The VLDB Journal (2009) 18:157–179
DOI 10.1007/s00778-008-0094-1

REGULAR PAPER

B-tries for disk-based string management

Nikolas Askitis · Justin Zobel

Received: 11 June 2006 / Revised: 14 December 2007 / Accepted: 9 January 2008 / Published online: 11 March 2008

Abstract A wide range of applications require that large quantities of data be maintained in sort order on disk. The B-tree, and its variants, are an efficient general-purpose disk-based data structure that is almost universally used for this task. The B-trie has the potential to be a competitive alternative for the storage of data where strings are used as keys, but has not previously been thoroughly described or tested. We propose new algorithms for the insertion, deletion, and equality search of variable-length strings in a disk-resident B-trie, as well as novel splitting strategies which are a critical element of a practical implementation. We experimentally compare the B-trie against variants of B-tree on several large sets of strings with a range of characteristics. Our results demonstrate that, although the B-trie uses more memory, it is faster, more scalable, and requires less disk space.

Keywords B-tree · Burst trie · Secondary storage · Vocabulary accumulation · Word-level indexing · Data structures

1 Introduction

Efficient storage and retrieval of data is one of the fundamental problems in computer science. Many applications such as databases and search engines, are built on infrastructures that require efficient access to large volumes of data. However, the choice of data structure is limited, as the majority of trees and tries that are efficient in memory cannot be directly mapped to disk without incurring high costs [52].

The best-known structure for this task is the B-tree, proposed by Bayer and McCreight [9], 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 is considered to be the most efficient data structure for maintaining sorted data on disk [3,43,83]. A key characteristic is the use of a balanced tree structure, which guarantees worst-case $O(\log_B N)$ performance for search of N keys with a branching factor (node fan-out) of B, regardless of the distribution of data. This access bound is often significantly better than the performance of an in-memory data structure using virtual memory [3]. In practice, due to its high branching factor, typically the total volume of internal nodes is very small and traversal requires only a single disk access. External hash tables [21,44,58,67] are also efficient data structures, but cannot guarantee a bounded worst-case cost, nor can they maintain strings in sort order, which would be required for efficient range search.

In addition to its widespread use in standard database systems [23], the B-tree has many applications, including databases, information retrieval, and genomic databases [76]. B-trees have been used to efficiently manage and retrieve large vocabularies that are associated with text databases [12]. Some file systems, such as Linux Reiser and Windows NTFS, are also based on B-trees.

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 [89], it has not previously been formally described, explored, or tested. In particular, there are no discussions on node splitting strategies,

N. Askitis (✉)
School of Computer Science and Information Technology,
RMIT University, Melbourne, Australia
e-mail: naskitis@cs.rmit.edu.au; nikolas.askitis@rmit.edu.au

J. Zobel
NICTA, University of Melbourne, Parkville, Australia
e-mail: jz@csse.unimelb.edu.au

which are a critical element of a practical implementation. The B-trie proposed by Szpankowski [89] is simply a static trie indexing a set of buckets that store up to b keys.

In this paper, 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 [51] to disk, and is therefore composed of two types of nodes: *trie* and *bucket*. In a burst trie, when a bucket is deemed as being full, it is *burst* into at most A new buckets that can be randomly accessed during the bursting phase; A represents the size of the alphabet used, which in our case, is the 128 characters of the ASCII table. Bursting is efficient in memory because buckets can be of variable-size and, as shown by Heinz et al. [51], their random access during the bursting phase has little to no impact on performance. On disk, however, a bucket must be kept to a size that is a multiple of the disk-block size used by the underlying architecture (for efficiency purposes). However this implies that bursting a bucket on disk will create up to A new disk-block sized buckets, which is a waste of space. A further issue is that these buckets can be accessed at random during the bursting phase, which can attract unacceptable costs due to excessive disk accesses.

A major contribution is therefore the development of an appropriate approach to bucket splitting, which allows the B-trie to reside efficiently on disk, by both minimizing the number of buckets created and the random access caused during the splitting process. In this approach, we classify buckets as either *hybrid* or *pure*, which differ in the manner of how they are split. These novel elements of pure and hybrid buckets are how the B-trie can reside efficiently on disk, while giving a B-tree-like organization of data. 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, which in our case, involves strings with an average length of up to 30 characters.

Existing disk-resident trie structures, such as the external suffix tree [65,90], 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 [47,90]. The B-trie, in contrast, maintains a *word-level* index, and is thus not as powerful as a *full-text* index. Hence, like the B-tree, the B-trie is ideally suited for common string processing tasks that typically involve basic string processing operations such as insertion, deletion, and equality match on individual strings. Range search (finding all strings that begin with a sequence of characters) is also a common string processing operation and can be readily applied to trie-based data structures, including the B-trie. However, we omit range search in this paper, as it is beyond the scope of our work.

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 conduct thorough experiments to compare our B-trie against efficient variants of B-trees. These variants include a prefix B^+-tree [10], where internal nodes only store the shortest distinct prefix of strings that are promoted from leaf nodes, and the Berkeley B^+-tree [77], which is a high-performance open-source B^+-tree implementation. Other variants we consider are the string B-tree [36], which can maintain unbounded-length strings efficiently on disk, and a cache-oblivious string B-tree [17,20], 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 [64].

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 creates a larger index structure than the B^+-trees, and thus requires more space to buffer its trie nodes in memory, but the amount of space 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 buckets that compensated for the space consumed by trie nodes. Overall, our results show that the B-trie is a superior data structure to the B-tree, for the task of efficient disk-based string management.

2 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 [9] to solve the problem of external data management. The B-tree employs a similar balancing scheme to that of AVL trees [40], 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 [43,78,83,91], 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 [94]. 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 [70]. 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 [76]. 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 [4] 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. [5] 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 [31] 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 [2]. 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 [29] 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 access time.

