increases. However, we revisit this issue below, in the context of a skew access pattern.

6.4.3 Front-coded B$^+$-tree

Front-coding can be used to increase the capacity of nodes in a standard or prefix B$^+$-tree. However, we do not expect an improvement in speed, despite the reduction in the number of nodes, due to the computational overhead of compressing and decompressing nodes on access. To test these claims, we have applied front-coding to the leaves of our buffered standard B$^+$-tree (the results are similar for the prefix B$^+$-tree). Front coding is a simple compression scheme that removes redundant prefixes in a sequence of sorted strings, and is capable of achieving over a 40% compression on sorted text datasets [Witten et al., 1999]. In this experiment, internal nodes remained uncompressed, as the overall space consumed by them is tiny relative to the space consumed by leaf nodes.

We repeated the insertion and self-search experiments as before, comparing the time and space required by our standard B$^+$-tree, with and without front-coding. The results are shown in Table 6.4. As anticipated, by front-coding leaf nodes, the cost of maintaining the B$^+$-tree

increased dramatically, being up to 93% slower for our largest DISTINCT dataset. Despite the cost in access time, the front-coded standard B$^+$-tree achieved up to a 35% reduction in space. For example, building a compressed standard B$^+$-tree using our largest DISTINCT dataset required over 3343 seconds and 453 megabytes of disk space. The equivalent uncompressed standard B$^+$-tree, in contrast, required only 429 seconds to build, but used over 646 megabytes of disk space. Decompression (and on modification, re-compression) are now mandatory tasks during tree traversal and, although fewer nodes are accessed, the computational cost of decompressing nodes can greatly exceed the cost of disk access, especially for large datasets. Front-coding can also be combined with bulk-loading, to speed up the cost of construction while conserving space. However, search will still remain expensive relative to a standard B$^+$-tree, due to the mandatory task of decompressing a node on access. Hence, the use of front-coding should only be applied to the B$^+$-tree when space is more valuable than access time.

6.4.4 Skewed search

In many applications such as text search, the ability to rapidly retrieve frequently accessed data is crucial. That is, the pattern of accesses is expected to be skew. To evaluate the performance of the B-trie and B$^+$-trees under skew access, we first construct these data structures using the DISTINCT datasets of Table 6.1. We then measure the time required to search for all strings in the TREC dataset as the size (the string cardinality) of the data structures increase, to determine their scalability. The results are illustrated in Figure 6.6. In addition, we measured the cost of skewed construction and self-search using the TREC dataset, shown in Table 6.5, which we discuss first.

Multi-way trie structures are among the fastest data structures under skew access, and the B-trie is no exception. The unbuffered B-trie was up to 33% faster (around 3170 seconds) to construct and self-search than the unbuffered standard and prefix B$^+$-trees. The Berkeley B$^+$-tree, in contrast, displayed competitive performance, being almost as fast as our unbuffered B-trie, but required more than twice the space. As we demonstrate later, however, the Berkeley B$^+$-tree does not scale well.

When constructed using the TREC dataset, the B-trie created 1009 trie nodes with an average trie height of about 3.6 nodes. The standard and prefix B$^+$-trees, however, created only 10 internal nodes with a balanced height of just 2. As a consequence, the B-trie required about 7% more disk space (or 2.2 MB more) than the standard and prefix B$^+$-trees. Nonetheless, its unbalanced and larger index had no impact on performance — with or without an index buffer — which is consistent with previous results. Furthermore, as we discussed in previous experiments, the B-trie can reduce its overall space consumption with an increase in the number of distinct strings stored.

When buffered, both the construction and self-search performance of the standard and prefix B$^+$-trees improve substantially, by as much as 59% (or around 5500 seconds) due to the elimination of disk access on index traversal. Similarly, the buffered B$^+$-trie also improves by as much as 56% (or around 3590 seconds), and remains faster to access than the buffered B$^+$-trees. These results

Dataset	Build		Search		Disk-space
	Buffered	*Unbuffered*	*Buffered*	*Unbuffered*	*(MB)*
					B-trie
TREC	**2904.9**	**6316.5**	**2748.5**	**6334.3**	33.3
URLS	**70.1**	205.7	**55.1**	201.0	91.1
GENOME	**160.3**	387.6	**156.7**	386.2	**4.3**
					Standard B$^+$-tree
TREC	3898.6	9396.1	3933.1	9506.3	**31.1**
URLS	71.8	143.6	60.8	133.2	75.8
GENOME	169.8	405.5	166.9	403.6	6.1
					Prefix B$^+$-tree
TREC	3871.1	9372.9	3893.8	9504.4	31.1
URLS	72.19	**141.8**	60.2	**131.5**	**75.1**
GENOME	170.1	391.0	167.6	389.8	6.1
					Berkeley B$^+$-tree
TREC	—	6390.4	—	6706.1	64.7
URLS	—	158.4	—	150.5	153.8
GENOME	—	**317.1**	—	**318.4**	13.7

Table 6.5: A comparison of construction and self-search performance of the B$^+$-trees and B-trie using the TREC, URLS, *and* GENOME *datasets of Table 6.1. The elapsed times are shown in seconds and the space in megabytes. The best measures of time and space are highlighted.*

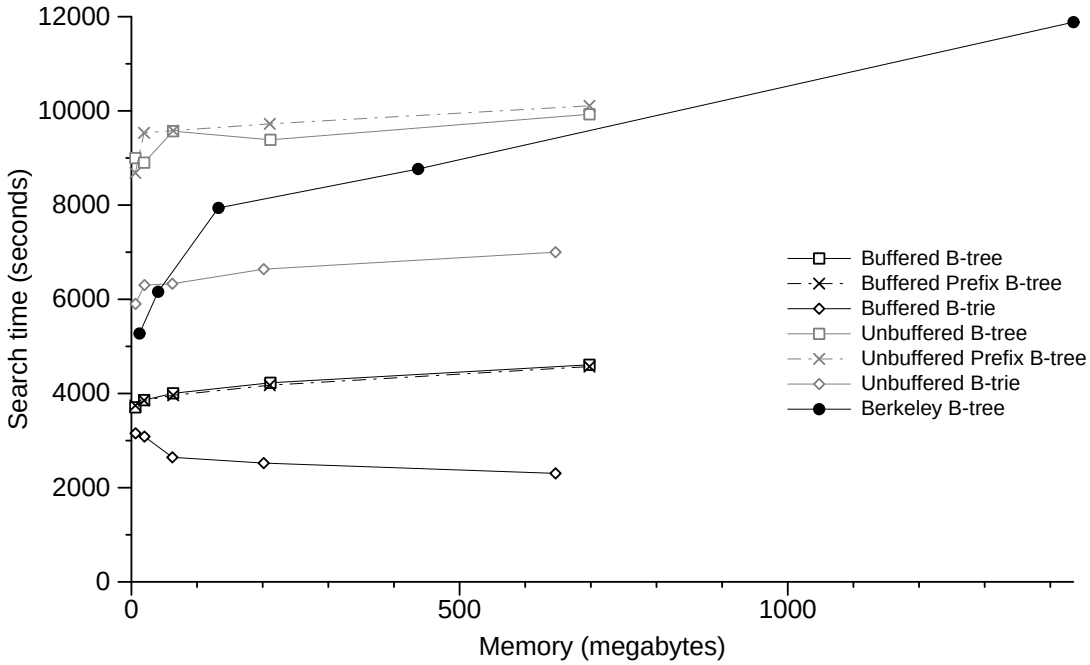


Figure 6.6: A comparison of skew search performance using the TREC *dataset, as the string cardinality of the data structures (the number of strings they store) increase. The* DISTINCT *datasets of Table 6.1 represent the points on the graph, with the left-most points representing our smallest* DISTINCT *dataset. For brevity, we label a B^+-tree as a B-tree in this figure.*

demonstrate that the B^+-tree is not efficient under skew access. With an index buffer enabled, every string searched will issue a system call to fetch a leaf node from disk. Hence, the number of disk accesses (or system calls) is determined by the number of query strings. Furthermore, every node accessed must be binary searched; a computational overhead that increases as the string cardinality of the B^+-tree increases. The B-trie however, requires at most, only a single (suffix-based) binary search per query, regardless of the size of its index. Traversing a B-trie is therefore far more computationally efficient than a B^+-tree.

We attribute the B-trie's superior performance primarily to the use of a trie-based index, albeit unbalanced. Trie nodes are sixteen times smaller than B^+-tree nodes, which improves spatial access locality resulting in better use of hardware buffers. They are also computationally efficient to traverse and strip away shared prefixes, which can result in the creation of fewer and more capacious containers. Furthermore, traversing a trie removes lead characters from a query string which can lead to its consumption, a phenomenon that becomes more frequent as the B-

trie increases in average height. In these cases, access to a container is avoided (assuming that the query string is consumed prior to accessing a container) and the search continues in the in-memory hash table. For example, during construction, the B-trie consumes 1612 strings from the TREC dataset, which are accessed over 230 million times during self-search. Without an index buffer, the B-trie can remain superior, as shown, due to its small average trie height. However, we anticipate that its performance will progressively deteriorate as its average height increases, which we demonstrate later.

