Asynchronous Generic Key/Value Database

by

Kyle R. Rose

B.A., Computer Science and Mathematics

Cornell University

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science at the

Massachusetts Institute of Technology

September 2000

©2000 Massachusetts Institute of Technology
All rights reserved

Signature of Author ........ Department of Electrical Engineering and Computer Science 
September 4, 2000

Certified by ...........

Frans Kaashoek
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .................................................. David Mazières
Ph.D. Student, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ...........

Professor A.C. Smith
Department of Electrical Engineering and Computer Science
Chairman, Committee on Graduate Students
ABSTRACT

B-Trees are ideal structures for building databases with fixed-size keys, and have been successfully extended in a variety of ways to accommodate specific key distributions; however, in the general case in which the key distribution either is unknown beforehand or is intentionally pathological, even the most time-honored B-Tree variants—such as prefix-compressed trees—provide sub-optimal performance; e.g., Sleepycat’s poor performance on key distributions with many large keys. Insufficient generality in dealing with different key distributions makes most B-Tree variants unsuitable for general applications such as file systems.

Furthermore, implementations of B-Trees are often limited to either preemptive or cooperative multithreaded operation with synchronous I/O primitives: the overhead caused by lock contention and multiple stacks makes this an insufficient solution for highly-parallel tasks.

This thesis helps fill the void in these areas by introducing and analyzing the performance of a C++ implementation of the String B-Tree of Ferragina and Rossi that meets certain efficiency and flexibility constraints—such as operating within typical B+ Tree time bounds and providing good performance on long arbitrarily-distributed keys—while requiring only asynchronous I/O primitives.

Thesis Supervisor: Frans Kaashoek
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: David Mazieres
Title: Ph.D. Student, Department of Electrical Engineering and Computer Science
1 Introduction

B-Tree variants are among the simplest, most elegant structures used to create databases, and are actually widely used in practice. Although the standard B-Tree is limited to fixed-size keys, later research resulting in implementations supporting variable-length keys and providing optimal pagination satisfy the following requirements, necessary for truly generic databases:

- Many B-Tree variants support variable length keys in a non-trivial manner. By this I mean that adding a few large keys to a key distribution increases the size of the tree only locally.

- B-Trees attempt to minimize the number of disk accesses. Local database performance is typically bounded by the high latency and low throughput of disk access. B-Trees, combined with an effective cache, minimize the number of disk accesses required for basic operations.

- B-Trees support random access inserts, finds, and removes in \(O(\log m)\) disk accesses and ordered, sequential seeks in \(O(1)\) disk accesses, where \(m\) is the number of keys in the database. Hash tables, while typically much faster for the first three operations, do not support ordered, sequential access.

Despite meeting these requirements, traditional B-Tree design relies on a single assumption that is not true for all databases: that keys are typically small compared with the block-size, with rare exceptions. Typical key/value databases store some of each key in the index nodes, providing overflow nodes for the remainder; unfortunately, these overflow nodes not only introduce extra latency into database accesses, but also provide the user little a priori information about the time required to complete an operation. One of the contributions of this project is a B+ Tree variant that performs well on key distributions with many large keys.

Additionally, a source of inefficiency in most current implementations is disk I/O. Making the assumption that the underlying operating system is good at scheduling disk accesses, we should like to provide the OS with as much choice as possible in determining the order of such accesses: typically, B-Trees (and databases in general) accomplish this through the use of cooperative or preemptive threads along with the synchronous I/O primitives provided by the operating system’s C library.

Unfortunately, this leads to problems with lock contention and to the overhead of multiple stack frames. Therefore, another primary contribution of this project is the use of asynchronous I/O primitives, which avoid these problems while still maximizing the concurrency of disk accesses. Specifically, we use the asynchronous I/O library developed for the Self-certifying File System (SFS, [6]).

Finally, we wish to evaluate the real-world performance of a new algorithm, the String B-Tree[5] of Ferragina and Rossi.
2 Design

The design of the algorithm implemented in this project—the String B-Tree—is motivated in large part by the limitations of B+ Trees and B+ Tree variants.

In a standard B+ Tree, keys are stored in the nodes of the tree, requiring a constant keysize to maintain invariants required for the standard pagination operations.[2] This led to the development of B+ Tree variants that supported variable-length keys.[2, 3]

However, these solutions only increased applicability; they did not attempt to deal with the resulting inefficiency caused by the many real-life databases with key distributions in which many keys share large substrings.¹ The entire key would still have to be stored in the index, leading to a very low branching factor for those index nodes with very large keys.

Prefix compression[1] addresses a specific instance of this problem: that many types of databases will contain many keys sharing a common prefix. The solution is to “compress” the keys in a particular index node by storing the shared prefix only once: only the portion of each key after the prefix would be individually saved.

This unfortunately does not work well for all databases. Consider a database with uniformly-chosen keys. The expected number of long keys with any x-byte uniformly-chosen prefix in a database of $2^{24}$ keys is $2^{24-8x}$: when x is just 3, we expect only one key to have a given prefix. In this instance, prefix compression is essentially useless, so the number of keys in any index node will be small, with a correspondingly low branching factor.

2.1 String B-Tree

Ferragina and Grossi come to the rescue with their String B-Tree.[5] Instead of storing prefix-compressed keys at each index node, each key is stored in full (perhaps along with its associated value) in a consecutive sequence of data blocks, and each downward-traversal decision is made by a combination of Patricia trie[7] search and the consultation of a single key.

2.1.1 Notation and conventions

- **String B-Tree nodes**—also called index nodes—are represented by Greek letters. Index nodes can be divided naturally into two sets: internal index nodes, and leaf index nodes. The number of child index nodes of an internal index node $\pi$ is denoted by $|\pi|$, and the child index node associated with a particular key $D$ (either $L_i$ or $R_i$ for some $i \in \{1, 2, \ldots, |\pi|\}$) is $C(D)$.

- For ease of exposition and analysis, we will assume that all keys in the database end with a unique termination character "$"; we later remove this requirement in the implementation.

- Keys are represented by capital letters. The $i^{th}$ character (starting from 0) of a key $T$ is denoted by $T[i]$.

¹This is evident in the UNIX file system: typically, half the files on a given machine will have the prefix "/usr/".
• Elements specific to a particular index node may be associated with index node name: e.g., a key $T$ that is associated with an index node $\pi$ may be written $\pi.T$. This does not necessarily mean that $T$ is stored in $\pi$; $T$ may be a temporary variable used in some context associated with $\pi$.

• The set of keys in the database is denoted by $\Delta$.

• The blocksize (in bytes) of our String B-Tree's underlying block database is $B$.

• The location of some index node $\pi$ on the disk is denoted $\&\pi$.

• Patricia tries in general are labelled by $PT$, perhaps subscripted (as in $PT_L$) or associated with some index node (as in $\pi.PT$).

2.1.2 Overview of String B-Tree design

A String B-Tree is like a B+ Tree in that the values are referenced only at the leaves of the tree, while the internal nodes contain only branching information. However, it differs from traditional B+ Tree variants in several ways:

• In a standard B+ Tree, an internal index node with $k$ keys points to $k + 1$ children: the first child points to those keys less than or equal to the first key; the second child points to those keys greater than the first key but less than or equal to the second key; and so forth.

In a String B-Tree, an internal index node $\pi$ with $|\pi|$ children "contains" (in a loose sense, as we shall see) $2|\pi|$ keys, represented by the ordered set $\pi.\delta = \{L_1, R_1, L_2, R_2, \ldots, L_{|\pi|}, R_{|\pi|}\}$ in which $L_1 < R_1 < L_2 < R_2 < \cdots < L_{|\pi|} < R_{|\pi|}$. Two useful invariants are maintained on the keys in this node:

1. The $i$th child index node contains both $L_i$ and $R_i$, and the leaves of the subtree rooted at this child contain exactly those keys in $\Delta$ greater than or equal to $L_i$ but less than or equal to $R_i$.

2. All keys $T$ in $\Delta$ such that $L_i \leq T \leq R_{|\pi|}$ must satisfy $L_i \leq T \leq R_i$ for some $i$. By induction over the structure of the String B-Tree, every key in the database must be "covered" by some index node at each level of the tree.

These invariants must be maintained by all operations, as they are an integral part of the String B-Tree's search algorithm.

• Unlike typical B+ Tree variants, neither keys nor subsequences thereof are stored in the index nodes themselves: rather, a Patricia trie is used to minimally distinguish between the keys "contained" in that node, while each actual key is stored in full elsewhere in the database.