A further advance in space efficiency is the method of partial expansion [68], 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 [8]. Another similar method is the adaptive overflow technique proposed by Baeza-Yates [7]. 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 [29]. 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 [10] 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 [10] 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”, “cat”, “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 [10] 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 [93]. However, the space saved is at the expense of access time, as internal nodes must be decompressed on access [82].

The string B-tree (SB-tree) is another variant proposed by Ferragina and Grossi [36]. 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 buckets 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 [3]. 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 [36] 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 can be split unevenly (that is, the contents of a node can not always be divided evenly amongst two new nodes), 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 [36] suggest sorting the entire dataset on disk, and to build the SB-tree from the bottom-up, that is, to bulk-load [3,36,56]. 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 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 [36,56,74]. 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 [81] experimentally compared the performance of the SB-tree against the Berkeley B$^+$-tree [77], 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 buckets and its subsequent increase in height. The SB-tree, in contrast, required no overflow buckets 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 [81] 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 buckets and the high fan-out compensated. Hence, the SB-tree is an efficient data structure for long strings, but has relatively poor performance otherwise [81], as we show in later experiments.

Ferragina and Grossi [35] compared the performance of the SB-tree to a suffix array [71], 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 [30,34,37].

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 [80]. 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 [57] 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 [42,64]. Brodal and Fagerberg [20], for example, theoretically investigated a static cache-oblivious string dictionary. Similarly, a dynamic cache-oblivious B-tree [14] has been described, but with no analysis of actual performance. The cache-oblivious dynamic dictionary [16] 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 [15].

Recently, Bender et al. [17] 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 [14], using uniformly distributed integers. Indeed, a dynamic cache-oblivious string B-tree has yet to be implemented [17].

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 [85] 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 [72] 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 [52].

Ciriani et al. [25,26] 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, in practice, this would be inefficient—particularly with strings—due to cost of updating its multi-layer index [79,92].

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. In practice, however, this approach implies that 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), which is expensive. 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.

Ko et al. [63] proposed a self-adjusting layout scheme for suffix trees on disk that can theoretically optimize the number of disk accesses required for a sequence of queries. However, the authors also argued that the cost of adjusting their layout scheme will likely be more expensive than using a balanced tree. As such, their layout, which begins as a balanced tree, is designed to make tree adjustments infrequent.

2.1 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 [81]. Furthermore, to the best of our knowledge, there is currently no implementation support for a dynamic SB-tree [33,35,36,56,74,81] 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 [88], but requires strings to be sorted in advance and, once built, it cannot 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 [29]. All nodes are of fixed size, which in our case is 8,192 bytes (Fig. 1). This is a typical disk-block size and has been shown to provide good performance [32,45]. In contrast to the SB-tree, strings are stored within nodes. To prevent the situation where a single string consumes an entire node, we enforce a

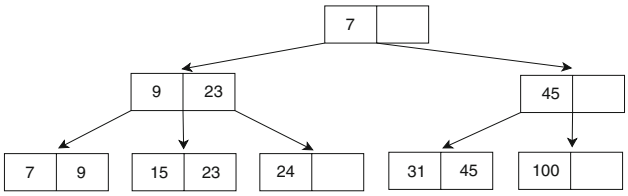

Fig. 1 A conventional B$^+$-tree. In this example, integers are used as keys for simplicity

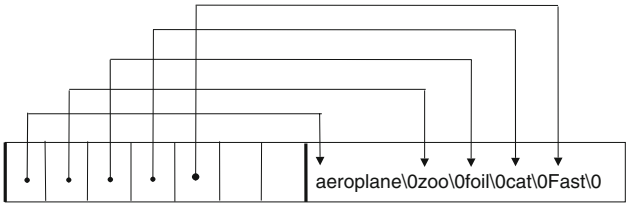

Fig. 2 Null-terminated strings are appended to existing strings in a node. Pointers to these strings are kept in sort order. This approach permits rapid insertion and search

string limit of 1,000 characters. (A production implementation would have to cater for longer strings by using overflow buckets, 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 [49] 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 [49]. 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 [13]. Figure 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. 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 [83]. A top-down technique has been proposed by Guibas and Sedgewick [48], which performs the splitting during tree traversal. This approach results in a slight decrease in space efficiency, as full nodes can be unnecessarily split. The main components of the B$^+$-tree are as follows:

Internal nodes: These are 8,192 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 [3].

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 [9,29,53]. 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* [53]. 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 [43,46]. Johnson and Shasha [54,55] 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 [53] 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.

3 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 buckets 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 [87], the k-splay tree [85], the k-forest [72] and the external skip list [26]. These data structures achieve cost-adaption through structural modifications that are often too expensive to apply in practice.

These benefits have made the trie popular for applications such as text compression [11], dictionary management [1], and pattern matching [39]. However, although fast, trie structures are space-hungry [28,51,73]. A simple implementation of an trie is to represent every node as an array of pointers, one for each letter of the alphabet [41]. This forms an array trie, where each leaf is the terminus of a chain of pointers representing a string, with k nodes in a string of length k. The array trie offers rapid access to strings, but is space-intensive. Several variations have been proposed to address this space problem. A compact trie [11] for example, omits chains of trie nodes that descend to a single node. The Patricia trie [83] uses a similar approach, but collapses all redundant chains, not just those that point to leaf nodes. These variants save space but at the expense of access time and maintaining a more complex data structure.

Another approach is to reduce the size of each trie by removing unused pointers. The de la Briandais trie [19] or list trie [61] saves space by changing the representation of trie nodes from arrays to linked lists that only maintain non-null pointers. These savings in space are at the expense of access time [84]. Bentley and Sedgewick [18] changed the representation of trie nodes from a linked list to a binary search tree, forming a structure known as a ternary search trie. Every node in a TST stores three outgoing pointers, and so, is not as compact as a list trie, but can be substantially faster to access while conserving more space than an array trie.