Our next experiment evaluates the scalability of these data structures, by measuring the time required to search relative to their size. The results are illustrated in Figure 6.6. The buffered B-trie is clearly the fastest and most compact data structure when compared to the buffered B^+-trees, and improves in performance as the number of strings stored increase. The use of a buffer eliminates the disk costs incurred during trie traversal, at only a small cost in memory (up to 10 MB). Furthermore, as the average trie height increases, more strings are likely to be consumed during search, which will further reduce disk access. For example, having stored almost 29 million distinct strings, the B-trie reached an average trie height of 5.2 nodes (Table 6.3), and consumed almost 400 million queries. As a result, the buffered B-trie showed a substantial reduction in access time, due to the caching of short strings and the use of an index buffer. We observed up to a 50% improvement in speed (or around 2306 seconds), relative to the buffered versions of the standard and prefix B^+-trees, which are not as scalable.

Without an index buffer, however, the B-trie is likely to become more expensive to access as its size increases, because the caching of short strings in-memory cannot compensate entirely for the cost of traversing the trie on disk. Shown in Figure 6.6, the unbuffered B-trie was up to 67% slower (or around 4700 seconds) to access than the buffered B-trie. Nonetheless, by consuming short strings, the unbuffered B-trie incurred fewer disk accesses than the unbuffered standard or prefix B^+-trees, which were up to 31% slower (or around 3100 seconds) to access. The Berkeley B^+-tree showed good performance under skew for small indexes sizes, rivaling the performance of the unbuffered B-trie and B^+-trees. This behavior is consistent with the results observed previously in Table 6.5. However, as the number of strings stored increase, the performance of the Berkeley B^+-tree rapidly deteriorates, and becomes the slowest and most space-intensive data structure to search.

6.4.5 URLs

We repeated the experiments of construction and self-search using the URLS dataset, which is also highly skew. Unlike the strings found in the TREC dataset however, these strings are much longer, on average being around thirty characters. Thus, they require more space and can be more expensive to compare. We present the performance of the B^+-trees and the B-trie on construction and self-search, in Table 6.5.

Similar to what we observed in the previous TREC experiments, the buffered B-trie was the

fastest data structure to construct and self-search, being up to 9% faster (or about 6 seconds) than the standard or prefix B$^+$-trees. Despite its improved speed however, the B-trie required about 17% (or 16 MB) more space than the standard or prefix B$^+$-trees. Nonetheless, the B-trie remained more space-efficient than the Berkeley B$^+$-tree.

URLs typically share many long prefixes; http://www is by far the most common example. Use of long strings implies that fewer can be stored within containers prior to being split. B$^+$-tree nodes are also forced to split more frequently, but being a balanced structure, the B$^+$-tree will spread out considerably before increasing in height. As a result, the B-trie created 10,367 trie nodes with an average trie height of about 14 nodes. The B$^+$-trees, in contrast, maintained a balanced height of only 2 nodes. The standard B$^+$-tree created 77 internal nodes, whereas the prefix B$^+$-tree created only 51 internal nodes (which is equivalent in space to 816 trie nodes). The Berkeley B$^+$-tree created 144 internal nodes and 18,630 leaf nodes (9,450 more than the prefix B$^+$-tree). As a consequence, the Berkeley B$^+$-tree was the most space-intensive data structure.

Although the B-trie creates a relatively larger index, it is cheap to access — compared to the computational cost of traversing a B$^+$-tree — provided that it is buffered in memory. During the TREC experiments, the unbuffered B-trie retained superior performance because of its relatively small trie height, which can make good use of hardware buffers. In these experiments, however, although trie access is still skew, the URLS dataset forced the B-trie to create a much larger trie where, on average, 14 trie nodes are expected to be accessed before a container is acquired. This implies that on search, the B-trie may typically issue 15 system calls to disk (including one to access to a container), which is expensive. Hence, it was not surprising to observe a performance decline of up to 35% (or around 70 seconds), compared to the unbuffered standard and prefix B$^+$-trees. Furthermore, the caching of consumed strings did not compensate for the cost of maintaining a larger trie, as only 1150 strings were consumed which were accessed only 320,894 times during self-search.

The Berkeley B$^+$-tree was also faster to access than the unbuffered B-trie, but was slower than both the unbuffered standard or prefix B$^+$-trees. These experiments demonstrate that the B-trie is a fast and compact data structure — given enough distinct strings to make efficient use of containers — when its trie is buffered in memory. Without the aid of an index buffer, however, it can only remain superior to the unbuffered B$^+$-trees when maintaining a small average trie height.

6.4.6 Genome

Our next experiment involves the GENOME dataset, which contains fixed-length strings of strong skew. However, these strings are distributed much more uniformly than those of text, such as the TREC dataset. The time and space required to construct and self-search the B-trie and B$^+$-trees using the GENOME dataset, are shown in Table 6.5.

The buffered B-trie was the fastest data structure to construct and self-search, being up to 6% (or about 10 seconds) faster than the buffered standard or prefix B$^+$-trees. It also required the

Data structure	Build *(sec)*	Search *(sec)*	Tree height	Disk-space *(MB)*
B-trie	52546	124023	3	2150.3
Standard B^+-tree	69449	139003	3	2157.4
Prefix B^+-tree	68092	137124	2	2151.9
Berkeley B^+-tree	418112	549206	3	5638.7

Table 6.6: A comparison of the time and space required to construct and self-search the unbuffered B-trie and B^+-trees, using the RANDOM *dataset from Table 6.1. These results show the performance of the data structures when their size exceeds the capacity of main memory (2GB). Although these data structures are not explicitly buffered in-memory, the operating system can maintain its own private file buffers. However, in these experiments, the operating system is unable to buffer the entire data structure in-memory and must therefore rely on virtual memory. The elapsed times required to construct and self-search are shown in seconds, and the space in megabytes.*

least amount of disk space. However, as observed in previous experiments, the B-trie created a larger index of 341 trie nodes, in contrast to the standard and prefix B^+-trees that created only 3 internal nodes. Similarly, the Berkeley B^+-tree created only 9 internal nodes, but which resulted in a subsequent increase in leaf nodes, causing its overall space consumption to be the highest.

The unbuffered B-trie retained its speed over the unbuffered standard and prefix B^+-trees. However, without an index buffer, it was no longer the fastest. Instead, the Berkeley B^+-tree required the least amount of time to construct and self-search, despite its high space requirement. This behavior is consistent with results from previous experiments. For example, the Berkeley B^+-tree also showed good performance for searching the TREC dataset when having stored only a small number of distinct strings. Indeed, in these experiments, only 262,084 genome-strings were stored. However, having noted its behavior in previous experiments, it is reasonable to assume that the Berkeley B^+-tree will not scale well in both time and space, as the number of genome-strings stored increase.

6.4.7 Random

Our next experiment involves the use of the RANDOM dataset which was artificially created by selecting letters, at random, from the English alphabet, to form a large set of fixed-length strings. The purpose of this experiment is to grow the size of the B-trie and B^+-trees to beyond the capacity of the main memory, which in our case was 2 GB. We considered only the unbuffered data structures in these experiments to avoid masking the cost of accessing the index. In previous experiments, these data structures were small enough to reside entirely within main memory. Although we maintained and accessed them from disk, the underlying operating system can, to

some extent, buffer their files. Hence, although we issue system calls to fetch nodes from disk, some requests may actually be serviced from the underlying file buffers.

By growing the size of these data structures to beyond the capacity of main memory, however, we ensure that the operating system cannot buffer the data structures entirely within main memory. As a result, these experiments demonstrate the performance of these data structures when the operating system has insufficient resources to mask the cost of accessing disk. We present the time and space required to construct and self-search the unbuffered B-trie and the unbuffered B$^+$-trees in Table 6.6.

Despite having grown large, the unbuffered B-trie remained the fastest and most compact data structure to construct and self-search, being up to 87% faster (or 365,566 seconds) than the Berkeley B$^+$-tree, and up to 24% faster than the unbuffered standard and prefix B$^+$-trees. Its performance is attributed to the use a trie-index, as we explained in previous experiments. Although the trie-index was not explicitly buffered in memory, it remained efficient to access due to its relatively small height. Furthermore, no strings were consumed by the B-trie, so the in-memory hash table was unused.

The standard and prefix B$^+$-trees, though slower than the B-trie, were nonetheless greatly superior to the Berkeley B$^+$-tree, which required almost 5.5 GB of disk space, which is about 62% or 3.5 GB more than the B-trie and the standard and prefix B$^+$-trees.

6.4.8 String B-tree

We downloaded a high quality implementation of a SB-tree from the Internet[1]. As discussed in Section 6.1, the SB-tree represents the internal and leaf nodes of a B-tree as Patricia tries, which are stored succinctly on disk [Ferragina and Grossi, 1999]. However, the current implementation is that of a static SB-tree. That is, the strings used to build the SB-tree must be known in advance, and, once built, the entire data structure must be destroyed and re-built to accommodate new strings. Furthermore, to simplify the complexity of building and maintaining Patricia tries on disk, the SB-tree is built from the bottom-up, that is, it is bulk-loaded. Hence, the strings used to build the SB-tree must be sorted in advance.