Briefly—as it will be described in full further on—a Patricia trie is a compacted trie with only the first character of each branching string stored in the trie. Figure 2 shows a compacted trie and its corresponding Patricia trie. Right now we need know only
Figure 1: An example of a String B-Tree with two levels. Each box represents an index node containing a Patricia trie, each leaf of which points to a key in some consecutive sequence of blocks in the database. Internal B-Tree index nodes (such as the root in the above example) contain Patricia tries with an even number of leaves, ordered lexicographically by the keys to which they point. The $i^{th}$ consecutive pair of leaves then points to keys $L_i$ and $R_i$, which are associated with a child B-Tree index node $C(L_i) = C(R_i)$.

that the search key $P$, $\pi.PT$, and the choice of a single additional key (one of the $L_i$ or $R_i$ from $\pi.\delta$) are sufficient to determine the lexicographic position of $P$ among $\pi.\delta$.

As a result of our assumption that all keys end with a unique termination character, we can assume for the remainder of this section that all complete keys represented in a Patricia trie terminate at leaf nodes. Thus, each of the keys “contained” in an internal index node is represented by a leaf in its Patricia trie. This assumption will be relaxed in the implementation section.

- As with the internal index nodes, a leaf index node also contains a Patricia trie. The number of trie leaf nodes in each of these tries is equal to the number of key/value pairs represented by the leaf index node, under the same assumption that keys end with a unique termination character.

To simplify the notation, the set of strings represented by a leaf index node $\pi$ is also denoted by $\pi.\delta$, although it does not have any of the restrictions imposed when $\pi$ is an internal index node (e.g., that $|\pi.\delta|$ be even).

A visualization of the String B-Tree structure is given in figure 1.
2.1.3 SBT-find(String P)

Traversing a String B-Tree is substantially different from traversing a typical B+ Tree variant, in which there is always an obvious child to which to traverse from any internal index node with \( k + 1 \) children: for any search key \( P \), there is always either one consecutive pair of keys \((K_i, K_{i+1})\) in the node for which \( K_i < P \leq K_{i+1} \), or \( P < K_1 \) or \( P > K_k \), and all pairs plus the two endpoints are each associated with a unique child index node.

The structure of the String B-Tree index nodes motivates a different search algorithm.

**Internal index nodes.** At each internal index node \( \pi \), we determine (using \( \pi.PT \) and either \( L_i \) or \( R_i \) for some \( i \)) the lexicographic position of \( P \) among the keys “contained” in that node. As will be shown later in our discussion of Patricia tries, our search on \( PT \) must result in one of two cases:

1. If \( L_i \leq P \leq R_i \) for some \( i \), then we continue the search for \( P \) at child index node \( C(L_i) = C(R_i) \). Note that, in this case, \( P \) may actually be in \( \Delta \), though we don’t know this a priori.

2. If \( R_i < P < L_{i+1} \) for some \( i \), then we traverse to \( C(R_i) \) or \( C(L_{i+1}) \) arbitrarily; if \( P < L_1 \), then we traverse to \( C(L_1) \); if \( P > R_{|\pi|} \), we traverse to \( C(R_{|\pi|}) \). From this point forward, we know that \( P \) is not in \( \Delta \).

**Leaf index nodes.** Once we reach a leaf index node \( \pi \), we determine (again using \( \pi.PT \) and a single additional key) the lexicographic position of \( P \) among the keys “contained” in that node. If \( P \in \Delta \), then \( P \) will match one of the keys in this node; otherwise, \( P \)'s proper insertion point is either immediately before \( \pi \)'s first key, between two consecutive keys, or immediately after \( \pi \)'s last key.

The proof that this procedure results in finding the lexicographic position of \( P \) among \( \Delta \) can be done simply by induction.

2.1.4 SBT-insert(String P, KeyPtr K, ValuePtr V)

We first perform a downward traversal of the String B-Tree to find \( P \)'s insertion point in a leaf index node \( \pi \). If \( P \) already exists in the database, we simply replace the pointer to the old value with a pointer to the new value, and deallocate the space used by the old value.

If \( P \) does not exist in the database, then we insert it into \( \pi.PT \) and associate the new trie leaf node with the key/value pointer pair \((K, V)\). At this point we may need to perform some cleanup: if \( P \) is the least or greatest key in \( \pi \), the parent index node will need to have one of its keys replaced in order to satisfy our String B-Tree invariants. Since this may change the least or greatest key in \( \pi \)'s parent, this operation may cascade to the root.

Additionally, \( \pi \) may exceed its key limitation and need to be split, an operation that is discussed later. Since a split introduces two new strings into \( \pi \)'s parent, this operation may also cascade to the root. The insertion of these two new strings at each level appears to require an unbounded number of block loads; however, as we will show later, this is not the case.
2.1.5 SBT-remove\((\text{String } P)\)

As with the insert operation, removing a key \(P\) from the database involves a traversal to find \(P\)'s insertion point in a leaf index node \(\pi\). If \(P\) does not exist in the database, then there is nothing to do.

If \(P\) does exist, then we remove \(P\) from \(\pi.\text{PT}\) and deallocate the space associated with \(P\)'s value. As in the insert case, we may need to perform cleanup if \(P\) was the least or greatest key in \(\pi\), although here it is more extensive since the next-to-least or next-to-greatest key may not be in memory: although this appears to require an unbounded number of block loads, the discussion on Patricia tries later demonstrates a procedure that enables us to do this with no additional block loads.

Finally, if \(\pi\) and one of its adjacent sibling nodes both have too few keys, we combine them using the join operation discussed later. Analogously to the insert case, this operation may remove two keys from \(\pi\)'s parent, causing it to cascade to the root. If the root index node drops to only two keys (i.e., one child), we remove it and make the only child the new root.

2.2 Patricia tries

A detailed description of Patricia tries can be found in [5]. As stated earlier, a Patricia trie is a compacted trie with only the first character of each branching string stored in the trie. Figure 2 shows a compacted trie and its corresponding Patricia trie.

We first describe a procedure for finding the insertion position of a key \(P\) in index node \(\pi\) using potentially all characters of some key in \(\pi.\text{PT}\), which leads to a worst-case bound on disk accesses of \(O(|P|/B \cdot \log_B m)\), where \(B\) is a minimum branching factor based on \(B\) and properties of Patricia tries; later, we improve this bound substantially to \(O(|P|/B + \log_B m)\) by noting that we can upper bound the total number of blocks loaded during an entire String B-Tree search.

2.2.1 Notation and conventions

- As with index nodes, Patricia trie nodes can also naturally be divided into two sets, one of the internal trie nodes and the other of the leaf trie nodes. The assumption that there is a unique termination character implies that a key represented by a Patricia trie must terminate in a leaf node.

- The string represented by a particular node in the Patricia trie is found by concatenating the substrings along the path from the root to that node in the corresponding compacted trie. Only some nodes have strings that are also full keys: under the assumption that each key ends in a unique termination character, these are exactly the leaves of the trie.

- Denote by \(S(x)\) the string associated with trie node \(x\). The "length" of a Patricia trie node \(x\) is the length of \(S(x)\), and is denoted by \(\text{len}(x)\). Note that \(S(x)\) ends with the termination character \(\$\) only if \(x\) is a leaf.

- The successor leaf node of a Patricia trie node \(x\) is denoted by \(\text{succ}(x)\); the corresponding predecessor leaf node would be \(\text{pred}(x)\).
Figure 2: On the left is a compacted trie. Note that any string represented by a node in the trie can be constructed by concatenating all of the substrings on that node's path from the root: e.g., the string at the leftmost leaf aaba is constructed by following the branches a, ab, and a in order. On the right is a Patricia trie, in which this property no longer holds: notice that only the first character of each branch is stored in the trie. A node is labelled with its "length," which is the length of the string represented by that node.

### 2.2.2 PT-search(PT, String P)

We wish to find a path from the root index node of the String B-Tree to a leaf index node in which either (a) we find a pointer to P's associated value or (b) we can add P and maintain a lexicographic ordering of the keys.

At any internal index node $\pi$ in our String B-Tree traversal, we perform a downward traversal ("blind search") of the Patricia trie in that index node using the characters of $P$: at an internal trie node with length $j$, we choose the branch associated with character $P[j]$.