A highly effective solution to the space problem is the burst trie [51]. The burst trie stores strings within bounded-sized buckets. Once a bucket is full, it is *burst*, which involves creating a new parent trie node that is represented as an array of pointers, one for each letter of the alphabet A. The strings within the original bucket are then distributed into at most A new buckets, in accordance to the lead character, which is then removed. By storing strings within buckets 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 buckets. Bursting a bucket on disk implies creating up to A new fixed-sized buckets 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 (Patricia trie) data structures that can reside on disk [27]. However, these data structures maintain *full-text* indexes that store every distinct suffix in a text collection [47,56]. 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 [65]. Their high space consumption is a major restriction on their application to common string processing tasks, such as vocabulary accumulation and text indexing [38,47,75]. Grossi and Vitter [47] 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 [75,95,96], 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 [22,62,90]. 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. [24] 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, the Patricia trie (and its variants) is an expensive structure to access and maintain when storing a large set of strings [51]. As claimed by Heinz et al. [51], Patricia tries 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 to 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.

4 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 [51] to maintain a space-efficient trie, by storing strings within buckets that are structurally similar to those described for the B$^+$-tree: fixed-sized disk blocks represented as arrays. Once a bucket becomes full, a splitting strategy is required that, in contrast to bursting, throttles the number of buckets and trie nodes created. The concept of a *B-trie* has been suggested by Szpankowski [89] and is briefly discussed elsewhere [59,60,69]. 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 buckets undergo a new splitting procedure called a *B-trie split*. In this approach, the set of strings in each bucket is divided on the basis of the first character that follows the trie path that leads to the bucket. (Each node in the path consumes one character.) When all the strings in a bucket have the same first character, this character can be removed from each string; subsequent splitting of this bucket will force the creation of a new parent trie. We label these buckets as *pure*. When the set of strings in a bucket have distinct first characters, several paths in the parent trie lead to it. We label these buckets as *hybrid*; subsequent splitting of this bucket will not create a new parent trie.

When a bucket 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 bucket, while others are moved into the new bucket. 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 bucket, forming a *hybrid* bucket. Figures 3, 4, and 5 illustrate examples of this splitting procedure, which we explain in more detail later.

In either case (hybrid or pure), each bucket 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 can be, in comparison to the traversal of internal B-tree nodes, significantly lower; identification of a bucket 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 a bucket 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 bucket cannot guarantee that the two new buckets are equally loaded. In most cases, the load is likely to be approximately equal (as we observed in our experiments described

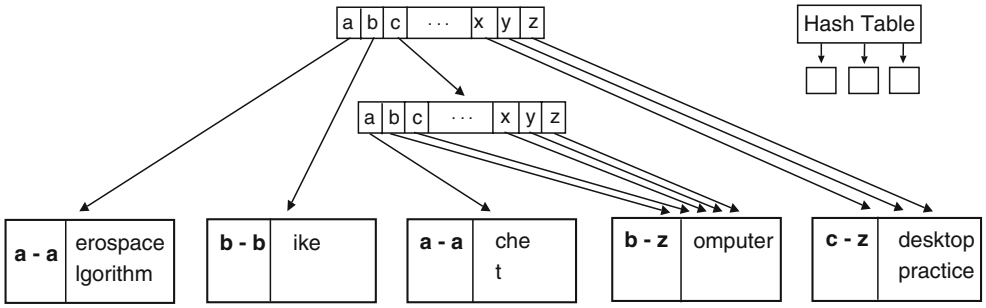

Fig. 3 The words "cat","algorithm","computer", "practice", "cache", "bike", "desktop" and "aerospace" were inserted into the B-trie, creating three pure buckets (first three from the left) along with two hybrids. The hash table stores strings that are consumed. "c" for example, would be consumed by the root trie and "a", would be consumed by the first pure bucket

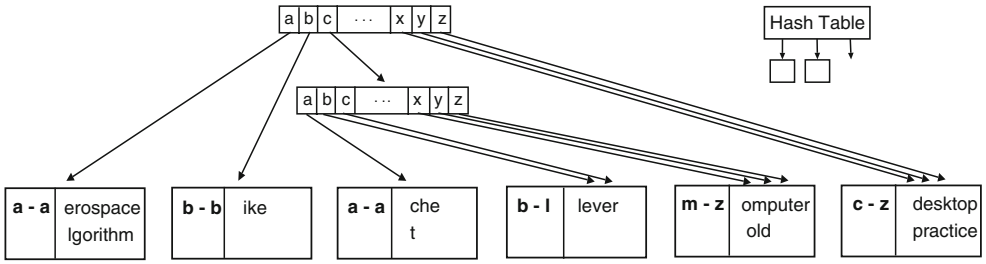

Fig. 4 The strings "cold" and "clever" were inserted into the B-trie in Fig. 3. The second hybrid bucket (from the right of Fig. 3) split, creating two new hybrids

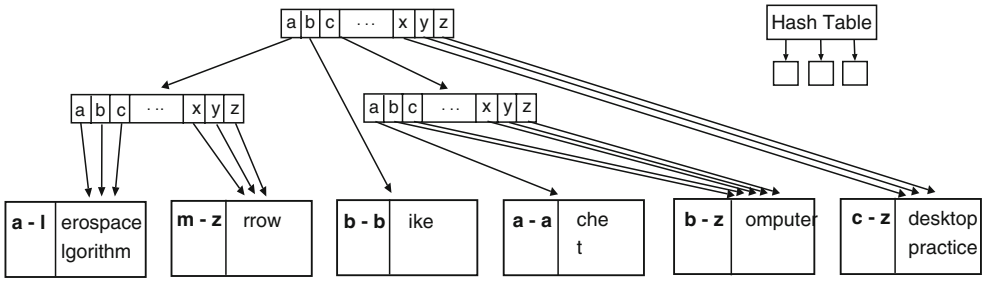

Fig. 5 The word 'arrow" was inserted into Fig. 3. The left-most pure bucket split into two hybrids and a new parent trie

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 buckets will contain one string only, while the other contains the rest. However, our B-trie splitting algorithm ensures that there are no empty buckets, and, as we demonstrate later, an occasional uneven split has little to no impact on performance.