We compare the performance of the SB-tree by measuring the time and space required to self-search using our DISTINCT datasets of Table 6.1. We do not consider the cost of construction due to requirement of bulk-loading. The Berkeley B$^+$-tree can be bulk-loaded, but we have yet to develop an efficient bulk-loading algorithm for the B-trie. Hence, both the Berkeley B$^+$-tree and the unbuffered B-trie were constructed from the top-down, using sorted versions of our DISTINCT datasets. We then measured the cost of self-search by using our original (unsorted) DISTINCT datasets. The time and space required to self-search the SB-tree, Berkeley B$^+$-tree, and unbuffered B-trie are shown in Figure 6.7.

[1]C++ String B-Tree Library, http://wikipedia-clustering.speedblue.org/strBTree.php

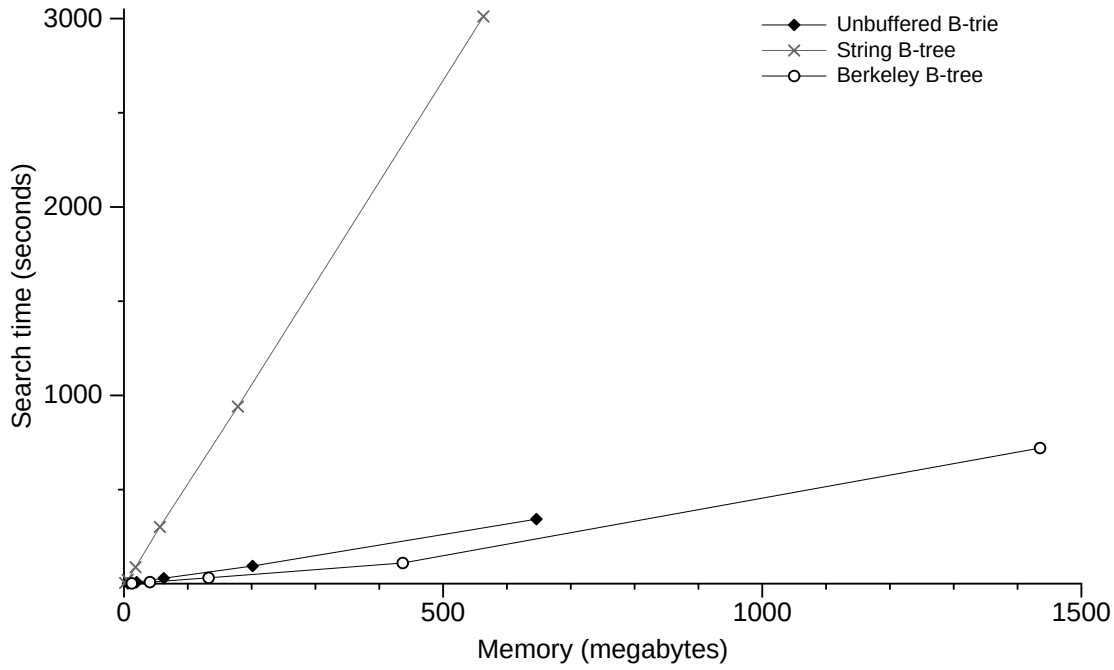


Figure 6.7: A comparison of the time in seconds and the space in megabytes required to self-search the SB-tree, the Berkeley B^+-tree, and the unbuffered B-trie, using the DISTINCT *datasets of Table 6.1. The left-most points, for example, represents the self-search cost of our smallest* DISTINCT *dataset. For brevity, we label a B^+-tree as a B-tree in this figure.*

No. strings deleted	Data Structure	Delete *(sec)*	Insert *(sec)*	Total time *(sec)*	No. nodes deleted	Total space *(MB)*
10 million	B-trie	**344.2**	**295.4**	**639.6**	149	**1072.4**
	Standard B$^+$-tree	364.9	362.0	726.9	0	1162.3
20 million	B-trie	**647.5**	**296.6**	**944.1**	537	**1043.7**
	Standard B$^+$-tree	678.8	374.3	1053.1	0	1139.7

Table 6.7: A comparison of the time and space required to delete and insert random strings from the unbuffered B-trie and unbuffered standard B$^+$-tree. The data structures were initially built using our largest DISTINCT *dataset from Table 6.1. The elapsed times required to construct and self-search are shown in seconds and the space in megabytes. The best measures are highlighted.*

Representing the nodes of a B$^+$-tree as Patricia tries can increase their string capacity, resulting in fewer nodes which saves space. As a result, the SB-tree was up to 61% more compact than the Berkeley B$^+$-tree, a saving of up to 872 MB of disk space. The SB-tree was also more compact than our unbuffered B-trie, but only by up to 13% or 84 MB, due to space saved by removing shared prefixes in containers. Although space-efficient, the SB-tree showed relatively poor performance. Access to a node in a SB-tree incurred up to two disk reads: one to read the node from disk, and then another to fetch the required string suffix for comparison. Furthermore, processing a node which involves traversing a Patricia trie, is computationally expensive compared to the comparison-less traversal of the array trie used by the B-trie. Hence, in these experiments — which involved strings with an average length of less than 10 characters — the SB-tree was up to 76% slower (or around 2290 seconds) to search than the Berkeley B$^+$-tree, and up to 89% slower (or around 2668 seconds) to search than our unbuffered B-trie. These results are consistent to those reported by Rose [2000], who claimed that the Berkeley B$^+$-tree was consistently faster to access than the SB-tree with short strings. These results demonstrate that the overall space saved by mapping a space-efficient trie structure to disk, such as the Patricia trie, can be small relative to the space consumed by the equivalent B-trie that employs a fast array trie, which is kept small in size by the use of containers.

6.4.9 Deletion

Our final experiment compares the cost of deletion between the standard B$^+$-tree and B-trie. We implemented lazy deletion, that is, when a node has had all of its strings deleted, it is not physically

deleted. Instead, its address is posted for reuse. Our first experiment measures the cost of deleting 10 million strings from the B$^+$-tree and B-trie, which were built using our largest DISTINCT dataset. The strings to delete were selected at random from this dataset. Our second experiment repeats the first, but with twice as many deletions. Once the strings have been deleted, we measure the time and space required to insert a further 15 million strings taken from the RANDOM dataset (no random strings were consumed by the B-trie). By inserting randomly generated strings, each leaf node and container in the B$^+$-tree and B-trie, respectively, has equal probability of access, that is, there is no skew.

In these experiments, we use the unbuffered standard B$^+$-tree and unbuffered B-trie to avoid masking the cost of accessing their index. For brevity, we omit results from the unbuffered prefix B$^+$-tree as they were similar to those of the standard B$^+$-tree. Similarly, we omit the Berkeley B$^+$-tree, as its performance was found to consistent with previous experiments; that is, it was slow and space-intensive relative to the standard B$^+$-tree and B-trie. The results are shown in Table 6.7.

The unbuffered B-trie was, in both experiments, faster than the standard B$^+$-tree for the deletion of strings and the subsequent insertion of random strings. The standard B$^+$-tree was slower due to the computational cost of binary search. However, with a balanced structure and considering that strings were randomly selected for deletion, no leaf nodes were deleted. The B-trie, in contrast, deleted up to 537 containers (no trie nodes were deleted). The B-trie is an unbalanced structure, and considering that containers can be unevenly split, more containers are likely to be deleted than the nodes of a B$^+$-tree. However, this is of no consequence when lazy deletion is employed. With the subsequent insertion of 15 million strings, for example, all 537 containers were reused, showing no impact on performance relative to standard B$^+$-tree.

Without lazy deletion, however, deleting strings in a B-trie can be more expensive than in a B$^+$-tree. Assuming that no containers are merged (which is the case in these experiments), the B-trie would have had to physically delete up to 537 containers on disk, which is expensive. The standard B$^+$-tree, however, having deleted no leaf nodes, would be more efficient in this case.

6.5 Summary

Many applications require efficient storage and retrieval of strings on disk. However, due to the high cost associated with disk usage, the range of efficient and viable data structures for this task is limited. The B$^+$-tree and its variants have been the most successful data structures on disk for the task of sorted string management, employing a balanced tree that guarantees a bounded worst-case cost, regardless of the distribution of data.

A B-trie is an alternative trie-based structure that has potential to be superior to the B$^+$-tree, but has yet to be formally described or evaluated in literature. In this chapter, we have described our novel variation of the B-trie, by proposing new string insertion, deletion, equality search, and node splitting algorithms that are designed to make efficient use of disk, for common string processing tasks such as vocabulary accumulation. Our variant of B-trie is effectively a novel application of burst trie to disk. Existing disk-resident tries, such as the suffix tree, are not practical solutions for common string processing tasks, due to their high space and update costs [Grossi and Vitter, 2000; Tian et al., 2005; Cheung et al., 2005].