Eventually, we will either "get stuck"—get to some node of length $j$ in which no branch matches $P[j]$—or reach some leaf node in the Patricia trie. (If we get stuck in the traversal, we choose some arbitrary leaf of the subtrie below the stuck node.) Call the leaf we reach $\ell$.

$\ell$ is associated with some key $T = S(\ell)$ located in one or more consecutive data blocks. Since the Patricia trie contains the minimal set of characters required to distinguish between the keys of that index node, if $P$ is not one of the strings in that index node it is possible that our traversal led to the wrong position in the trie, i.e., one in which a move to $C(T)$ will lead to an incorrect leaf index node. See figure 3 for an example.
**Smart search.** As our goal is to find the insertion point of $P$, we need to perform additional work if $P$ is not in $\pi.\delta$; this we call “smart search,” a procedure for correcting the possible mistake made during blind search.

After completing the blind search for $P$, we load $T$ and compare it character-by-character with $P$ to determine in which position $i$ they first differ. We then move back upwards through the trie to the highest node $h$ along the path to $\ell$ for which $\text{len}(h)$ is greater than or equal to $i$. Ferragina and Grossi call this the “hit node.”

From $h$ and $T[i]$, we can determine the correct lexicographic position for $P$ among $\pi.\delta$:

1. If $\text{len}(h) = i$, then none of the branching characters $b_0 < b_1 < \cdots < b_k$ from $h$ match $P[i]$. If $b_j < P[i] < b_{j+1}$ for some consecutive branching characters $b_j$ and $b_{j+1}$, then the lexicographic position of $P$ in $\pi.\delta$ is directly after string $D = S(x)$, where $x$ is the rightmost leaf of the subtrie rooted at the child along branch $b_j$. (Note that $D = L_x$ or $D = R_x$ for some $x \in \{1, \ldots, |\pi|\}$.)

   If $P[i] < b_0$, then the lexicographic position of $P$ is immediately before the string $D = S(y)$, where $y$ is the leftmost leaf of the subtrie rooted at the child along branch $b_0$.

   Similarly, $P[i] > b_k$ implies that $P$’s lexicographic position is immediately following the string $D = S(z)$, where $z$ is the rightmost leaf of the subtrie rooted at the child along branch $b_k$.

2. If $\text{len}(h) > i$, then all strings associated with leaves in the subtrie rooted at $h$ have character $T[i]$ in position $i$. Denote by $x$ the leftmost leaf of this subtrie, and by $y$ the rightmost leaf. Then, if $P[i] < T[i]$, the lexicographic position of $P$ is immediately before the string $D = S(x)$; otherwise, if $P[i] > T[i]$, it is immediately after the string $D = S(y)$.

Note that all strings associated with leaves of subtries rooted at $h$’s siblings must be either strictly less than or strictly greater than $P$ lexicographically, since they differ from $P$ in some index $j < i$.

Using this “smart search,” we can determine the correct child index node $\tau = C(D)$ of $\pi$ to which to branch, without requiring the index nodes to themselves contain the keys. An example of the smart search procedure is given in figures 3 and 4.

**Reducing disk access.** Ferragina and Grossi note that we don’t actually need to load all of string $T$ (which can be very large) when performing the smart search if we know a priori that the first $l$ characters (indices $0, \ldots, l-1$) of $P$ and $T$ match: in this case, we need only load those blocks of $T$ containing indices $l, \ldots, i$. It turns out that we can keep track of $l$ iteratively through the String B-Tree search.

Say we determine $\pi.i$, $\pi.T$, and $\pi.D$ in the blind and smart search procedures in $\pi$’s Patricia trie, and traverse to child index node $\tau = C(D)$ of $\pi.D$. It must be the case that some string $Z \in \pi.\delta$ matches the first $\pi.i$ characters of $P$, since $P$ matches the first $\pi.i$ characters of $\pi.T$, $\pi.T$ matches (at least) the first $\pi.i$ characters of $\pi.D$, and $\pi.D$ must be in $\tau.\delta$ by our primary invariant.
Figure 3: In the blind search for \( P = \text{abba} \), we will follow the path indicated by the dashed line. Note that we make a mistake on the second arc: in the corresponding compacted trie, that branch has \( a \) as its second character, but the character in the same position in the search string is \( b \). We find the mistake by loading string \( T = \text{abaaa} \) and comparing it character-by-character with \( P \), until we find they do not match at index \( i = 2 \).

Figure 4: We then proceed back to the “hit node,” the highest node with length \( \geq i \), which in this case is the node on our path immediately below the mistake. We then follow rule 2 since \( \text{len}(h) = 3 \) and \( i = 2 \), and proceed to find the insertion point between \( \text{ababb} \) and \( \text{baa} \) since \( b = P[2] > T[2] = a \).

Consider a string \( X \) in \( \tau \delta \) that differs from \( P \) in some index \( j < \pi.i \). Then, since \( P[j] = Z[j] \), it must be that \( X[j] \neq Z[j] \), so there must be some node in \( \tau \cdot \text{PT} \) with length \( j \) at which one branch heads toward \( X \) and another toward \( Z \). Clearly, a blind search for \( P \) starting at this node would take the branch leading toward \( Z \). By induction over the nodes with length \( \leq \pi.i \), our blind search must lead to a string \( \tau.T \) which matches at least the first \( \pi.i \) characters.

Thus, for the \( l \) in a Patricia trie search we can actually use the \( i \) from the smart search performed in the parent’s Patricia trie.

**Analysis.** Given the technical description of Patricia tries, we can finally demonstrate bounds on block accesses for the String B-Tree search.

The first thing to notice is that we can derive a lower bound on the branching factor of each index node. A Patricia trie of \( 2|\pi| \) keys contains at most \( 4|\pi| + 1 \) nodes (since every node except the root must have at least two children), so we can represent the trie in \( O(|\pi|) \) space.
Its independence of the sizes of the individual keys gives us a lower bound $B$ on the number of keys per node. Thus, the height of a String B-Tree with $m$ strings is at most $\log_B m$.

Given the lower bound on the branching factor, we can derive good bounds on the number of disk accesses required by a search. Throughout a String B-Tree search through index nodes $\pi_0, \pi_1, \ldots, \pi_M$, we need to load at most

$$\left\lceil \frac{i_{\pi_0}}{B} \right\rceil - \left\lfloor \frac{0}{B} \right\rfloor + \left( \left\lceil \frac{i_{\pi_1}}{B} \right\rceil - \left\lfloor \frac{i_{\pi_0}}{B} \right\rfloor \right) + \left( \left\lceil \frac{i_{\pi_2}}{B} \right\rceil - \left\lfloor \frac{i_{\pi_1}}{B} \right\rfloor \right) + \cdots + \left( \left\lceil \frac{i_{\pi_{\log_B m}}}{B} \right\rceil - \left\lfloor \frac{i_{\pi_{\log_B m-1}}}{B} \right\rfloor \right)$$

$$\leq \left\lceil \frac{i_{\pi_{\log_B m}}}{B} \right\rceil + \log_B m$$

$$\leq \left\lceil \frac{|P|}{B} \right\rceil + \log_B m$$

blocks of key data. The $\log_B m$ comes from the fact that we may be loading the same section from more than one distinct key during our traversal down through the index nodes, but this is eaten up by the index node loading itself. Thus, the total number of blocks to be loaded in any one String B-Tree search operation is $O(|P|/B + \log_B m)$.

### 2.2.3 PT-insert(PT, String P, KeyPtr K, ValuePtr V)

We first perform a traversal using the above procedure to find $i$, $h$, and $D$. We then ascend from the leaf associated with $D$ to the hit node $h$. From here, one of two cases applies:

1. If $\text{len}(n) = i$, then we insert a branch from $n$ along character $P[i]$ to a new node with key/value pointer pair $(K, V)$. We know there is not already a node here since $P[i] \neq D[i]$ and there is no string in PT that shares a longer prefix with $P$ than $D$.

2. If $\text{len}(h) > i$, then we insert a new node $n'$ between $n$ and its parent, with one branch along character $D[i]$ leading to $n$ and another along character $P[i]$ leading to a new node with key/value pointer pair $(K, V)$.

### 2.2.4 PT-remove(PT, PTNode n)

Given a node $n$ in trie PT, PT-remove removes $n$ and perhaps removes $n$'s parent. If $n$'s parent now has only one child, we remove that node as well.

### 2.2.5 PT-split(PT)

Given an input trie PT, PT-split needs to return two roughly equivalent-sized tries $PT_L$ and $PT_R$ together covering all strings in PT and such that all strings in $PT_L$ are less than all strings in $PT_R$.