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 buckets.

We now describe algorithms for maintaining a B-trie. The main components of our B-trie are as follows:

Buckets: Structurally—apart from the added character-range field—buckets are identical to the leaf nodes used by our implementation of a B^+-tree. However, the lead character of each string in the bucket 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 bucket is acquired. Recall that a pure bucket 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 bucket or a trie.

As discussed by Heinz et al. [51], the number of trie nodes—and hence the space they require—can be kept small due to the use of buckets. 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 bucket 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 bucket. When this occurs, an auxiliary data structure is used to store such short strings. We use our cache-conscious hash table [6] 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 bucket, as described for the burst trie [51]. 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 buckets 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 bucket 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 bucket. 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 $c_1 \cdots c_m$. There are two types of buckets: *hybrid* and *pure*. Pure buckets are those that have a range of a single character. Hybrid buckets are those that have a range comprised of two or more distinct characters. All strings in a bucket share some prefix h, and thus the prefix need not be stored. That is, a pure bucket 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 bucket 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 bucket.
2. There is only a single route from the root to any trie node.
3. For a route R leading to a pure bucket $B^P(h)$, the sequence $s(R) = h$.
4. For a route R leading to a hybrid bucket $B^H(h, l, u)$, the sequence $s(R) = h \cdot c$ where $c \in [l, u]$.
5. In a hybrid bucket, $l \neq u$.
6. For a hybrid bucket $B^H(h, l, u)$ where $h = c_1 \cdots c_{m-1}$, there is a set $\mathscr{R}$ of routes

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

   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 bucket with a parent trie. When a hybrid bucket splits, it creates one new sibling bucket (the original bucket is re-used). This action grows the B-trie horizontally. An example is shown in Fig. 4. Eventually, splitting a hybrid will lead to the creation of a pure bucket. As strings are distributed into the pure bucket, 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 bucket) and stored in the hash table. When a pure bucket is split, a new trie node is created and assigned as its parent, after which the pure bucket is transformed into a hybrid and the split proceeds as a hybrid. When a pure bucket splits, the B-trie is grown both vertically and horizontally. An example is shown in Fig. 5. We adhered to the following design principles to allow for a fairer comparison to the B$^+$-tree:

1. Buckets are structured and managed in much the same manner as the leaf nodes used by our implementation of a B$^+$-tree, as discussed in Sect. 2.1. That is, buckets are initialized with 128 string pointers and are given an additional kilobyte of free space when read into memory. No string duplicates are maintained. Instead, strings that are stored in buckets or the hash table are proceeded by a 4-byte accumulator.
2. A pointer refers to a trie 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 node or bucket is immediately synchronized (written) to disk, to ensure data integrity.

4.1 B-trie initialization

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

4.2 To search for a string

Equality search takes a query string Q as input and may return a pointer to a bucket 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 bucket $B^P(h)$ such that $t \in B^P(h)$ and $Q = h \cdot t$, or to find a hybrid bucket $B^H(h \cdot c, l, u)$ 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 bucket, 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 bucket B is acquired, a binary search for Q' concludes the search.

4.3 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 bucket B. That is, insertion of a string Q is the task of finding a pure bucket $B^P(h)$ such that $Q = h \cdot t$, and adding t to $B^P(h)$, or finding a hybrid bucket $B^H(h \cdot c, l, u)$ such that $Q = h \cdot c \cdot t$, and adding $c \cdot t$ to $B^H(h \cdot c, l, u)$. If the bucket 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 bucket is created to store Q'. The new bucket 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 bucket is hybrid or pure. In the latter case, care must be taken to discard the bucket 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.

4.4 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 bucket, 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 bucket is acquired which is binary searched for the string suffix Q'. If the suffix is found, it is removed and the bucket is internally re-organized to avoid space wastage due to internal fragmentation. The computational cost of re-organizing the bucket is small and bound by the size of the bucket. Once the bucket 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 [53]. That is, once a bucket 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 bucket and will not become empty on split. (A pure bucket 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 bucket. 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 bucket cannot be merged and must be deleted. One way to delete the bucket is to apply lazy deletion as described, and simply flag the bucket as having been deleted. However, to conserve space, the bucket should be deleted from disk, but which can involve shifting a potentially large number of buckets (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 out number insertions, or when space is highly restrictive.

4.5 Splitting a bucket

Splitting takes place when the insertion algorithm deems a bucket as full, due to insufficient space. Splitting a hybrid bucket $B^H(h, l, u)$ produces two buckets, both of which can be either pure or hybrid. This action grows the B-trie

horizontally. An example is shown in Fig. 4. The basis of the split is the first character of the strings in the bucket. 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 bucket or the other will be under-loaded but no empty buckets are maintained.

That is, a split-point must be found that can achieve good distribution across two new buckets. To determine a split point, we count the number of occurrences of each leading character in the original bucket. 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 bucket, which is not maintained. Hence, the pointers in the parent trie P between the range of $[l, d]$ are nulled.

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

The splitting procedure for a pure bucket $B^P(h \cdot u)$ 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 bucket. The old parent becomes the grandparent. All pointers in the new parent are assigned to the pure bucket, which changes the bucket into a hybrid. The split then proceeds as described for a hybrid bucket, and so, when a pure bucket splits, the B-trie is grown both vertically and horizontally. An example is shown in Fig. 5. The splitting process terminates only when both children are not full, in which case, the new buckets and their parent trie are written to disk. Otherwise, the process continues recursively by splitting the full child bucket. The non-full child is written to disk and discarded from memory. These novel elements of pure and hybrid buckets are how the trie properties are maintained, while giving a B-tree-like organization of data on disk.