We ran a series of experiments to compare the B-trie to the Berkeley B$^+$-tree [Oracle, 2007] and our own high performance implementation of a standard B$^+$-tree (where internal nodes store full-length strings), and a prefix B$^+$-tree [Bayer and Unterauer, 1977], using string datasets extracted from documents made available through real-world data repositories, such as TREC [Harman, 1995]. We considered alternative data structures such as the string B-tree [Ferragina and Grossi, 1999] and the cache-oblivious string B-tree [Bender et al., 2006], but which were found to be unsuitable for common string processing tasks.

We compared the time and space required to store and retrieve strings from a variety of sources and string distributions, and evaluated the scalability of these data structures under skew access. The B-trie was found to be superior in both time and space, offering performance gains of up to 50% when its trie is buffered in memory. There were cases where the B-trie required more disk space than standard and prefix B$^+$-trees, but with no impact on speed. Furthermore, the amount of excess disk space required was small — in the order of a few megabytes. The Berkeley B$^+$-tree, however, was the largest data structure, and was also — in the majority of cases — the slowest to access.

A minor drawback of the B-trie is that it generates a larger index than that of the standard and prefix B$^+$-trees. The difference in space, however, is small, as our novel B-trie splitting algorithm successfully throttles the creation of containers and trie nodes. Furthermore, the removal of shared prefixes in containers can compensate (in space) by allowing the B-trie to consume less disk space in total, than the variants of B$^+$-trees for large numbers of insertions.

The consequence of maintaining a larger index, however, became apparent when we disabled the index buffer, which is an unlikely decision in practice due to the high costs of disk access. In these cases, the unbuffered B-trie could only remain superior to the unbuffered B$^+$-trees, when

maintaining a small average trie height, which is generally sustained under skew access. Therefore, with the use of a small index buffer and for the task of managing a large set of strings on disk, we have shown the B-trie to be a superior data structure, being faster and more scalable than common variants of B^+-tree that are currently in standard use.

Chapter 7

Conclusion and future work

Data structures are fundamental tools used in computing applications that require efficient storage and retrieval of data. Much of the research involving string data structures, however, has focused on reducing algorithmic complexity. While reducing complexity or the number of instructions executed is of value, research has shown that on current processors, the number of instructions executed can be increased, if doing so improves the performance of a data structure through a reduction in cache misses. Most existing string data structures, however, assume a non-hierarchical RAM model, which states that all memory addresses are accessible and are of uniform cost. These assumptions are not realistic on current computing architectures, which typically employ a hierarchy of caches to hide the high access cost of main memory.

Caches are typically several orders of magnitude faster than main memory and are not accessible by software — instead being administrated by the underlying hardware. As a result, most existing string data structures are oblivious to cache, and despite efforts at reducing the number of instructions executed, they can attract excessive cache misses. The current best data structures for strings in the research literature are the chaining hash table with move-to-front, the burst trie, and the binary search tree. These data structures are fast because they minimize the number of instructions executed. However, they assume a non-hierarchical RAM model and can therefore be inefficient in their use of cache.

7.1 Contributions and results

We have explored novel techniques at improving the cache-efficiency of the three current best string data structures, by re-considering the principles of their design in order to exploit cache without compromising their dynamic characteristics. Our results show that the standard representation of these data structures, where strings are stored and accessed using linked lists of fixed-size nodes (a standard chain), consisting of a string pointer and a node pointer, is neither cache nor

Data storage	Hash table	Burst trie	BST
Standard	2380.6	63.8	758.6
Compact	2082.1	54.1	594.8
Array	199.1	23.9	495.4

Table 7.1: A comparison of the total time, in seconds, required to build a hash table, burst trie, and BST, using almost 30 million distinct strings. The results compare the difference in cost between the standard, compact, and array representations of storing data. The array-based versions, in all cases, showed considerable reductions in build time, despite the added computation cost of array resizing.

space efficient. Pointers are the principal cause of cache-inefficiency, and their elimination in data structures and programs is key to the successful use of cache. Pointers often cause random access to memory, which can render a cache useless. In addition, the address they point to cannot be known in advance, which hinders the effectiveness of hardware prefetchers. Yet the elimination of pointers in dynamic string data structures — in particular the hash table, burst trie, and BST — has attracted virtually no attention in the research literature. Hence, the pointer elimination techniques presented in this thesis are a major contribution and have led to significant reductions in both time and space.

We proposed new representations for managing collisions in string hash tables, and new representations for storing strings in a burst trie, which make good use of cache without compromising their dynamic characteristics. Similarly, we proposed a novel representation of a dynamic binary search tree for strings that can successfully exploit cache. In every case, the simple alternative of replacing the fixed-length string pointer with the variable-length string, yielding what we call a compact chain, proved faster and smaller. We proposed 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. While our proposal might be seen as simplistic, it is nonetheless attractive in the context of current architectures. Furthermore, with nodes and pointers eliminated, a substantial amount of space can be saved, at no cost.

To demonstrate the significance of pointer elimination, we present a sample of the results discussed in Chapter 4 in Table 7.1, which shows the elapsed time required to build a hash table, burst trie, and BST using their standard, compact, and array representations, and almost 30 million unique strings. Although arrays require more instructions — whenever a string is inserted, the array must be resized — no pointers are traversed. Instead, the array is simply scanned (with word-skipping) from left-to-right, which makes good use of both cache and hardware prefetchers. This results in a substantial reduction in cache misses, which greatly compensates for the added

computational cost associated with array resizing. As a result, we witnessed a dramatic reduction in build time, with the array hash table being up to 92% faster to construct than its standard representation. Similar results were observed for the burst trie and BST.

For the hash tables and binary search trees, in most cases, the difference in speed compared to the compact chain is small, but the space savings are large; in the best case for hashing, the total space overhead was less than a bit per string, a reduction from around 100 bits for compact chaining, and around 200 bits for standard 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 for hash tables 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. When sort order is required or when space is of primary interest, our cache-conscious or array burst trie showed strong and consistent improvements over its chained variants, being up to 74% faster. In all cases, the array burst trie required the least amount of space, while approaching the access speed of the array hash table. Although not as fast as the burst trie, the array BST also showed similar improvements, being up to 35% faster than the standard BST while requiring only around third of its space. These results demonstrate the importance of considering the impact of caching on data structures that are otherwise computationally efficient.

In a burst trie, it is often preferable to conserve space by using large containers. However, large containers are expensive to access. In a chaining burst trie (with standard or compact chains), large containers — which store over 100 strings for example — are computationally efficient, but can incur high numbers of cache misses. In an array burst trie, in contrast, large containers (dynamic arrays) are cache-efficient but require more instructions. We have shown that the high reduction of cache misses greatly compensate for the increase of instructions, allowing the array burst trie to sustain superior performance. Nonetheless, as the size of containers increase, the array burst trie became, in most cases, slower to access relative to smaller containers. Although larger containers further reduced the number of cache misses incurred, their computational cost of access grew to such an extent that it could no longer be masked entirely by the reduction of cache misses. This demonstrated that, in order to develop a scalable data structure on current processors, it is necessary to reduce cache misses while sustaining low instruction costs, relative to that of chaining.

We therefore introduced a new in-memory data structure for strings, called a HAT-trie, which is a variant of the burst trie that combines the array hash table and burst trie, to yield a fast, compact, and scalable data structure that can exploit cache while sustaining sorted access to containers. The principle idea of the HAT-trie was to change the structural representation of containers, from arrays to array hash tables. Although some space is wasted as a result —

containers must now reserve space for slots — this approach allows containers to grow sufficiently large, while sustaining low cache and instruction costs.

We experimentally compared the HAT-trie to the standard, compact, and array representations of the hash table, burst trie, and BST, to measure the time and space required to construct and search, using large string datasets with a range of characteristics. The HAT-trie was found to be up to 80% faster than the chaining burst tries, and in most cases was faster to construct and search than the array burst trie, by as much as 50%. Although the HAT-trie could not match the space efficiency of the array burst trie, it nonetheless remained competitive, almost matching the speed and space consumption of the array hash table while maintaining sorted access to containers. The chaining hash tables could only compete in speed once given excessive space, while the binary search trees proved to be too inefficient in most cases. To our knowledge, the HAT-trie is the first trie-based data structure that can approach the speed and space efficiency of hashing, while maintaining sorted access to strings. These are strong results that further substantiate the effectiveness of using dynamic arrays in the structural design of pointer-intensive data structures that are otherwise computationally efficient.

These cache-conscious data structures, however, were not designed to cater for situations where main memory is exhausted, as they are oblivious to the high costs of disk access. In such cases, data needs to be maintained efficiently on disk. We therefore extended our research to disk-resident data structures for strings. Having observed the high performance of the array burst trie, and the benefits thereof — such as the removal of shared prefixes, an implicit cost-adaptive structure, and sorted access to strings — the development of a disk-resident burst trie was attractive. However, the burst trie was not a suitable candidate for disk, due to the high space and update costs caused by bursting containers. Similarly, alternative disk-resident tries such as variants of suffix trees, were incompatible for common string processing tasks such as vocabulary accumulation, text indexing, or dictionary management, due to their high space and update costs on disk.