We first choose some appropriate leaf node $n$ at which to split PT. We then divide PT into two pieces: $PT_L$ contains copies of all nodes along the path, plus every node to the left of the path; $PT_R$ also contains copies of all nodes along the path except $n$, plus every
Figure 5: An illustration of a Patricia trie split. All nodes on and to the left of the dashed path are included in \( X_L \), and all nodes on and to the right of the dashed path (except \( N \)) are included in \( X_R \). The nodes marked with crosses are unnecessary and will be removed during clean up.

node to the right of the path. We then proceed to “clean up” \( PT_L \) and \( PT_R \) by removing unimportant nodes, i.e., those nodes other than the root having only one child. A trie split is diagrammed in figure 5.

### 2.2.6 \( PT\text{-concat}(PT_L, PT_R) \)

\( PT\text{-concat} \) is essentially the opposite of the \( PT\text{-split} \) operation: given two tries \( PT_L \) and \( PT_R \), the position \( i \) in which the greatest string \( S_L \) in \( PT_L \) and the least string \( S_R \) in \( PT_R \) differ, and \( S_L[i] \) and \( S_R[i] \), we join the right spine of \( PT_L \) to the left spine of \( PT_R \) through position \( i \), removing redundant nodes if necessary.

### 2.2.7 Revisiting SBT-remove

We now return to a problem mentioned previously: that removing the least or greatest key from \( \pi \text{. PT} \) requires us to insert a new key into the Patricia trie \( \tau \text{. PT} \) of \( \pi \text{’s parent index node} \) \( \tau \) in order to maintain our String B-Tree invariants. While it seems at first that we need to load the next-to-least or next-to-greatest key in order to insert it into \( Y \), this is not actually the case.

In the following, \( \pi \) is a String B-Tree index node and \( \tau \) is the parent index node of \( \pi \).

First we define some helper functions. Briefly, \( PT\text{-diff} \) produces the description of the first difference between two strings \( A \) and \( B \) in nodes \( a \) and \( b \) of \( \pi \text{. PT} \), respectively, and \( PT\text{-patch-insert} \) uses this difference information to patch \( B \) into \( \tau \text{. PT} \), which contains \( A \) but not \( B \).²

²\( PT\text{-patch-insert} \) actually has a more specific restriction: \( \pi \text{. PT} \) cannot contain any string that shares a longer prefix with \( B \) than \( A \) does.
- \text{PT-diff}(\pi.\text{PT}, \text{PTNode } a, \text{PTNode } b)$, where $a$ and $b$ are trie leaf nodes in $\text{PT}$.

We know just from looking at the trie how exactly $\pi.S(b)$ first differs from $\pi.S(a)$: the path from the root to $a$ will diverge from the path from the root to $b$ at some node whose length we call $i$. Then, we know that $\forall k < i$, $\pi.S(a)[k] = \pi.S(b)[k]$ but $\pi.S(a)[i] \neq \pi.S(b)[i]$, where $\pi.S(a)[i]$ is the branching character leading toward $a$ and $\pi.S(b)[i]$ is the branching character leading toward $b$.

We return $(i, \pi.S(a)[i], \pi.S(b)[i])$.

- \text{PT-patch-insert}(\tau.\text{PT}, \text{PTNode } x, i, \text{char } x_i, \text{char } n_i, \text{KeyPtr } K, \text{ValuePtr } V)$, where $x$ is a trie leaf node in $\tau.\text{PT}$ and $x_i = \tau.S(x)[i]$.

Say $N$ is the string we are attempting to insert into $\tau.\text{PT}$ (i.e., one for which $N[i] = n_i$). We find the highest node $h$ along the path from the root of $\tau.\text{PT}$ to $x$ for which $\text{len}(x) > i$; then, one of two cases applies:

1. If $\text{len}(h) = i$, then we add a branch from $h$ along character $n_i$ to a new leaf node with key/value pointer pair $(K, V)$.
2. If $\text{len}(h) > i$, then we add a new node $h'$ with $\text{len}(h') = i$ between $h$ and its parent. A branch from $h'$ labelled by character $x_i$ leads to $h$, and the other branch from $h'$ labelled by character $n_i$ leads to a new node with key/value pointer pair $(K, V)$.

The following is then an approximation of the SBT-remove procedure, neglecting the orthogonal matters of joining two undersized index nodes and of removing the root if it becomes empty:

\textbf{SBT-remove}(P)

1. Traverse to the leaf index node $\pi$ with key $P$, or return if $P \notin \Delta$
2. Set $d$ to be the leaf trie node in $\pi.\text{PT}$ for which $\pi.S(d) = P$
3. If $d$ is neither the leftmost node $\ell$ nor the rightmost node $r$ in $\pi.\text{PT}$, or $\pi$ is the root of the String B-Tree, \text{PT-remove}(\pi.\text{PT}, d)$ and return
4. If $d = \ell$, set $n := \text{succ}(d)$
5. Otherwise, $d = r$, so set $n := \text{pred}(d)$
6. $(i, \ell_i, n_i) := \text{PT-diff}(\pi.\text{PT}, \ell, n)$
7. $(j, r_j, n_j) := \text{PT-diff}(\pi.\text{PT}, r, n)$
8. Set $\tau$ to be the parent of $\pi$
9. If $i > j$
   1. Set $x$ to be the node in $\tau.\text{PT}$ corresponding to $\pi.S(\ell)$
   2. \text{PT-patch-insert}(\tau.\text{PT}, x, i, \ell_i, n_i, n.K, \&\pi)$
10. Otherwise
    1. Set $x$ to be the node in $\tau.\text{PT}$ corresponding to $\pi.S(r)$
    2. \text{PT-patch-insert}(\tau.\text{PT}, x, j, r_j, n_j, n.K, \&\pi)$

\footnote{If, among all strings in $\tau.\text{PT}$, $\pi.S(x)$ shares the longest prefix with $N$, then there is not already a branch along character $n_i$ here: no string $Y$ in $\tau.\text{PT}$ sharing the first $i$ characters with $N$ also satisfies $Y[i] = n_i$.}
11. \( PT\text{-}remove(\pi.PT, d) \)
12. Set \( \pi := \tau \)
13. Set \( d := x \)
14. Go to 3

Using this procedure, we can remove any key from the database without needing to load additional keys beyond those used for the downward traversal.

**Analysis.** We can now complete the analysis of \( SBT\text{-}remove \) very simply: since the only disk accesses required are those on the downward traversal—even in the case that we remove the leftmost or rightmost node from some index node’s Patricia trie—we do not exceed \( O(|P|/B + \log_B m) \) for the entire operation.

### 2.2.8 Revisiting \( SBT\text{-}insert \)

As with \( SBT\text{-}remove \), \( SBT\text{-}insert \) can require us to perform Patricia trie inserts on keys we do not have in memory; and, as before, we find that we do not actually need the keys in memory in order to do this. We can directly use the infrastructure built for \( SBT\text{-}remove \) to complete \( SBT\text{-}insert \), ignoring the orthogonal matter of replacing ancestor keys if the key we insert is the leftmost or rightmost in some Patricia trie:

\[
\text{SBT-insert}(P, \text{KeyPtr } K, \text{ValuePtr } V)
\]
1. Traverse to \( P \)'s insertion point in leaf index node \( \pi \)
2. If \( P \in \Delta \), then replace its associated value with \( V \) and return
3. \( PT\text{-}insert(\pi.PT, P, K, V) \)
4. If \( \pi \) does not exceed its key count limitation, return
5. If \( \pi \) is the root of the String B-Tree, create a new empty root
6. Set \( \tau \) to be the parent of \( \pi \)
7. Set \( \ell \) to be the leftmost node of \( \pi \)
8. Set \( r \) to be the rightmost node of \( \pi \)
9. Create a new index node \( \phi \) as \( \pi \)'s new successor, redirecting pointers as necessary
10. \( (PT_L, PT_R) := PT\text{-}split(\pi.PT) \)
11. Set \( n \) to be the node in \( \pi.PT \) corresponding to the rightmost node in \( PT_L \)
12. \( (i, \ell_i, n_i) := PT\text{-}diff(\pi.PT, \ell, n) \)
13. \( (j, r_j, n_j) := PT\text{-}diff(\pi.PT, r, n) \)
14. If \( i > j \)
   1. Set \( x \) to be the node in \( \pi.PT \) corresponding to \( \pi.S(\ell) \)
   2. \( PT\text{-}patch\text{-}insert(\pi.PT, x, i, \ell_i, n_i, n.K, &\pi) \)
15. Otherwise
   1. Set \( x \) to be the node in \( \pi.PT \) corresponding to \( \pi.S(r) \)
   2. \( PT\text{-}patch\text{-}insert(\pi.PT, x, j, r_j, n_j, n.K, &\pi) \)
16. Set \( m \) to be the node in \( \pi.PT \) corresponding to the leftmost node in \( PT_R \)
17. \( (i, n_i, m_i) := PT\text{-}diff(\pi.PT, n, m) \)
18. \( (j, r_j, m_j) := PT\text{-}diff(\pi.PT, r, m) \)
19. If \( i > j \)
1. Set \( x \) to be the node in \( \tau.\text{PT} \) corresponding to \( \pi.S(n) \)