5 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 [77], 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 [10], which only stores the shortest distinct prefix of strings that are promoted from leaf nodes. We also explore front-coding [93], 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 [36] and the cache-oblivious string B-tree [17], 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) [81]. In addition, it is not well suited for tasks where the number of strings to insert is 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 [36, 56, 81]. 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 [17, 20, 66].

As test data, we used the string sets shown in Table 1 that were extracted from documents made available through TREC [50] and its GOV2 test collection. They are composed of null-terminated variable-length strings (up to a 1,000

Table 1 Characteristics of the datasets used in the experiments. Our DISTINCT dataset containing 28772169 strings was scaled down geometrically to create four distinct subsets of 9098559, 2877217, 909855, and 287721 strings

Dataset	Distinct strings	String occs	Average length	Volume of distinct (MB)	Volume total (MB)
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

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 [50]. 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 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 [92] 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. [92] 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 [32,45]. We consider the height of the B-trie or B^+-trees as the number of nodes accessed prior to reaching a bucket or leaf node, respectively.

5.1 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 bucket 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 [86], 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.

5.2 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 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, in their original order of occurrence. This process is useful in evaluating the performance of the data structures during search for known strings.

Table 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 1. The elapsed time is shown in seconds and the space in megabytes. The best measures of time and space are in bold

Dataset	Build		Search		Disk-space (MB)
	Buffered	Unbuffered	Buffered	Unbuffered	
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

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, which will subsequently reduce the space required by internal nodes. As a consequence, however, more leaf nodes will be created, which is likely to increase overall space consumption.

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 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 buckets (by stripping away common prefixes). This requires that a sufficient number of distinct strings are inserted to improve the space utilization within buckets, which, in turn, reduce the number of splits that occur. From our results in Table 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 bucket. The number of consumable strings is therefore bound by the number of trie nodes or pure buckets. Thus, the hash table cannot grow large relative to the overall size of the B-trie, as shown in Table 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. Hence, a long chain of trie nodes can be traversed rapidly, allowing frequently accessed buckets to be fetched

Table 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 1

Dataset	Trie nodes	Buckets	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
	Internal	Leaves					
Standard B$^+$-tree							
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	–	–	–	–

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.

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 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 bucket, 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 bucket 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 buckets are cre-

ated, 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 s to construct and 151 s to self-search, which is about 200 s and 193 s 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.

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 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.

5.3 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 [93]. 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 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 3,343 s and 453 MB of disk space. The equivalent uncompressed standard B$^+$-tree, in contrast, required only 429 s to build, but used over 646 MB 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.

Table 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 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

Dataset	Construction (s)	Self-search (s)	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

5.4 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 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 Fig. 6. In addition, we measured the cost of skewed construction and self-search using the TREC dataset, shown in Table 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 3,170 s) to construct and self-search than the unbuffered standard and prefix B$^+$-trees. The Berkeley B$^+$-tree, in contrast, displayed

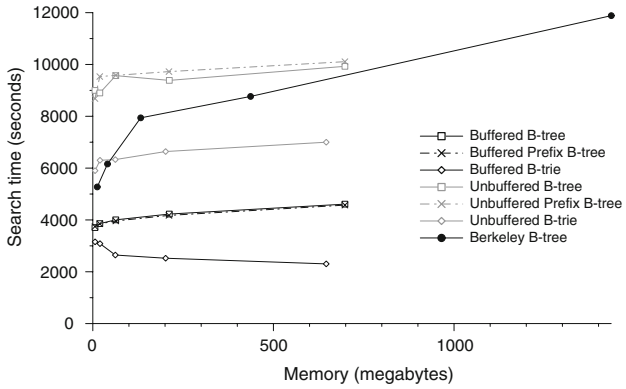

Fig. 6 A comparison of skew search performance using the TREC dataset, as the string cardinality of the data structures (the number of strings they store) increase. The DISTINCT datasets of Table 1 represent the points on the graph, with the left-most points representing our smallest DISTINCT dataset. For brevity, we label a B^+-tree as a B-tree in this figure

Table 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 1. The elapsed times are shown in seconds and the space in megabytes. The best measures of time and space in bold

Dataset	Build		Search		Disk-space (MB)
	Buffered	Unbuffered	Buffered	Unbuffered	
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	72.2	**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

competitive performance, being almost as fast as our unbuffered B-trie (Table 5), 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 1,009 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 5,500 s) due to the elimination of disk access on index traversal. Similarly, the buffered B^+-trie also improves by as much as 56% (or around 3,590 s), and remains faster to access than the buffered B^+-trees. These results 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 can improve 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 buckets. 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 bucket is avoided (assuming that the query string is consumed prior to accessing a bucket) and the search continues in the in-memory hash table. For example, during construction, the B-trie consumes 1,612 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 Fig. 6. Unlike the previous experiments, however, some searches were unsuccessful. For example, after having inserted almost 29 million distinct strings into the data structures which were then

searched using the TREC dataset, a total of 1,059,166 searches were unsuccessful.

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 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 2,306 s), 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 Fig. 6, the unbuffered B-trie was up to 67% slower (or around 4,700 s) 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 3,100 s) to access. The Berkeley B$^+$-tree showed good performance under skew for small index sizes, rivaling the performance of the unbuffered B-trie and B$^+$-trees. This behavior is consistent with the results observed previously in Table 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.

5.5 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 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 s) 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 buckets 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 bucket is acquired. This implies that on search, the B-trie may typically issue 15 system calls to disk (including one to access a bucket), which is expensive. Hence, it was not surprising to observe a performance decline of up to 35% (or around 70 s), 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 1,150 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 buckets—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.

5.6 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 5.