We thereby proposed a trie-based data structure for disk, called a B-trie, which is structurally similar to a burst trie except that it employs a new node splitting routine, called a *B-trie split*, which is a major contribution as it solved the difficult problem of maintaining a trie efficiently on disk, for common string processing tasks. The data structure we proposed is a novel variation of a B-trie, with new string insertion, deletion, equality search, and node splitting algorithms that were previously unavailable in research literature; the existing B-trie is simply a theoretical construct denoting a static trie indexing a set of containers that can store up to b items.

We experimentally compared the performance of our B-trie against variants of B^+-trees that have been the most successful data structures for managing strings on disk. With an index buffer, the B-trie was shown to be consistently faster than the variants of B^+-trees, and being up to 50% faster under skew access. The minor drawback of the B-trie was that it created a larger index than that of the B^+-trees. However, the difference in space was small — only a few megabytes. In most cases, the overall disk space required by the B-trie remained smaller due to the removal of shared

prefixes in containers. There were cases, however, where the B-trie required more disk space, but these were rare and involved only a few extra megabytes, when compared to the equivalent B^+-trees. Without an index buffer — which is an unlikely omission in practice — the B-trie, being unbalanced, could only remain superior to the B^+-trees when maintaining a small average trie height, which was generally sustained under skew access. At the cost of reserving a few megabytes of main memory as an index buffer, the B-trie was shown to be a superior data structure, being faster and more scalable than the common variants of B^+-tree that are currently in standard use.

7.2 Lessons learnt

We learnt several lessons from our work that can be applied to future work on data storage and retrieval algorithms. Specifically:

- To realize good performance on current computing architectures, it is of paramount importance that data structures and programs reduce or eliminate the use of pointers.
- Techniques such as clustering, which attempt to re-organize the physical location of nodes in a data structure — to improve cache utilization — are only effective when no skew is present in the data distribution. Implicit clustering, where nodes are clustered and accessed via arithmetic offsets, is slightly more efficient than clustering alone, but showed equally poor performance under skew.
- To yield high gains in performance in dynamic string data structures, such as the hash table and burst trie, we must combine clustering with pointer elimination by storing and accessing strings within dynamic arrays.
- Replacing a linked list with a dynamic array is not recommended, however, when the list contains all of the strings used. Despite the reduction in cache misses, our results have shown that a dynamic array can be slower to access in this case, due to its high computational costs. In such cases, the compact chain is preferable.
- Dynamic arrays, however, are exceptionally useful at improving the cache-efficiency of pointer-intensive data structures, such as the hash table, burst trie, and the BST. The key distinction here, is that strings are likely to be scattered across a set of linked lists. In these cases, nodes are likely to be scattered in memory, which can dramatically reduce the effectiveness of both cache and hardware prefetch.
- Space efficiency is another key attribute to performance. A smaller data structure can have a larger portion of it nodes in cache and can store more strings before exhausting main memory. However, conventional space reduction methods, such as front-coding, often reduce space at the expense of access time. By eliminating pointers, we can greatly improve cache utilization while simultaneously saving a considerable amount of space, at no cost.

- To achieve a fast and scalable data structure, we must reduce cache misses while conserving the number of instructions executed, although this usually involves sacrificing some space. For example, resolving collisions in a string hash table by using compact chains with move-to-front on access, will often be slightly faster than dynamic arrays under skew access, but at the expense of space.
- Move-to-front on access is effective for chains but not for dynamic arrays, due to the requirement of physically moving a string to the start of the array.
- As the speed of processors continue to advance ahead of main memory, instructions will become cheaper to execute while cache misses are anticipated to become more expensive. Hence, we expect that on future processors, the use of dynamic arrays will yield further gains in performance.
- The B-trie showed us that a trie structure, albeit unbalanced, can operate efficiently on disk for common string processing tasks, particularly under skew access.

7.3 Future work

In this section, we discuss some open research topics that can improve and optimize the cache-conscious data structures presented in this thesis. Although the HAT-trie was shown to be the fastest in-memory trie structure for strings, it has considerable scope for improvement, particularly with regards to space consumption. We discuss some variants of HAT-trie that have promising prospects for future work.

HAT-D: The HAT-trie can be adapted for disk, forming a variant called a HAT-D. The HAT-D is structurally similar to a B-trie, except that containers are partitioned into fixed-sized regions or slots. The goal is to allow containers to remain sufficiently large on disk, without attracting high access costs. Once the base address of a container is known, the query string can be hashed and the required partition is fetched from disk. We expect the HAT-D to be more scalable than the B-trie, reducing both time and space. However, work is needed to determine how containers are split efficiently on disk. For example, once a partition becomes full, do we split the entire container which can be costly, or is there a way to split individual partitions?

Bulk-loading the B-trie: Can the B-trie be bulk-loaded? Bulk-loading involves taking advantage of sorted data to rapidly build a tree-based data structure from the bottom-up, to avoid consulting an index. However, bulk-loading assumes that the index is independent from the data, which is not the case for the B-trie, as its trie-based index removes lead characters from strings.

Multi-processor technology: The data structures explored in this thesis are designed for single-processor environments. However, recent hardware trends have shown that multi-core processors are likely to become commonplace as a way of alleviating the growing cost of accessing memory, by improving processor throughput. Hence, it is of value to find a way to efficiently adapt the cache-conscious data structures presented in this thesis, to exploit multi-core processors. Similarly, programmable graphic processing units (GPUs) are gaining popularity, and may also be used to improve the performance of string data structures.

Fixed-length keys: The cache-conscious data structures presented in this thesis can be readily converted to store fixed-length keys, such as integers, and we expect to see consistent gains in performance compared to their chained variants. Furthermore, the performance of these data structures can be compared against existing fixed-length key data structures, such as the in-memory cache-conscious B^+-tree.

7.4 Final remarks

When strings need to be stored and retrieved from disk, such as maintaining a vocabulary in a search engine, for text indexing, or dictionary management, our B-trie can be a superior alternative to the variants of B^+-trees that are in standard use. We have shown that in order to develop a fast, compact, and scalable in-memory data structure for current cache-oriented processors, it is imperative that the use of pointers is reduced or eliminated, by storing and accessing data items within dynamic arrays. We believe that our array hash table, array burst trie, and HAT-trie, are the best choice of string data structures for virtually any computing application that requires efficient storage and retrieval of strings in memory.

Appendix A

Results on a Sun Ultra Sparc Server

We compare the performance (with respect to both time and space) of a selection of data structures — namely the TST, hash table, burst trie, and HAT-trie — for the task of self-searching the highly skew TREC dataset (discussed in Chapter 4), which contains about 178 million strings but of which only around 622 thousand are distinct. The machine used for this experiment was a Sun Ultra Sparc server running a Solaris (release 5.10) operating system. The server has 16GB of RAM with 2 Ultra Sparc III processors running at 1.33Ghz. Our goal here is to provide a brief demonstration on the effectiveness of our cache-conscious data structures on an alternative computing architecture (that is, other than an Intel).

In Chapters 4 and 5, we showed that for a variety of string datasets, the array burst trie, HAT-trie, and array hash table were the fastest and most compact in-memory string data structures. The performance of these data structures was credited to the elimination of nodes and their pointers via the contiguous allocation of strings. As such, we expect to observe similar results on an Ultra Sparc server, as well as other computing architectures such as an AMD processor, Intel dual core, and a PowerPC.

The results are shown in Figure A.1. As expected, the array burst trie and array hash were markedly superior in both time and space relative to their chained variants. The HAT-trie also displayed strong results, outperforming the array burst trie and rivaling the speed and space consumption of the array hash table.