2. \text{PT-patch-insert}(\tau.\text{PT}, x, i, n_i, m_i, m.K, \&\phi)

20. Otherwise

1. Set \( x \) to be the node in \( \tau.\text{PT} \) corresponding to \( \pi.S(r) \)

2. \text{PT-patch-insert}(\tau.\text{PT}, x, j, r_j, m.j, m.K, \&\phi)

21. Set \( \pi.\text{PT} := \text{PT}_L \)

22. Set \( \phi.\text{PT} := \text{PT}_R \)

23. Set \( \pi := \tau \)

24. Go to 4

Using this procedure, we can successfully perform a String B-Tree insert with cascading splits without loading any additional blocks of key data from the disk.

\textbf{Analysis.} Just as with SBT-remove, we do not need to perform additional disk accesses in SBT-insert above the \( O(|P|/B + \log_B m) \) required for the traversal.

3 Implementation

An implementation of a modified version of the String B-Tree algorithm has been developed in C++ using the asynchronous I/O libraries (hereafter designated “AIO”) developed for SFS.

3.1 Asynchronous I/O

The AIO library provides facilities for performing disk I/O using callbacks: essentially, when a programmer wishes to perform an I/O operation, instead of blocking on completion, he passes to the AIO operation a callback with the logical “next step” in the algorithm. So, for instance, to read a block from disk using blocking I/O and then operate on it with some function \( g \), we would do something like

\begin{verbatim}
FILE *f;
    fread(buf, len, 1, f);
    g(buf);
\end{verbatim}

whereas with AIO we would provide a callback to perform the action:

\begin{verbatim}
ptr<aiofh> f;
    f->read(pos, buf, wrap(g));
\end{verbatim}

The interesting part about this is that we can dispatch many read calls in parallel, whereupon the callbacks will be called in whatever order the underlying I/O operations complete. This allows the operating system to reorder the accesses optimally for the hardware:

\begin{verbatim}
ptr<aiofh> f;
    f->read(pos1, buf1, wrap(g1));
    f->read(pos2, buf2, wrap(g2));
    f->read(pos3, buf3, wrap(g3));
    f->read(pos4, buf4, wrap(g4));
\end{verbatim}
aiobuf. The AIO library provides aiobuf, a class of buffer that is allocated in an asynchronous manner and which avoids memory fragmentation; however, for the purposes of this discussion, it can be thought of as a character array.

3.2 Database components

The implementation of the database has been split up into three primary logical components: a generic block database; a Patricia trie implementation; and a String B-Tree implementation based on the previous two components. Various support data structures (container maps, sets, and lists) support these three components.

3.2.1 Block database

The block database is a database operating on whole blocks and consecutive groups of whole blocks, and that enforces atomicity on user defined operations (called "requests"). It provides facilities for allocating and deallocating groups of blocks, for loading and saving multi-block structures (called "elements"), as well as for retrieving unstructured, read-only subsequences of one or more whole blocks. It also provides an element cache for recently-accessed elements.

It is implemented as class BDB, and provides a minimal interface:

typedef uint32_t blockd; // block descriptor
typedef uint32_t blockct; // block count

class BDB {
    // Buf<T> is a container buffer for an aggregate or scalar
    // parameterized type T
    typedef Buf<unsigned char> Cbuf;

    // attempts to dispatch any new requests that have been created
    void dispatch_requests();
};

As is obvious from the spartan interface above, nearly all interaction with the block database is done through the friend class BDB::Request, which are discussed below. Internally, BDB stores pointers to cached elements and requests in private hash tables; to minimize the impact of memory management, nearly all data managed directly by the block database are reference counted.

Elements. An "element" is a structure that can be written in binary form to one or more blocks in the database: examples include index nodes and data nodes in a String B-Tree. Elements must provide minimal facilities for constructing themselves from binary data and converting themselves to binary data, to simplify the storage model.

Elements are implemented as classes derived from the base class BDB::Element. The following interface must be implemented by each derived element X:  

4 from_bin.aiobuf and allocate do not, of course, have to be class functions.
class X : public BDB::Element {
    X(BDB *bdb, blockd start, blockct count, ...) : BDB::Element(bdb, start, count) {
        ... }

    // converts an instance of X into a binary data
    // and writes it to an aiobuf
    void
    to_bin_aiobuf (ptr<aiobuf>) const;

    // passed as a callback to Request::load_element,
    // constructs a new instance of X from an aiobuf
    static ptr<BDB::Element>
    from_bin_aiobuf (ptr<aiobuf>,
                    BDB *db,
                    blockd,  
                    blockct);

    // duplicates an instance of X
    ptr<BDB::Element>
    copy () const;

    // passed as a callback to Request::allocate_element,
    // creates a new instance of X
    static ptr<BDB::Element>
    allocate () const;
};

From the time of its allocation by the block database, an element is associated with some con-
secutive sequence of blocks on the disk, and cannot be moved to another location or resized.
Therefore, the base class BDB::Element can provide two useful methods: block_start(),
which returns the first disk block allocated for the element; and block_count(), which re-
turns the number of blocks allocated for the element.

Blocks are allocated for elements using the binary buddy block allocation algorithm: the
total store is first divided into regions of some equal size $2^N$ bytes; then, each of these regions
is recursively subdivided into two equally-sized subregions (called “buddies”), down to regions
of size $2^n$ bytes. Allocation of a sequence of bytes must be performed on a region boundary,
and removes all its subregions from the free block list; this implies that an allocation request
for a sequence of $x$ bytes is effectively rounded up to one for $2^{\lceil \log_2 x \rceil}$ bytes, i.e., up to the
next power of two.[4]

Conversion to and from binary form (using to_bin_aiobuf and from_bin_aiobuf) can be
expensive, depending on the internal representation of the element. In the case of a String
B-Tree index node, for example, the Patricia trie is represented internally as an arbitrary tree
with memory pointers leading from node to node; externally, they are represented as binary
strings structured recursively from the root in prefix order. As a result of both the dynamic
allocation used in the Patricia trie structure and the general complexity of the structure, this
operation is very CPU intensive.

Requests. A “request” to the block database is a logical sequence of operations that must
together be performed atomically (i.e., without interruption the asynchronous I/O facilities). As a result, if a request would block on the retrieval of data from the disk, the request is completely reset and all changes undone; it is later reinitialized once the requested data has been loaded. The rationale behind this design was that, since databases are largely I/O-bound, a transaction system requiring fewer resets would be far more complex without much added performance.

In addition to loading a complete element and constructing it from binary form, a request may ask for sets of single blocks (“partial-elements”), which are loaded from the disk and stored in read-only binary form. However, a partial-element can be in memory only when the associated full element is not; therefore, a request needs to be able to glean its needed information from either form, since it does not know a priori which form it will have access to.

Requests are implemented as classes derived from the base class BDB::Request. The following interface must be implemented by each derived request X:

class X : public BDB::Request {
    X(BDB *bdb, ...) : BDB::Request(bdb) {
        ...
    }

    // called by BDB::dispatch_requests to start (or restart)
    // the execution of the request
    void init();

    // reports an exception resulting from the failed execution
    // of some asynchronous operation
    void exception(const BDB::Exception&);

    // called after the request has been completed but before
    // this instance of X has been destroyed
    void post();
};

Creating a new request is as simple as calling new X(bdb, ...).