The buffered B-trie was the fastest data structure to construct and self-search, being up to 6% (or about 10 s) faster

than the buffered standard or prefix B$^+$-trees. It also required the 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.

5.7 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.

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 s) 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.

Table 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 1. These results show the performance of the data structures when their size exceeds the capacity of main memory (2 GB). 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

Data structure	Build (s)	Search (s)	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

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.

5.8 String B-tree

We downloaded a high quality implementation of a SB-tree from [88]. As discussed in Sect. 2, the SB-tree represents the internal and leaf nodes of a B-tree as Patricia tries, which are stored succinctly on disk [36]. 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 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

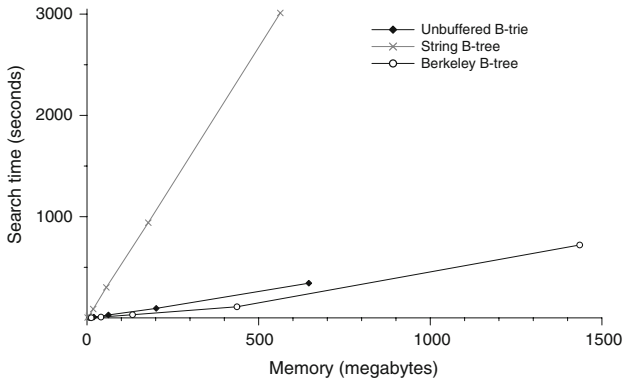

Fig. 7 A comparison of the time in seconds and the space in megabytes required to self-search the SB-tree, the Berkeley B$^+$-tree, and the unbuffered B-trie, using the DISTINCT datasets of Table 1. The left-most points, for example, represents the self-search cost using our smallest DISTINCT dataset. For brevity, we label a B$^+$-tree as a B-tree in this figure

space required to self-search the SB-tree, Berkeley B$^+$-tree, and unbuffered B-trie are presented in Fig. 7.

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. Similarly, 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 buckets. 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 2,290 s) to search than the Berkeley B$^+$-tree, and up to 89% slower (or around 2,668 s) to search than our unbuffered B-trie. These results are consistent to those reported by Rose [81], 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 that is kept small in size by the use of buckets.

5.9 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 bucket in the B$^+$-tree and B-trie, respectively, has equal probability of access.

Table 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 1. The elapsed times required to construct and self-search are shown in seconds and the space in megabytes. The best measures in bold

No. strings deleted	Data Structure	Delete (s)	Insert (s)	Total time (s)	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

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 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 buckets (no trie nodes were deleted). The B-trie is an unbalanced structure, and considering that buckets can be unevenly split, more buckets 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 buckets 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 buckets are merged (which is the case in these experiments), the B-trie would have had to physically delete up

to 537 buckets on disk, which is expensive. The standard B$^+$-tree, in contrast, having deleted no leaf nodes is more efficient in this case, as both internal and leaf nodes can be under-loaded.

6 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 paper, 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 [22,47,90].

We ran a series of experiments to compare the B-trie to the Berkeley B$^+$-tree [77] and our own high performance implementation of a standard B$^+$-tree (where internal nodes store full-length strings), and a prefix B$^+$-tree [10], using string datasets extracted from documents made available through real-world data repositories, such as TREC [50]. We considered alternative data structures such as the string B-tree [36] and the cache-oblivious string B-tree [17], 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 buckets and trie nodes. Furthermore, the removal of shared prefixes in buckets can compensate 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 sustain superior access times relative to the unbuffered B$^+$-trees, when maintaining a small average trie height, which, however, is generally sustained under skew access or with strings that have a small average length. 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, smaller (overall), and more scalable than common variants of B$^+$-tree that are currently in standard use.

Acknowledgments This work was supported by the Australian Postgraduate Award, a scholarship from the Australian Research Council and the School of Computer Science and Information Technology at RMIT University.

References

1. Aoe, J., Morimoto, K., Sato, T.: An efficient implementation of trie structures. Softw Practice Exp **22**(9), 695–721 (1992)
2. Arge, L.: The buffer tree: a new technique for optimal I/O-algorithms. In: Proc. Int. Workshop on Algorithms and Data Structures, pp. 334–345. Kingston (1995)
3. Arge, L.: External memory data structures. In: Handbook of Massive Data Sets, pp. 313–357. Kluwer, Norwell (2002)
4. Arnow, D.M., Tenenbaum, A.M.: An empirical comparison of B-trees, compact B-trees and multiway trees. In: Proc. ACM SIGMOD Int. Conf. on the Management of Data, pp. 33–46. Boston (1984)
5. Arnow, D.M., Tenenbaum, A.M., Wu, C.: P-trees: Storage efficient multiway trees. In: Proc. ACM SIGIR Int. Conf. on Research and Development in Information Retrieval, pp. 111–121. Montreal (1985)
6. Askitis, N., Zobel, J.: Cache-conscious collision resolution in string hash tables. In: Proc. SPIRE String Processing and Information Retrieval Symp., pp. 91–102. Buenos Aires (2005)
7. Baeza-Yates, R.A.: An adaptive overflow technique for B-trees. In: Proc. Int. Conf. on Extending Database Technology, pp. 16–28, Venice (1990)
8. Baeza-Yates, R.A., Larson, P.A.: Performance of B$^+$-trees with partial expansions. IEEE Trans Knowl Data Eng **1**(2), 248–257 (1989)
9. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf **1**(3), 173–189 (1972)
10. Bayer, R., Unterauer, K.: Prefix B-trees. ACM Trans Database Systems **2**(1), 11–26 (1977)
11. Bell, T.C., Cleary, J.G., Witten, I.H.: Text Compression, 1st edn. Prentice-Hall, New Jersey (1990)
12. Bell, T.C., Moffat, A., Witten, I.H., Zobel, J.: The MG retrieval system: compressing for space and speed. Commun ACM **38**(4), 41–42 (1995)
13. Ben-Asher, Y., Farchi, E., Newman, I.: Optimal search in trees. SIAM J. Comput. **28**(6), 2090–2102 (1999)

14. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. In: Proc. IEEE Foundations of Computer Science, pp. 399–409, Redondo Beach (2000)
15. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Efficient tree layout in a multilevel memory hierarchy. In: Proc. European Symp. on Algorithms, pp. 165–173, Rome (2002)
16. Bender, M.A., Duan, Z., Iacono, J., Wu, J.: A locality-preserving cache-oblivious dynamic dictionary. J. Algorithms **53**(2), 115–136 (2004)
17. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-oblivious string B-trees. In: Proc. of ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp. 233–242. Chicago (2006)
18. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 360–369. New Orleans (1997)
19. de la Briandais, R.: File searching using variable length keys. In: Proc. Western Joint Computer Conference, pp. 295–298, New York (1959)
20. Brodal, G., Fagerberg, R.: Cache-oblivious string dictionaries. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 581–590, Miami (2006)
21. Chang, Y., Lee, C., ChangLiaw, W.: Linear spiral hashing for expansible files. IEEE Trans. Knowl. Data Eng. **11**(6), 969–984 (1999)
22. Cheung, C., Yu, J.X., Lu, H.: Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans. Knowl. Data Eng. **17**, 90–105 (2005)
23. Chong, E.I., Srinivasan, J., Das, S., Freiwald, C., Yalamanchi, A., Jagannath, M., Tran, A., Krishnan, R., Jiang, R.: 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)
24. Chowdhury, N.M.M.K., Akbar, M.M., Kaykobad, M.: DiskTrie: An efficient data structure using flash memory for mobile devices. In: Workshop on Algorithms and Computation, pp. 76–87. Bangladesh Computer Council Bhaban, Agargaon (2007)
25. Ciriani, V., Ferragina, P., Luccio, F., Muthukrishnan, S.: Static optimality theorem for external memory string access. In: IEEE Symp. on the Foundations of Computer Science, pp. 219–227, Vancouver (2002)
26. Ciriani, V., Ferragina, P., Luccio, F., Muthukrishnan, S.: A data structure for a sequence of string accesses in external memory. ACM Trans. Algorithms **3**(1), 6 (2007)
27. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 383–391, Atlanta (1996)
28. Comer, D.: Heuristics for trie index minimization. ACM Trans. Database Systems **4**(3), 383–395 (1979)
29. Comer, D.: Ubiquitous B-tree. ACM Comput. Surv. **11**(2), 121–137 (1979)
30. Crauser, A., Ferragina, P.: On constructing suffix arrays in external memory. In: Proc. of European Symp. on Algorithms, pp. 224–235, Prague (1999)
31. Culik, K., Ottmann, T., Wood, D.: Dense multiway trees. ACM Trans. Database Systems **6**(3), 486–512 (1981)
32. Deschler, K.W., Rundensteiner, E.A.: B+Retake: Sustaining high volume inserts into large data pages. In: Proc. Int. Workshop on Data Warehousing and OLAP, pp. 56–63, Atlanta (2001)
33. Fan, X., Yang, Y., Zhang, L.: Implementation and evaluation of String B-tree. Tech. rep., University of Florida (2001)
34. Farach, M., Ferragina, P., Muthukrishnan, S.: Overcoming the memory bottleneck in suffix tree construction. In: IEEE Symp. on the Foundations of Computer Science, p. 174, Palo Alto (1998)
35. Ferragina, P., Grossi, R.: Fast string searching in secondary storage: theoretical developments and experimental results. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 373–382, Atlanta (1996)
36. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. J. ACM **46**(2), 236–280 (1999)
37. Ferragina, P., Luccio, F.: Dynamic dictionary matching in external memory. Inf. Comput. **146**(2), 85–99 (1998)
38. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
39. Flajolet, P., Puech, C.: Partial match retrieval of multimedia data. J. ACM **33**(2), 371–407 (1986)
40. Foster, C.C.: Information retrieval: information storage and retrieval using AVL trees. In: Proc. National Conf., pp. 192–205, Cleveland (1965)
41. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
42. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: IEEE Symp. on the Foundations of Computer Science, p. 285, New York City (1999)
43. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: the Complete Book, 1st edn. Prentice-Hall, New Jersey (2001)
44. Gonnet, G.H., Larson, P.: External hashing with limited internal storage. J. ACM **35**(1), 161–184 (1988)
45. Gray, J., Graefe, G.: The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Record **26**(4), 63–68 (1997)
46. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, 1st edn. Morgan Kaufmann, San Francisco (1992)
47. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: Proc. ACM Symp. on Theory of Computing, pp. 397–406, Portland (2000)
48. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: IEEE Symp. on the Foundations of Computer Science, pp. 8–21, Ann Arbor (1978)
49. Hansen, W.J.: A cost model for the internal organization of B^+-tree nodes. ACM Trans. Program. Languages Systems **3**(4), 508–532 (1981)
50. Harman, D.: Overview of the second text retrieval conf. (TREC-2). Inf. Process. Manage. **31**(3), 271–289 (1995)
51. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: A fast, efficient data structure for string keys. ACM Trans. Inf. Systems **20**(2), 192–223 (2002)
52. Hui, L.C.K., Martel, C.: On efficient unsuccessful search. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 217–227, Orlando (1992)
53. Jannink, J.: Implementing deletion in B^+-trees. Proc. ACM SIGMOD Int. Conf. Manag. Data **24**(1), 33–38 (1995)
54. Johnson, T., Shasha, D.: Utilization of B-trees with inserts, deletes and modifies. In: Proc. of ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp. 235–246, Philadelphia (1989)
55. Johnson, T., Shasha, D.: B-trees with inserts and deletes: why free-at-empty is better than merge-at-half. J. Comput. System Sci. **47**(1), 45–76 (1993)
56. Kärkkäinen, J., Rao, S.S.: Full-text indexes in external memory. In: Algorithms for Memory Hierarchies, pp. 149–170. Dagstuhl Research Seminar, Schloss Dagstuhl (2002)
57. Kato, K.: Persistently cached B-trees. IEEE Trans. Knowl. Data Eng. **15**(3), 706–720 (2003)
58. Kelley, K.L., Rusinkiewicz, M.: Multikey extensible hashing for relational databases. IEEE Softw. **05**(4), 77–85 (1988)
59. Knessl, C., Szpankowski, W.: A note on the asymptotic behavior of the height in B-tries for B large. Electron. J. Combinat. 7(R39) (2000)
60. Knessl, C., Szpankowski, W.: Limit laws for the height in Patricia tries. J. Algorithms **44**(1), 63–97 (2002)

61. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd edn. Addison-Wesley Longman, Redwood City (1998)
62. Ko, P., Aluru, S.: Obtaining provably good performance from suffix trees in secondary storage. In: Proc. Symp. on Combinatorial Pattern Matching, pp. 72–83, Barcelona (2006)
63. Ko, P., Aluru, S.: Optimal self-adjusting trees for dynamic string data in secondary storage. In: Proc. SPIRE String Processing and Information Retrieval Symp., pp. 184–194, Santiago (2007)
64. Kumar, P.: Cache oblivious algorithms. In: Algorithms for Memory Hierarchies, pp. 193–212. Dagstuhl Research Seminar, Schloss Dagstuhl (2003)
65. Kurtz, S.: Reducing the space requirement of suffix trees. Softw. Practice Exp. **29**(13), 1149–1171 (1999)
66. Ladner, R.E., Fortna, R., Nguyen, B.: A comparison of cache aware and cache oblivious static search trees using program instrumentation. In: Experimental Algorithmics: from Algorithm Design to Robust and Efficient Software, pp. 78–92, New York City (2002)
67. Larson, P.: Linear hashing with separators—a dynamic hashing scheme achieving one-access. ACM Trans. Database Systems **13**(3), 366–388 (1988)
68. Lomet, D.B.: Partial expansions for file organizations with an index. ACM Trans. Database Systems **12**(1), 65–84 (1987)
69. Mahmoud, H.M.: Evolution of Random Search Trees, 1st edn. J Wiley, New York (1992)
70. Makawita, D., Tan, K., Liu, H.: Sampling from databases using B^+-trees. In: Proc. CIKM Int. Conf. on Information and Knowledge Management, pp. 158–164, McLean (2000)
71. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. In: Proc. ACM SIAM Symp. on Discrete Algorithms, pp. 319–327, San Francisco (1990)
72. Martel, C.: Self-adjusting multi-way search trees. Inf. Process. Lett. **38**(3), 135–141 (1991)
73. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–271 (1976)
74. Na, J.C., Park, K.: Simple implementation of String B-trees. In: Proc. SPIRE String Processing and Information Retrieval Symp., pp. 214–215, Padova (2004)
75. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), 1–61 (2007)
76. Ooi, B.C., Tan, K.: B-trees: Bearing fruits of all kinds. In: Proc. Australasian Database Conf., pp. 13–20, Melbourne (2002)
77. Oracle: Berkeley DB, Oracle Embedded Database (2007). http://www.oracle.com/technology/software/products/berkeley-db/index.html. Version 4.5.20
78. Pagh, R.: Basic external memory data structures. In: Algorithms for Memory Hierarchies, pp. 14–35. Dagstuhl Research Seminar, Schloss Dagstuhl (2002)
79. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
80. Rao, J., Ross, K.A.: Making B^+-trees cache conscious in main memory. In: Proc. ACM SIGMOD Int. Conf. on the Management of Data, pp. 475–486, Dallas (2000)
81. Rose, K.R.: Asynchronous generic key/value database. Master's thesis, Massachusetts Institute of Technology (2000)
82. Rosenberg, A.L., Snyder, L.: Time and space optimality in B-trees. ACM Trans. Database Systems **6**(1), 174–193 (1981)
83. Sedgewick, R.: Algorithms in C, Parts 1-4: Fundamentals, Data structures, Sorting, and Searching, 3rd edn. Addison-Wesley, Boston (1998)
84. Severance, D.G.: Identifier search mechanisms: a survey and generalized model. ACM Comput. Surv. **6**(3), 175–194 (1974)
85. Sherk, M.: Self-adjusting k-ary search trees. In: Proc. of Workshop on Algorithms and Data Structures, pp. 381–392, Ottawa (1989)
86. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7th edn. Wiley, Boston (2004)
87. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985)
88. Software, T.M.: C++ string B-tree library (2007). http://wikipedia-clustering.speedblue.org/strBTree.php
89. Szpankowski, W.: Average Case Analysis of Algorithms on Sequences, 1st edn. Wiley, New York City (2001)
90. Tian, Y., Tata, S., Hankins, R.A., Patel, J.M.: Practical methods for constructing suffix trees. Int. J. Very Large Databases **14**(3), 281–299 (2005)
91. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. **33**(2), 209–271 (2001)
92. Williams, H.E., Zobel, J., Heinz, S.: Self-adjusting trees in practice for large text collections. Softw. Practice Exp. **31**(10), 925–939 (2001)
93. Witten, I.H., Bell, T.C., Moffat, A.: Managing Gigabytes: Compressing and Indexing Documents and Images, 1st edn. Morgan Kaufmann, San Francisco (1999)
94. Yao, A.C.: On random 2-3 trees. Acta Inf. **9**, 159–170 (1978)
95. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**, 1–56 (2006)
96. Zobel, J., Moffat, A., Ramamohanarao, K.: Inverted files versus signature files for text indexing. ACM Trans. Database Systems **23**(4), 453–490 (1998)