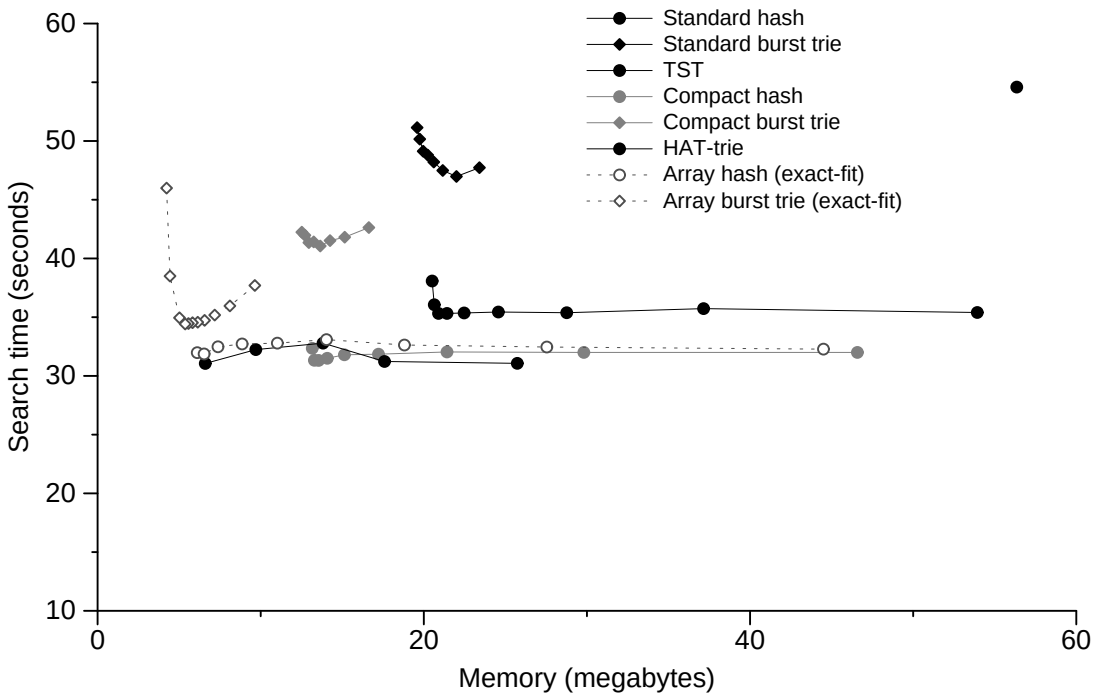


Figure A.1: The time and space required to self-search the data structures using the TREC *dataset on a Sun Ultra Sparc Server (running Solaris 5.10). The points on the graph represent the container thresholds for the burst trie, and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of* 10 *from* 30 *to* 100*. For the array burst trie, the sequence is extended to include* 128*,* 256*, and* 512 *strings. The hash tables commenced with* 2^{15} *slots, doubling up to* 2^{23}*. The HAT-trie commenced with a container threshold of* 2048*, doubling up to* $32,768$*.*

Bibliography

A. Acharya, H. Zhu, and K. Shen. Adaptive algorithms for cache-efficient trie search. In M. T. Goodrich and C. C. McGeoch, editors, *Proc. ALENEX Workshop on Algorithm Engineering and Experiments*, pages 296–311, Baltimore, Maryland, United States, January 1999. Source code at http://www.cs.rochester.edu/~kshen.

G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 16(2):263–266, 1962.

R. Agarwal. A super scalar sort algorithm for RISC processors. In H. V. Jagadish and I. S. Mumick, editors, *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, pages 240–246, Montreal, Quebec, Canada, June 1996.

A. Aggarwal. Software caching vs. prefetching. In *Proc. Int. Symp. on Memory Management*, pages 157–162, Berlin, Germany, June 2002.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Boston, Massachusetts, United States, first edition, 1974. ISBN 0201000296.

A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Addison-Wesley, Boston, Massachusetts, United States, first edition, 1986. ISBN 0201000237.

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

R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, California, United States, first edition, 2001. ISBN 1558602860.

D. Anderson and T. Shanley. *Pentium Processor System Architecture*. Addison-Wesley, Boston, Massachusetts, United States, second edition, 1995. ISBN 0201409925.

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

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

L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In S. G. Akl, F. K. H. A. Dehne, J. Sack, and N. Santoro, editors, *Proc. Int. Workshop on Algorithms and Data Structures*, pages 334–345, Kingston, Ontario, Canada, August 1995.

L. Arge. External memory data structures. In J. M. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357, Norwell, Massachusetts, United States, 2002. Kluwer Academic Publishers. ISBN 1402004893.

L. Arge, M. A. Bender, E. Demaine, C. Leiserson, and K. Mehlhorn. Abstracts collection. In *Cache-Oblivious and Cache-Aware Algorithms*, number 04301 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, 2005a.

L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. P. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*, pages 34–1. CRC Press, 2005b. ISBN 1584884355.

D. M. Arnow and A. M. Tenenbaum. An empirical comparison of B-trees, Compact B-trees and Multiway trees. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, pages 33–46, Boston, Massachusetts, United States, June 1984.

D. M. Arnow, A. M. Tenenbaum, and C. Wu. P-trees: Storage efficient multiway trees. In J. M. Tague, editor, *Proc. ACM SIGIR Int. Conf. on Research and Development in Information Retrieval*, pages 111–121, Montreal, Quebec, Canada, June 1985.

D. F. Bacon, S. L. Granham, and O. J. Sharp. Compiler transformation for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

A. A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimization. In *Proc. Int. Conf. on Supercomputing*, pages 486–500, Sorrento, Italy, June 2001.

A. A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism*, 6 (7), 2004.

G. Badr and B. J. Oommen. On using conditional rotations and randomized heuristics for self-organizing ternary search tries. In *Proc. of ACM Southeast Regional Conf.*, pages 109–115, Kennesaw, Georgia, United States, March 2005.

J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In J. L. Martin, editor, *Proc. ACM/IEEE Conf. on Supercomputing*, pages 176–186, Albuquerque, New Mexico, United States, November 1991.

J. Baer and T. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

R. A. Baeza-Yates. An adaptive overflow technique for B-trees. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proc. Int. Conf. on Extending Database Technology*, pages 16–28, Venice, Italy, March 1990.

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

R. A. Baeza-Yates and P. A. Larson. Performance of B^+-trees with partial expansions. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):248–257, 1989.

D. Baskins. A 10-minute description of how Judy arrays work and why they are so fast, 2004. URL http://judy.sourceforge.net/.

R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

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

T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, New Jersey, United States, first edition, 1990. ISBN 0139119914.

T. C. Bell, A. Moffat, I. H. Witten, and J. Zobel. The MG retrieval system: Compressing for space and speed. *Communications of the ACM*, 38(4):41–42, 1995.

Y. Ben-Asher, E. Farchi, and I. Newman. Optimal search in trees. *SIAM Journal of Computing*, 28(6):2090–2102, 1999.

M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Foundations of Computer Science*, pages 399–409, Redondo Beach, California, United States, November 2000.

M. A. Bender, E. D. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In R. H. Mohring and R. Raman, editors, *Proc. European Symp. on Algorithms*, pages 165–173, Rome, Italy, September 2002.

M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 53(2):115–136, 2004.

M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 233–242, Chicago, Illinois, United States, June 2006.

M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *ACM Symp. on Parallel Algorithms and Architectures*, pages 81–92, San Diego, California, United States, June 2007.

J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, United States, January 1997.

E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proc. of ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 1–12, Seattle, Washington, United States, November 2002.

K. Beyls and E. H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proc. Conf. on Computing Frontiers*, pages 373–382, Ischia, Italy, May 2006.

K. Boland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, 1994.

R. P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

G. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In C. Stein, editor, *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 581–590, Miami, Florida, United States, January 2006.

G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In D. Eppstein, editor, *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 39–48, San Francisco, California, United States, January 2002.

D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proc. Int. Symp. on Computer Architecture*, pages 78–89, Philadelphia, Pennsylvania, United States, May 1996.

B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, California, United States, October 1998.

D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, United States, April 1991.

S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, United States, October 1994.

Y. Chang, C. Lee, and W. ChangLiaw. Linear spiral hashing for expansible files. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):969–984, 1999.

R. P. Cheetham, B. J. Oommen, and D. T. H. Ng. Adaptive structuring of binary search trees using conditional rotations. *IEEE Transactions on Knowledge and Data Engineering*, 5(4): 695–704, 1993.

C. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.

T. M. Chilimbi. *Cache-Conscious Data Structures–Design and Implementation.* PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1999.

T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 252–262, Ottawa, Ontario, Canada, 2006.

T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, Atlanta, Georgia, United States, May 1999a.

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, Atlanta, Georgia, United States, May 1999b.

T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

E. I. Chong, J. Srinivasan, S. Das, C. Freiwald, A. Yalamanchi, M. Jagannath, A. Tran, R. Krishnan, and R. Jiang. A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary B^+trees. *ACM SIGMOD Record*, 32(2):78–88, 2003.

N. M. M. K. Chowdhury, M. M. Akbar, and M. Kaykobad. DiskTrie: An efficient data structure using flash memory for mobile devices. In M. Kaykobad and M. S. Rahman, editors, *Workshop on Algorithms and Computation*, pages 76–87, Bangladesh Computer Council Bhaban, Agargaon, Dhaka, Feburary 2007.

V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *IEEE Symp. on the Foundations of Computer Science*, pages 219–227, Vancouver, British Columbia, Canada, November 2002.

V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. A data structure for a sequence of string accesses in external memory. *ACM Transactions on Algorithms*, 3(1):6, 2007.

D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 383–391, Atlanta, Georgia, United States, January 1996.

J. Clement, P. Flajolet, and B. Vallee. The analysis of hybrid trie structures. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 531–539, San Francisco, California, United States, January 1998.

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

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, Istanbul, Turkey, November 2002.

D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979a.

D. Comer. Heuristics for trie index minimization. *ACM Transactions on Database Systems*, 4(3): 383–395, 1979b.

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

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

A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. In J. Nesetril, editor, *Proc. of European Symp. on Algorithms*, pages 224–235, Prague, Czech Republic, July 1999.