Since a request may be restarted at any time, no new requests may be allocated anywhere in the execution tree starting from init(): due to AIO-induced restarts, such a request may inadvertently be re-created several times. Along these lines, some restrictions on the data that requests may preserve across restarts unenforceable at either compile-time or run-time include:

- no pointers to any elements or partial-element data
no block descriptors

These may not be preserved across request restarts because elements may move around in memory, partial-element data may be evicted from memory, and blocks on disk may be allocated or deallocated as the result of another request's completion; systems build on top of the block database must determine a set of invariants that will be true at the start of any request (e.g., the location of the superblock in the String B-Tree implementation) so the request can actually assume something useful about the state of the database.

Requests have access to the following interface from BDB::Request:

typedef callback<void> generic_cb;

class BDB {
    // callback<R, P1, P2, ...> is a template class that performs a
    // type of limited function currying on functions with return
    // type R and parameter types P1, P2, ...
    typedef callback<void, ptr<const Element> >::ref
        view_data_cb;

    typedef callback<ptr<Element>, ptr<aiobuf>, BDB*, blockd, blockct>::ref
        from_bin_aiobuf_cb;
};

class BDB::Request {
    // allocates space in the database and constructs a new
    // instance of an element using allocate_new_cb
    ptr<Element>
    allocate_element (size_t,
        BDB::allocate_new_cb);

    // frees space in the database associated with an element
    // that does not necessarily have to be in memory
    void
    free_element (blockd,
        blockct);

    // executes the viewer callback if the requested element is
    // already in memory; otherwise, initiates an asynchronous load
    void
    load_element (blockd,
        blockct,
        BDB::from_bin_aiobuf_cb,
        BDB::view_data_cb);

    // executes the callback if the requested partial-element

Element cache. Once a request asks for a particular element, that element is locked in memory—in the element cache—until the completion of the request. Only when all requests locking a particular element have completed can the element be saved and evicted from the cache.

An element is not immediately flushed from the cache once the last request for which it is locked completes; rather, some arbitrary elements are flushed once the cache reaches a threshold size and a new element is loaded from the disk. This delayed eviction is motivated both by the complexity of the binary-internal conversion and by the presumption that keeping an element’s data in user space is significantly more efficient than moving it in and out of the buffer cache as needed.

There is also a partial-element cache that operates in much the same way, with one primary difference: if the full element encompassing some cached partial-element is loaded, the partial-element is evicted to preserve consistency.
3.2.2 Patricia tries

Patricia tries are implemented basically as stated in [5], with the following exceptions:

- No termination character. The requirement for a unique termination character is trivial on paper, but is inefficient to implement. The result is that nearly all Patricia trie functions have to change in at least a minor way to support the lack of a termination character.

  - The structure of the Patricia trie is changed to support key/value pointer pairs at each node, and the dichotomy between leaf trie nodes and internal trie nodes is replaced by one between those with key pointers and those without. Call those nodes with key pointers “key nodes” and those without “non-key nodes.”

  - For any string \(A\) and any non-empty string \(B\), \(A < AB\) lexicographically.

  - For a key node \(x\), \(\text{succ}(x)\) points to the key node of the lexicographically succeeding key; similarly, \(\text{pred}(x)\) points to the key node of the lexicographically preceding key.

  - Wherever there is a reference to the “leftmost leaf,” we now refer to the “bottom” key node, i.e., the one for which there is no predecessor.\(^5\)

  - The blind search portion of \(\text{PT.search}(P)\) will not actually reach an index for which \(P[\text{len}(x)]\) does not match any of the branching characters of \(x\) if there is some leaf \(y\) such that \(S(y)\) is a prefix of \(P\); in this case, we choose \(\ell = y\).

  - In the smart search portion of \(\text{PT.search}(P)\), we have two new cases:
    
    - if \(P\) is a prefix of some string in \(\pi.\text{PT}\), then the hit node \(h\) is the first node \(x\) along the path from the root to \(T\) such that \(\text{len}(x) \geq |P|\). The correct lexicographic position of \(P\) is immediately before \(S(x)\) for \(x\) the highest key node along the left spine of the subtrie rooted at \(h\).
    
    - if there is some leaf \(y\) in \(\pi.\text{PT}\) such that \(S(y)\) is a prefix of \(P\), then the hit node \(h = y\). (Note that this case is a true exception: this is the only case in which the hit node \(h\) does not satisfy \(\text{len}(h) \geq |P|\).)

  - \(\text{PT.insert}(\text{PT}, P, K, V)\) has two new cases:
    
    - if \(P\) is the prefix of some string in \(\pi.\text{PT}\) and \(\text{len}(h) > |P|\) for the hit node \(h\), we need to insert the new node between \(h\) and \(h\)’s parent
    
    - if \(h\) is a leaf and \(S(h)\) is a prefix of \(P\), then we insert the new node along branch \(P[\text{len}(h)]\).

  - \(\text{PT.remove}(\text{PT}, n)\) must be changed in the following ways:

\(^5\)The astute reader will note that this must be some node along the left spine of the trie.
• If \( n \) is an interior node, we first check to see if it is a key node; if so, we remove the key/value pointer pair and then remove \( n \) if it is redundant (i.e., has only one child).
• If \( n \) is a leaf and we remove it, we do not remove \( n \)'s parent if it is a key node.

• \texttt{PT.split()} first chooses some key node \( n \), and splits \( PT \) into two pieces: \( PT_L \) contains copies of all nodes on the path \texttt{root\rightarrow n} plus copies of those nodes to the left of the path; \( PT_R \) contains copies of all nodes on the path \texttt{root\rightarrow n} plus copies of those nodes to the right of the path and a copy of the subtrie rooted at \( n \).

  Additionally, since key nodes can be interior nodes, we need to remove the key/value pointer pairs from the copy of the path \texttt{root\rightarrow n} in \( PT_R \).

  Finally, we remove the redundant nodes (non-key nodes with only one child) along the right spine of \( PT_L \) and along the left spine of \( PT_R \).

• \texttt{PT.concatenate(PTL, PTR)} is essentially the same operation as \texttt{PT-concat}, except that only non-key nodes can be considered redundant during clean-up.

  o No \texttt{PT.diff} or \texttt{PT.patch-insert}. Due to time considerations and the difficulty of adapting these operations to the lack of a termination character, these operations were not implemented in time for the analysis.

\textbf{Asynchronous operation.} When the programmer wishes to perform a Patricia trie search, insert, or remove operation, he must provide a callback that performs the string comparison between the search string \( P \) and the blind search string \( T \); this callback then continues the trie search procedure with the location of the difference between the strings (\( i \)), the strings' characters at that point (\( P[i] \) and \( T[i] \)), and whether \( P \) is a prefix of \( T \) or lexicographically before \( T \), \( T \) is a prefix of \( P \) or lexicographically before \( P \), or \( P \) is equal to \( T \). This setup is designed to allow the comparison function to make AIO calls that may cause the request to restart.

  Since this information is sufficient to perform the search, insert, and remove operations, the complexity of the string comparison is left up to the owner. For example, in a simpler block database implementation without the ability to load partial-elements, all of \( T \) can be loaded to perform the comparison.

\textbf{Parameterized character type.} The Patricia trie is implemented as a template class parameterized over the character type (typically \texttt{char} or \texttt{unsigned char}), the string location descriptor type (an instance of which is passed to the comparison function to describe the location of the requested string), and a data type an instance of which is associated with each complete key. The use of templates here allows one to use more complex character types (say \texttt{wchar} for Unicode) without modifying any of the Patricia trie implementation.

\textbf{Node structure.} The structure of a Patricia trie node is what one would expect:

\begin{verbatim}
struct PTrie<C,K,V>::Node {

24
\end{verbatim}
// Pointers to other nodes, where applicable (0 otherwise)
Node parent;
Node pred;
Node succ;

// Character along branch from parent to this node
C branch_char

// Branches to child trie nodes
typedef Map<C, Node*, less<C>> branch_map;
branch_map branches;

// Pointer to key/value for key nodes
ptr<K> key;
ptr<V> value;
}

The precise container used for the branch map has only one restriction: that iteration through it be done from least character value to greatest character value; therefore, hash-based maps are not applicable, while all balanced, ordered tree-based maps are.

3.2.3 String B-Tree

The String B-Tree is an implementation of the simpler of the two algorithms in [5], which solves only the prefix search problem, not the substring search problem. It is implemented atop the block database, with each operation type (allocate, insert, remove, find, and iterate) a request and each node type (superblock, index node, and data node) an element. The only functionality discussed in the design section not yet implemented at the time of this writing is sequential access.

Superblock. The superblock is the first block in the database. Currently, it contains only the location of the root index node.