P. Crescenzi, R. Grossi, and G. F. Italiano. Search data structures for skewed strings. In K. Jansen, M. Margraf, M. Mastrolilli, and J. D. P. Rolim, editors, *Experimental and Efficient Algorithms: Second Int. Workshop, WEA*, pages 81–96, Ascona, Switzerland, May 2003.

K. Culik, T. Ottmann, and D. Wood. Dense multiway trees. *ACM Transactions on Database Systems*, 6(3):486–512, 1981.

R. de la Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, pages 295–298, New York, United States, 1959.

R. H. Dennard. Field-effect transistor memory, 1968. U.S. Patent 3,387,286.

P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11 (5):323–333, 1968.

P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

K. W. Deschler and E. A. Rundensteiner. B+Retake: Sustaining high volume inserts into large data pages. In J. Hammer, editor, *Proc. Int. Workshop on Data Warehousing and OLAP*, pages 56–63, Atlanta, Georgia, United States, November 2001.

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

C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, United States, May 1999.

J. Dongarra, K. London, S. Moore, S. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. In *Proc. Conf. on Linux Clusters: The HPC Revolution*, Urbana, Illinois, United States, June 2001. URL http://icl.cs.utk.edu/papi/.

J. Dundas and T. Mudge. Improving data cache performance by pre-executing instruction under a cache miss. In *Proc. ACM/IEEE Conf. on Supercomputing*, pages 176–186, Vienna, Austria, July 1997.

F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proc. ACM Symp. on Theory of Computing*, pages 483–491, Hersonissos, Greece, July 2001.

J. Esakov and T. Weiss. *Data structures: an advanced approach using C*. Prentice-Hall, Upper Saddle River, New Jersey, United States, first edition, 1989. ISBN 0131988476.

X. Fan, Y. Yang, and L. Zhang. Implementation and evaluation of String B-tree. Technical report, University of Florida, 2001.

M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *IEEE Symp. on the Foundations of Computer Science*, page 174, Palo Alto, California, United States, November 1998.

P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In E. Tardos, editor, *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 373–382, Atlanta, Georgia, United States, January 1996.

P. Ferragina and R. Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

P. Ferragina and F. Luccio. Dynamic dictionary matching in external memory. *Information and Computation*, 146(2):85–99, 1998.

P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

P. Flajolet and C. Puech. Partial match retrieval of multimedia data. *Journal of the ACM*, 33(2): 371–407, 1986.

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

C. C. Foster. Information retrieval: Information storage and retrieval using AVL trees. In L. Winner, editor, *Proc. National Conf.*, pages 192–205, Cleveland, Ohio, United States, August 1965.

E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symp. on the Foundations of Computer Science*, page 285, New York City, New York, United States, October 1999.

S. L. Fritchie. A study of Erlang ETS table implementations and performance. In *Proc. of ACM SIGPLAN Workshop on Erlang*, pages 43–55, Uppsala, Sweden, August 2003.

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.

H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice-Hall, New Jersey, United States, first edition, 2001. ISBN 0130319953.

A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on modern and emerging processors. *Int. Journal on Very Large Databases*, 16(1):77–96, 2006.

G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C*. Addison-Wesley, Boston, Massachusetts, United States, second edition, 1991. ISBN 0201416077.

G. H. Gonnet and P. Larson. External hashing with limited internal storage. *Journal of the ACM*, 35(1):161–184, 1988.

G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL server. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proc. Int. Conf. on Very Large Databases*, pages 86–97, New York City, New York, United States, August 1998.

E. D. Granston and H. A. G. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In Y. Muraoka, editor, *Proc. Int. Conf. on Supercomputing*, pages 11–20, Tokyo, Japan, July 1993.

J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, California, United States, first edition, 1992. ISBN 1558601902.

B. Green. Computer languages for symbol manipulation. *IRE Transactions*, 10(4):729–734, 1961.

D. Grinberg, S. Rajagopalan, R. Venkatesan, and V. K. Wei. Splay trees for data compression. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 522–530, San Francisco, California, United States, January 1995.

R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. ACM Symp. on Theory of Computing*, pages 397–406, Portland, Oregon, United States, May 2000.

L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symp. on the Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, United States, October 1978.

C. Halatsis and G. Philokyprou. Pseudochaining in hash tables. *Communications of the ACM*, 21 (7):554–557, 1978.

J. Hallberg, T. Palm, and M. Brorsson. Cache-conscious allocation of pointer-based data structures revisited with HW/SW prefetching. In B. Black and M. Lipasti, editors, *Second Annual Workshop on Duplicating, Deconstructing, and Debunking*, San Diego, California, United States, June 2003.

J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, United States, May 2000.

J. Handy. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, California, United States, second edition, 1998. ISBN 0123229804.

W. J. Hansen. A cost model for the internal organization of B^+-tree nodes. *ACM Transactions on Programming Languages and Systems*, 3(4):508–532, 1981.

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

G. L. Heileman and W. Luo. How caching affects hashing. In C. C. C. Demetrescu and R. Tamassia, editors, *Proc. ALENEX Workshop on Algorithm Engineering and Experiments*, pages 141–154, Vancouver, British Columbia, Canada, January 2005.

S. Heinz and J. Zobel. Practical data structures for managing small sets of strings. In M. J. Oudshoorn, editor, *Proc. of the Australasian Computer Science Conf.*, pages 75–84, Melbourne, Victoria, Australia, January 2002.

S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, California, United States, third edition, 2003. ISBN 1558605967.

Hewlett-Packard. Programming with Judy: C language Judy version 4.0. Technical report, HP Part Number: B6841-90001, 2001. URL http://docs.hp.com/en/B6841-90001/index.html.

M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1):1–13, 2001.

I. Horton. *Beginning C: From Novice to Professional*. Apress, Berkely, California, United States, fourth edition, 2006. ISBN 978-1590597354.

C. J. Hughes and S. V. Adve. Memory-side prefetching for linked data structures for processor-in-memory systems. *Journal of Parallel and Distributed Computing*, 65(4):448–463, 2005.

L. C. K. Hui and C. Martel. On efficient unsuccessful search. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 217–227, Orlando, Florida, United States, January 1992.

Intel. Intel 64 and IA-32 architectures software developer's manual. Volume 1: Basic architecture. Technical report, Intel Developer's Manual, 2007. http://www.intel.com/products/processor/manuals/index.htm.

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

J. Jannink. Implementing deletion in B$^+$-trees. *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 24(1):33–38, 1995.

T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 235–246, Philadelphia, Pennsylvania, United States, March 1989.

T. Johnson and D. Shasha. B-trees with inserts and deletes: why free-at-empty is better than merge-at-half. *Journal of Computer System Sciences*, 47(1):45–76, 1993.

W. D. Jonge and A. S. Tanenbaum. Two access methods using compact binary trees. *IEEE Transactions on Software Engineering*, 13(7):799–810, 1987.

D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proc. Int. Symp. on Computer Architecture*, pages 252–263, Denver, Colorado, United States, June 1997.

J. Kärkkäinen and S. S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 149–170, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proc. Symp. on High-Performance Computer Architecture*, pages 206–217, Toulouse, France, January 2000.

K. Kato. Persistently cached B-trees. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):706–720, 2003.

K. L. Kelley and M. Rusinkiewicz. Multikey extensible hashing for relational databases. *IEEE Software*, 05(4):77–85, 1988.

D. R. Kerns and S. J. Eggers. Balanced scheduling: instruction scheduling when memory latency is uncertain. In R. Cartwright, editor, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 278–289, Albuquerque, New Mexico, United States, June 1993.

T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3): 490–505, 2000.

C. Knessl and W. Szpankowski. A note on the asymptotic behavior of the height in B-tries for B large. *Electronic Journal of Combinatorics*, 7(R39), 2000.

C. Knessl and W. Szpankowski. Limit laws for the height in Patricia tries. *Journal of Algorithms*, 44(1):63–97, 2002.

D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Longman, Redwood City, California, United States, second edition, 1998. ISBN 0201896850.

P. Ko and S. Aluru. Obtaining provably good performance from suffix trees in secondary storage. In M. Lewenstein and G. Valiente, editors, *Proc. Symp. on Combinatorial Pattern Matching*, pages 72–83, Barcelona, Spain, July 2006.

M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 213–232, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 193–212, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29 (13):1149–1171, 1999.

R. E. Ladner, R. Fortna, and B. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In R. Fleischer, B. Moret, and E. M. Schmidt, editors, *Experimental algorithmics: from algorithm design to robust and efficient software*, pages 78–92, New York City, New York, United States, 2002.

A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1(4), 1996.

P. Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–587, 1982.

P. Larson. Linear hashing with separators—a dynamic hashing scheme achieving one-access. *ACM Transactions on Database Systems*, 13(3):366–388, 1988.

C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 129–142, Chicago, Illinois, United States, June 2005.

A. R. Lebeck. Cache conscious programming in undergraduate computer science. In *Proc. SIGCSE Technical Symp. on Computer Science Education*, pages 247–251, New Orleans, Louisiana, United States, March 1999.

S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, University of Washington, Department of Computer Science and Engineering, 1995.