Index nodes. An index node is simply the binary representation of a Patricia trie, along with pointers to preceding and succeeding index nodes on the same level of the tree. As stated above, internally a Patricia trie is represented as a complex data structure making use of splay tree-based maps; thus, when an index node is loaded, its binary representation is converted to this internal representation, and back again when it is saved.

The string comparison callback passed to the Patricia trie operations uses the partial-element loading ability of the block database, loading one block at a time to perform string comparisons on traversals.

Data nodes. A data node is a series of blocks containing a windowed key (a 32-bit unsigned integer i in network byte order followed by i key bytes) followed by the associated value. The
data nodes are placed in no particular order in the database: there is no locality of data nodes based on key value.

**Operations.** The interface of the String B-Tree is slightly different from that of other databases:

```cpp
class SBTree : protected BDB {
    struct value_loc_t {
        blockd start;
        blockct count;
    };

typedef callback<void, ptr<const Cbuf> >::ref found_cb;
typedef callback<void, value_loc_t, ptr<Cbuf> >::ref post_alloc_cb;

    // Initializes the database and executes a user-defined 'main'
    // function once the database is ready
    void start (generic_cb main);

    // Called before insert to allocate space for some new data
    void alloc (const_key_t key,
                size_t free_space,
                post_alloc_cb post_request);

    // Inserts a previously allocated key/value into the database
    void insert (const_key_t key,
                 value_loc_t insert_loc,
                 value_t value,
                 generic_cb post_request);

    // Removes a key and its associated value from the database
    void remove (const_key_t key,
                 generic_cb post_request);

    // Searches for a key and executes the appropriate callback
    void find (const_key_t key,
               found_cb found,
               generic_cb not_found,
               generic_cb post_request);
```
// Signals the String B-Tree that it should shut down once all
// requests have completed
void
    terminate (generic_cb post_shutdown);
};

The primary difference between this setup and those of other databases is that the data
node allocation and insertion operations are logically separate: this is necessary since the
user cannot know a priori how much space is required for the value and key, which are both
stored together. Therefore, upon allocation, the user specifies a minimum amount of free
space he desires in the data node, but the alloc function may provide much more: this will
be evident by the size of the Cbuf passed to the post_request callback, all of which may be
filled as desired.

Once the user has finished modifying the value buffer, he passes it to insert which
performs the actual insertion into the tree. It is important that an alloc actually be followed
by an insert, or those allocated blocks will be lost.

3.2.4 Support data structures

Several data structures support the operation of the above components. These are described
here briefly.6

Splay tree. Based on the algorithm in Sleator and Tarjan’s original paper,[8] the class
SplayTree is a container with parameters for the key, value, and an asymmetric and transitive
key comparison relation class. SplayTree provides the expected basic functionality: insertion
(both exclusive values for a key and multiple values associated with the same key), removal,
search, and sequential iteration.

Two containers are derived from the SplayTree: Map, which enforces a unique value for
each key; and Set, which has no value for a particular key.

The key and value can each be any scalar or aggregate type, but the comparison relation
class is worth noting. It is a template class on some type T with only one method—the
application method, operator(const T &a, const T &b)—returning true iff a < b.

Hash table. A typical hash table with adaptive expansion, HashTable is a template class on
key and value parameters that, on n buckets, hashes keys modulo n according to the unsigned
result of a call to hash_primary(key). A future improvement will be to parameterize the
hash function as was done for the comparison relation in SplayTree.

6The C++ standard template library containers were not used primarily because they cause significant
global namespace pollution on some primitive C++ compilers—including any version of gcc before 2.9—that
interferes with the compilation of the AIO library.
4 Evaluation

To demonstrate the performance of this database against a well-known baseline, in some tests I have chosen to compare it to the freely-available Sleepycat implementation of the standard Berkeley database in B-Tree mode. The version of Sleepycat used for the tests was 3.1.14.

There were two machine configurations used:

- **Small.** Intel Celeron 233 with a 6GB 5400 RPM ATAPI hard drive and 64 MB of 60ns SIMMs running Linux 2.2.14
- **Large.** AMD K6-2 450 with a 10GB 5400 RPM ATAPI hard drive and 256 MB of PC-100 DIMMs running Linux 2.2.16

Wallclock time in the below results was timed from the beginning of the test—generally, opening the database—until all of the test data was written to disk using the sync operations provided by the database and operating system. In particular, the measured time does not include that spent scanning the Sleepycat database to produce statistics.

In each test comparing the String B-Tree to Sleepycat, both databases use the same specified blocksize.

Finally, to ensure consistent results from back-to-back tests, flushb (a Linux ext2 filesystem command to flush the read buffers from the VFS buffer cache) was called immediately before a test began.

4.1 Benefit from asynchronous I/O

One of our primary goals for this project was to determine the benefit derived from asynchronous I/O. In this test on the String B-Tree only, a database with a blocksize of 8192 bytes and already populated with 30,000 16000-byte keys was hit with 100,000 random accesses (in the proportion of 5% inserts, 5% removes, and 90% searches). Figure 6 graphs the number of concurrent requests versus the wallclock time required for the random accesses to complete on the large machine.

When there is only one request, all I/O operations are essentially serialized: only one element fetch at a time can be initiated by a request. When multiple requests run concurrently,

- the AIO subsystem has greater choice in scheduling the I/O operations;
- requests desiring data already in the buffer cache do not have to wait for a costly disk access to complete, reducing latency; and
- requests not blocked on I/O and which consume non-trivial CPU time can run while disk I/O is occurring, essentially eliminating disk idle time

This is clearly evident from figure 6, in which the total running time drops by 23% from the single-request scenario to the one in which there is an optimal number of concurrent requests (8-9). Concurrency beyond this point appears to add only overhead and not additional performance, but of course, this most likely depends on the application.
4.2 Large keys

Another of our goals for this project was to demonstrate that this algorithm would perform comparatively well on key distributions composed of keys large relative to the blocksize.

4.2.1 Varying keysize

For each of several keysizes between 500 and 10,000 bytes, we populate a database of 1024-byte blocks\(^7\) with 30,000 keys and then perform 10,000 searches for random keys in the database. The String B-Tree had eight concurrent requests at any one time. Tests were performed on the small machine, to minimize the impact of the buffer cache.

Figure 7 graphs the keysize versus the running time of the insert operations, while figure 8 does the same for the search operations. In both tests, the String B-Tree maintains a sizeable advantage over Sleepycat over the entire range of tested keysizes, despite the absence of PT-diff and PT-patch-insert. This can likely be attributed to Sleepycat’s frequent access of key overflow blocks in the search test (which is a series of searches for keys we know are in the database) and to fewer index nodes in the String B-Tree, leading both to more effective use of the buffer cache and to fewer levels in the tree.

While the String B-Tree also needs to load a complete key when the search key is in the database, it needs to do this only once: Sleepycat may need to do this several times on a downward traversal, a problem that is exacerbated by the low branching factor on trees with

---

\(^7\)1024-byte blocks were used due to limitations on the test machine: smaller blocks mean smaller keys are large relative to the block size, allowing us to fit more keys into a database of the same size.
4.2.2 Varying population

For database sizes ranging from 500 to 50,000 4000-byte keys, we wished to determine the relative performance of the String B-Tree and Sleepycat. There were two stages to this test: first, the total running time of the inserts was clocked; and second, the running time for 100,000 random accesses was clocked. The databases each were configured to use 1024-byte blocks, and the tests were run on the small machine to minimize the impact of the buffer cache.

Figure 9 graphs the database size versus the wallclock time for the insertions to complete. Sleepycat has performance similar to the String B-Tree at first, but slows down considerably at around 10,000 keys and continues to fare poorly thereafter. This is likely due to the overhead of writing multiple key overflow blocks, and to the operating system’s inability to cache the large number of index nodes resulting from a low branching factor: the String B-Tree has fewer index nodes—which presumably are all cached—and must therefore load at most one block per level.

Figure 10 graphs the database size versus the wallclock time for the searches to complete. As for inserts, the performance of Sleepycat is better than or similar to the String B-Tree until about 10,000 keys, when it becomes markedly worse. This is probably due to accessing key overflow blocks on finds and removes, and to the large number of index nodes, as above.
4.3 Discussion

The preceding tests, though sufficient to demonstrate that the String B-Tree satisfies the desired properties, are not the whole story: although the String B-Tree succeeds admirably in certain cases, it may not yet be truly "generic." Some implementation issues and a single primary deficiency of the String B-Tree design related to this application add up to the need for some improvements.

4.3.1 Small databases

Although the String B-Tree seems to perform well on databases with large numbers of large keys, it fares worse on smaller databases. A test on the large machine involving 100,000 random accesses on an already-existing database of 5,000 keys was performed to gauge the performance of the String B-Tree (at eight concurrent requests) versus Sleepycat on small databases. The blocksize for each database was set at 4096 bytes, and the test varied the keysize. Figure 11 shows the running time versus the keysize. The results here indicate that when the database has few keys, Sleepycat outperforms the String B-Tree at any particular keysize.

Profiling suggests that much time is wasted in the support data structures: in the splay trees used to store Patricia trie node branches, in the conversion between the internal Patricia trie representation and the binary representation, and in keeping reference counts in many data structures that change frequently.

The String B-Tree’s AIO advantage may not manifest itself on such small databases, which makes CPU usage a much greater portion of the overall running time. This suggests
that reducing CPU usage would make a database which is already very I/O efficient even faster.

4.3.2 Block loads

A test similar to the one on varying keysize in section 4.2 was performed, except that some searches were performed for keys not in the database: the impact of this on the String B-Tree search is that at most one block needs to be loaded for each level of the tree, regardless of how long the keys are. We performed 10,000 random accesses on an existing database of 30,000 keys of a fixed size. As in the other tests, we used eight concurrent requests.

To provide a "worst-case" scenario for this test, the blocksize was set at 512 bytes, forcing a low branching factor and, consequently, adding many levels to the tree over a larger blocksize.

The results are graphed in figure 12. Although its wallclock performance is not indicative of key-independence, the constancy in the number of block accesses versus varying keysize suggests this database provides optimal performance for this particular input parameter. As suggested above, other factors—including but not limited to high CPU usage—may account for the difference. In this case, the increasing size of the database and the resulting greater latency in accessing a random block may account for the increase.

4.3.3 Small keys

The one main design deficiency related to this analysis of the String B-Tree is its failure to deal adequately with keys that are small relative to the blocksize. The problem is that on distributions of very small keys, two entire blocks must be loaded at every level of the tree: one for the index node itself, one for the key used in smart search.
Assume the branching factor of a String B-Tree is $B$. For a tree in which only one block needs to be loaded at each level for distributions with small keys (such as Sleepycat, which stores small keys entirely within the index nodes), the equivalent branching factor $b$ is given by

\[
\log_b m = 2 \log_B m \\
\Rightarrow \frac{\log m}{\log b} = 2 \frac{\log m}{\log B} \\
\Rightarrow \log_b b = 2 \log_2 B \\
\Rightarrow b = \sqrt{B}
\]

which is significantly smaller, as we would expect in a prefix-compressed tree. Thus, the large branching factor of the String B-Tree doesn’t help in databases with small keys, since other databases can have twice as many levels of index nodes and still require fewer disk accesses. The real win with the String B-Tree is for databases with large keys, for which designs like Sleepycat’s require key overflow blocks.

A possible solution to this problem is to design a hybrid tree: one in which individual index nodes can each take on a form best suited to its data: small keys would be governed by a prefix compressed index node, while large keys that would otherwise end up in overflow blocks could be governed by Patricia trie-based index nodes.

Another solution which is easier to implement would be to traverse the entire tree assuming the key exists. When the key is actually in the tree, no mistakes would be made, so we would naturally end up at the correct node; however, if the key isn’t in the tree, we could...
find this out by loading the associated key blocks one at a time in succession, backtracking to the point in the traversal at which we made a mistake, and continuing from there in a second search for the insertion point using the standard algorithm.

Denote by $|V|$ the length of the data node, which includes both the key and its associated value. Using this strategy we would load the following numbers of blocks in each case:

<table>
<thead>
<tr>
<th></th>
<th>1st search</th>
<th>comparison</th>
<th>2nd search</th>
<th>value fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>not in tree</td>
<td>$\log_B m$</td>
<td>$</td>
<td>P</td>
<td>/B + \log_B m$</td>
</tr>
<tr>
<td>in tree</td>
<td>$\log_B m$</td>
<td>$</td>
<td>P</td>
<td>/B + \log_B m$</td>
</tr>
</tbody>
</table>

In the case that the key is not in the tree, we wind up loading a possible $|P|/B + 3\log_B m$ blocks, worse than the $|P|/B + 2\log_B m$ from the standard algorithm; however, in the case that the key is in the tree, we load only $\log_B m + |V|/B$, the minimum number required to traverse the height of the tree and load the entire data node. Thus, when most of our searches are for keys in the database, it makes sense to use this strategy. Furthermore, when we are concerned only with the case in which a key exists—we.e., when we don't need to know the insertion position of a key not in the database—we do not perform any disk accesses over the standard algorithm in either case.

### 4.3.4 File system emulation

Some of the files in the /usr tree on a Debian 2.2 box were added to a database, and then random accesses (in the same percentages as in the uniform key tests) were performed. To account for the limited storage on the test machine, the size of a single file was capped at
300,000 bytes. A blocksize of 4096 bytes—a typical cluster size for a Linux machine—was used. As before, the String B-Tree had eight concurrent requests at any one time.

Figure 13 graphs random access time versus the number of files (chosen randomly from /usr) in the database. Although Sleepycat jumps out to an early lead, the String B-Tree performance seems to level-off after about 15,000 files. This is likely due to the small key inefficiency discussed above, and to the inability to fragment large values discussed below.

Figure 14 graphs database size versus the number of files in the database. Sleepycat automatically breaks up large values into smaller pieces, and can scatter them throughout the database; this String B-Tree implementation has no such mechanism, and therefore wastes more space as a result of the binary buddy block allocation algorithm: a key/value combination of $2^{16} + 1$ bytes would actually take up $2^{17}$ bytes, nearly double the space actually needed. (This extra data also needs to be loaded at the end of a search, further burdening an already busy disk.)

5 Future work

In addition to the aforementioned performance improvement suggestions, there are several possibilities for future implementation work:

- **Reduce complexity of underlying data structures.** The inability of the String B-Tree to perform acceptably relative to Sleepycat on small databases and small keysizes is not necessarily due to the String B-Tree algorithm itself, but may be caused by relatively high CPU usage. Simplifying the support data structures and the complexity of the Patricia trie implementation has the potential to remove this roadblock.

Reduction the time spent saving a copy of each modified element would provide additional performance gain, since each element must be copied in its entirety in case some request is backed out. A log-structured approach to modifying elements may provide the desired time relief, with the added benefit of crash recovery.

- **Implement PT-diff and PT-patch-insert.** Insertion and removal times are abyssmal when small blocksizes are combined with large keysizes, because the String B-Tree insert and remove operations may require one key load in the case of a parent key replacement or two full key loads per level in the case of a split. Adapting these procedures to the lack of a termination character and implementing them properly should greatly improve the performance of large key inserts and rmoves.

- **Improve the element cache.** Abominable performance caused by the rampant conversion between the binary and internal representations of an element motivated the rapid development of an internal element cache. It is possible that the parameters on this cache can be tuned to provide better performance; even more likely is that a better element cache altogether would dramatically reduce the number of binary/internal conversions. The average element cache hit-rate was only $\approx 80\%$ on all tests performed.

- **Use a more intelligent transaction system.** Right now, requests are limited to "all-or-nothing" atomic operation: either complete the request without blocking, or
restore everything and start over. Fine-grained locking and resultingly fewer restarts may lower CPU usage, despite the added complexity of deadlock detection.

- **Allow for a variable maximum branching factor.** Currently, the branching factor of an index node is fixed at the pessimal number of strings per Patricia trie fitting in a single block in binary form; this situation occurs only when the Patricia trie is a complete binary tree with an extra node above the root.

  This restriction is caused by the dichotomy between the insert and remove code: under a system with a variable split threshold, a String B-Tree remove operation can cause some nodes to increase in size. Currently, there is no way to split a node in the remove operation. Joining the insert and remove code would trivially allow for variability in the split threshold.

- **Provide an automatic value-fragmenting capability.** This will improve space efficiency in databases with large values.

**Acknowledgements**

I wish to acknowledge the assistance and thoughtfulness of my advisers Frans Kaashoek and David Mazieres, both in giving me the opportunity to participate in a large software project and in providing guidance along the way. I also wish to thank David Karger for the timely 6.854 project that resulted in the study of the String B-Tree algorithm.

**References**