M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spiad: Software prefetching in pointer and call-intensive environments. In *Proc. Annual ACM/IEEE MICRO Int. Symp. on Microarchitecture*, pages 252–263, Ann Arbor, Michigan, United States, November 1995.

D. B. Lomet. Partial expansions for file organizations with an index. *ACM Transactions on Database Systems*, 12(1):65–84, 1987.

D. Loshin. *Efficient Memory Programming*. McGraw-Hill Professional, New York City, New York, United States, first edition, 1998. ISBN 0070388687.

C. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multi-threading processors. In P. Stenström, editor, *Proc. Int. Symp. on Computer Architecture*, pages 40–51, Göteborg, Sweden, June 2001.

C. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Massachusetts, United States, October 1996.

A. W. Madison and A. P. Batson. Characteristics of program localities. *Communications of the ACM*, 19(5):285–294, 1976.

H. M. Mahmoud. *Evolution of Random Search Trees*. John Wiley and Sons, New York, United States, first edition, 1992. ISBN 0471532282.

D. Makawita, K. Tan, and H. Liu. Sampling from databases using B^+-trees. In *Proc. CIKM Int. Conf. on Information and Knowledge Management*, pages 158–164, McLean, Virginia, United States, November 2000.

U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. ACM SIAM Symp. on Discrete Algorithms*, pages 319–327, San Francisco, California, United States, January 1990.

N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In A. Elmaghraby and R. A. M. Ammar, editors, *Proc. Int. Conf. on Parallel and Distributed Computing Systems*, September 1995.

C. Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, 1991.

C. Martinez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

J. McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.

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

U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures*, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

A. Moffat, G. Eddy, and O. Petersson. Splaysort: fast, versatile, practical. *Software—Practice and Experience*, 26(7):781–797, 1996.

B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *Monographs in Discrete Mathematic and Theoretical Computer Science*, 15(1): 99–117, 1994.

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

T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, 1995.

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, California, United States, first edition, 1997. ISBN 1558603204.

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, Dallas, Texas, United States, December 1986.

D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Redwood City, California, United States, first edition, 1995. ISBN 0201633981.

J. C. Na and K. Park. Simple implementation of String B-trees. In A. Apostolico and M. Melucci, editors, *Proc. SPIRE String Processing and Information Retrieval Symp.*, pages 214–215, Padova, Italy, October 2004.

A. Naz, M. Rezaei, K. Kavi, and P. Sweany. Improving data cache performance with integrated use of split caches, victim cache, and stream buffers. In *Proc. MEDEA Workshop of Memory Performance*, pages 41–48, Antibes Juan-les-Pins, France, September 2004.

A. Newell and F. M. Tonge. An introduction to information processing language V. *Communications of the ACM*, 3(4):205–211, 1960.

J. Nievergelt. Binary search trees and file organization. *ACM Computing Surveys*, 6(3):196–207, 1974.

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

S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In K. Mehlhorn, editor, *Proc. of Workshop on Algorithm Engineering*, pages 25–36, Saarbrucken, Germany, 1998.

K. Oksanen. Memory reference locality in binary search trees. Master's thesis, Helsinki University of Technology, Department of Computer Science, 1995.

B. C. Ooi and K. Tan. B-trees: Bearing fruits of all kinds. In X. Zhou, editor, *Proc. Australasian Database Conf.*, pages 13–20, Melbourne, Victoria, Australia, January 2002.

Oracle. Berkeley DB, Oracle Embedded Database, 2007. URL http://www.oracle.com/technology/software/products/berkeley-db/index.html. Version 4.5.20.

R. Pagh. Basic external memory data structures. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 14–35, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

P. R. Panda, L. Semeria, and G. Micheli. Cache-efficient memory layout of aggregate data structures. In *Proc. Int. Symp. on System Synthesis*, pages 101–106, Montreal, Quebec, Canada, September 2001.

D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kazyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, California, United States, third edition, 2005. ISBN 1558606041.

W. W. Peterson. Open addressing. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

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

N. Rahman. Algorithms for hardware caches and TLB. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 171–192, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In G. S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. Int. Workshop on Algorithm Engineering*, pages 67–78, Aarhus, Denmark, August 2001.

B. Rais, P. Jacquet, and W. Szpankowski. Limiting distribution for the depth in Patricia tries. *SIAM Journal of Discrete Mathematics*, 6(2):197–213, 1993.

M. V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In R. W. Topor and K. Tanaka, editors, *Proc. Symp. on Databases Systems for Advanced Applications*, volume 6, pages 215–224, Melbourne, Australia, April 1997.

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

J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proc. Int. Conf. on Very Large Databases*, pages 78–89, Edinburgh, Scotland, United Kingdom, September 1999.

J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, pages 475–486, Dallas, Texas, United States, May 2000.

B. Raphael. The structure of programming languages. *Communications of the ACM*, 9(2):67–71, 1966.

A. Rathi, H. Lu, and G. E. Hedrick. Performance comparison of extendible hashing and linear hashing techniques. *ACM SIGSMALL/PC Notes*, 17(2):19–26, 1991.

G. Rivera and C. Tseng. Data transformation for eliminating conflict misses. In A. M. Berman, editor, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 38–49, Montreal, Quebec, Canada, June 1998.

T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. Int. Symp. on Computer Architecture*, pages 176–187, S. Margherita Ligure, Italy, June 1995.

K. R. Rose. Asynchronous generic key/value database. Master's thesis, Massachusetts Institute of Technology, 2000.

A. L. Rosenberg and L. Snyder. Time and space optimality in B-trees. *ACM Transactions on Database Systems*, 6(1):174–193, 1981.

A. L. Rosenberg and L. J. Stockmeyer. Hashing schemes for extendible arrays. *Journal of the ACM*, 24(2):199–221, 1977.

A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proc. Int. Symp. on Computer Architecture*, pages 111–121, Atlanta, Georgia, United States, May 1999.

A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, California, United States, October 1998.

S. Rubin, D. Bernstein, and M. Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In S. Jähnichen, editor, *Proc. Int. Conf. on Compiler Construction*, pages 259–273, Amsterdam, The Netherlands, March 1999.

P. Sanders. Memory hierarchies - models and lower bounds. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 1–13, Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, March 2003.

D. V. Sarwate. A note on universal classes of hash functions. *Information Processing Letters*, 10 (1):41–45, 1980.

R. Sedgewick. *Algorithms in C, Parts 1-4: Fundamentals, Data structures, Sorting, and Searching.* Addison-Wesley, Boston, Massachusetts, United States, third edition, 1998. ISBN 0201314525.

R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

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

T. Shanley. *The Unabridged Pentium 4: IA32 Processor Genealogy.* Addison-Wesley, Boston, Massachusetts, United States, first edition, 2004. ISBN 032124656X.

M. Sherk. Self-adjusting k-ary search trees. In F. K. H. A. Dehne, J. Sack, and N. Santoro, editors, *Proc. of Workshop on Algorithms and Data Structures*, pages 381–392, Ottawa, Canada, August 1989.

A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* John Wiley and Sons, Boston, Massachusetts, United States, seventh edition, 2004. ISBN 0471250600.

A. Silverstein. Judy IV shop manual, 2002. URL http://judy.sourceforge.net/.

R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithmics*, 9(1.5), 2005.

R. Sinha, D. Ring, and J. Zobel. Cache-efficient string sorting using copying. *ACM Journal of Experimental Algorithmics*, 11(1.2), 2006.

D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3): 652–686, 1985.

A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

A. Stoutchinin, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In R. Wilhelm, editor, *Proc. Int. Conf. on Compiler Construction*, pages 289–303, Genova, Italy, April 2001.

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

W. Szpankowski. On the height of digital trees and related problems. *Algorithmica*, 6(2):256–277, 1991.

W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. John Wiley and Sons, New York City, New York, United States, first edition, 2001. ISBN 047124063X.

Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel. Practical methods for constructing suffix trees. *Int. Journal on Very Large Databases*, 14(3):281–299, 2005.

K. Torp, L. Mark, and C. S. Jensen. Efficient differential timeslice computation. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):599–611, 1998.

D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 322–329, Paris, France, October 1998.

S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2): 174–199, 2000.

J. S. Vitter. Analysis of the search performance of coalesced hashing. *Journal of the ACM*, 30(2): 231–258, 1983.

J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proc. Int. Symp. on Computer Architecture*, pages 388–398, San Diego, California, United States, June 2003.

H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software—Practice and Experience*, 31(10):925–939, 2001.

I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, California, United States, first edition, 1999. ISBN 1558605703.

W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

C. Yang, A. R. Lebeck, H. Tseng, and C. Lee. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Transactions on Architecture and Code Optimization*, 1(4):445–475, 2004.

A. C. Yao. On random 2-3 trees. *Acta Informatica*, 9(2):159–170, 1978.

Q. Zhao, R. Rabbah, and W. Wong. Dynamic memory optimization using pool allocation and prefetching. *ACM SIGARCH Computer Architecture News*, 33(5):27–32, 2005.

J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38:1–56, 2006.

J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.