Stratified Indexes – A Method for Creating Balanced Search Structures

Alon Itai[*] Moshe Shadmon[†]

May 22, 2000

1 Introduction

Data bases comprise of huge amounts of data whose size is far too large for volatile internal memory but must be kept on disks. An *access method* is a data structure used to to store and retrieve data from disks. The choice of the access method is greatly influenced by the characteristics of disks: disk storage is partitioned into contiguous fixed sized blocks, each disk access stores or retrieves an entire block. Since disk access is slow in comparison with CPU, the time of an algorithm depends on the number of disk accesses. The task of the implementor is to devise access methods that minimize the number of disk accesses.

Currently, two access methods are wildly used [9]:

- Direct access, i.e., hash tables,
- Multi-way trees, i.e., B$^+$-trees.

Ideally, hashing enables one to retrieve a data item within a single disk access. However, in practice because of overflow of buckets, and the need to extend the hash table [2] the number of disk accesses is larger. Moreover, hashing does not provide an efficient method to sequentially processing the file by increasing key. Since sequential processing is essential for answering many types of queries, (sub-range queries, joins, etc.) hash tables are not the method of choice for most databases.

B$^+$-trees are by far the most popular access method. In practice, they provide accessing an item within 2-3 disk accesses. In addition they allow to efficiently process the file sequentially by the primary key and provide reasonable performance for sequential processing by a secondary key. However, we need to keep an index (separate B$^+$-tree) for each key and each index has to be updated with each update of the database. Moreover, a node with k children, must contain $k-1$ keys. For long keys that arise in many application, most of the storage of the node is dedicated to keys. Given that the size of the block is fixed, when the

[*]Address: Computer Science Department, Technion, Haifa, Israel. email: itai@cs.technion.ac.il

[†]Ori Software, Tel Aviv

keys are long, the maximum degree of a node is small, thus the height of the tree is large, requiring more disk accesses to retrieve an item.

Tries [6] can be used instead of search trees to store and retrieve data. However, they have two drawbacks that lead to a lack of efficiency:

- They are wasteful in space,
- they lead to unbalanced structures: i.e., accessing some data might require passing through a long chain of blocks.

In our proposal we suggest a new way to implement tries that overcome both these problems.

Traditional implementations of tries are discussed in Section 2. In Section 3 we introduce the stratified indexing scheme for tries. In section 4 we show how PAATRICIA tries save space by keeping only part of each key. We then create a stratified index over the PATRICIA tries. However, applying this savings to might lead us to read the wrong block, however, such errors are immediately detected, and the correct block is read. In Section 5 we compare our data structure with the String B-tree proposed by Ferragina and Grossi [4, 3]. Finally in Section 6 we compare the performance of the new access method to B$^+$-trees.

2 Tries

Let Σ be a finite set of *characters* (to be referred to as the *alphabet*), a *string over* Σ is a finite sequence of characters, and Σ^* denotes the set of all such strings.

Let T be a tree such that every edge has a label $\ell \in \Sigma$ and the labels of all edges that emanate from a node are distinct. For each vertex $v \in T$ there is a unique path $P(v)$ leading from the root to v . Let the string $S(v)$ denote the sequence of labels of the edges of $P(v)$. Let $W(T) = \{S(v) : v \text{ is a leaf of } T\}$. For a set W of strings, if $W = W(T)$ then T is a *trie* for W .

Each trie node is consists of several pointer field. A trie can be very wasteful in space since there might be a long path of vertices each having a single child (Figure 1 (a)). To overcome this problem, such paths are condensed to a single edge (Figure 1 (b)). If $(v_0, \ldots, v_m)$ is such a path with edge labels $(\ell_1, \ldots, \ell_m)$, we replace the path by a single edge (v_0, v_m) with label ℓ_1, and keep a pointer to the string $\ell_2 \cdots \ell_m$ at v_m . This representation requires less space since we need a single pointer for the entire path, instead a pointer for each node of the path. Note that since in the resultant *condensed trie* every internal node has at least two children, there are fewer internal nodes than leaves. Hence, a condensed trie with n keys has at most $2n - 1$ nodes.

A more serious problem is that in many cases the trie is not balanced, even in the absence of long paths (Figure 1 (c)).

The manner in which nodes are implemented is also an important issue. A straightforward implementation calls for nodes each containing an array of $|\Sigma|$ pointers, where the i-th pointer connects v to a child v' such that the edge (v, v') has label $i \in \Sigma$. This implementation wastes space if most vertices have much less than $|\Sigma|$ children, since we have to allocate a large node,

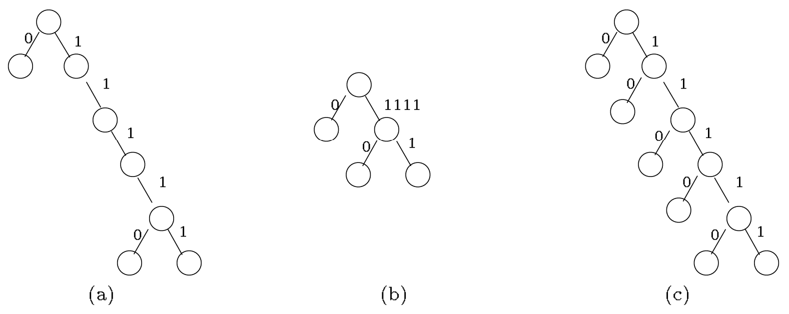


Figure 1: (a) A trie with a long path (b) Condensed trie (c) an unbalanced trie.

even when the actual node has only a small number of children. This representation is time efficient, since the search requires constant time at each node, and no comparisons are made.

Alternatively, for a node with d children one maintains a pointer to each child. Thus the search time depends on d, and since this implementation requires maintaining d labels. When $d \approx |\Sigma|$ it requires more space than the previous method.

Consider a stratified index over tries. It is a sequence of tries: On the lowest level we have a trie of the keys. The next levels provide an *index* to facilitate the search. Thus we overcome both the space and time deficiencies of the basic trie structure and provide an efficient implementation.

3 A stratified index over a trie

3.1 Description of the data structure

As explained in the Introduction, a trie can be very unbalanced, and hence might not be efficient. To allow efficient searching in the trie, we add an *index*, which provides an efficient search structure to the trie.

We utilize the fact that external storage, such as disks, stores data in pages and partition the trie into page sized *blocks*, such that the induced subgraph of each block is connected, i.e., a tree (See Fig. 2). Hence, each block B is a sub-trie rooted at $root(B)$. For each block B we define the key, $common(B)$, to be the string associated with $root(B)$, the root of B's sub-trie, i.e., $common(B) = W(root(B))$. The string $common(B)$ is a prefix of all the keys of the block B, and is thus called the *common prefix* of block B.

Let T^0 denote the trie. To facilitate searching T^0, we create another trie T^1 on the common prefixes $K^1 = \{common(B) \ : B \text{ is a block of } T^0\}$. (For the keys of the trie of Fig 2 the set of keys is $\{\Lambda, 10, 11, 1110, 1111\}$, see Fig 3.)

Every node $v^1 \in T^1$ for which $W(v^1) \in K^1$, has a pointer, $v^1.outptr$, to the block B of T^0, for which $common(B) = W(v^1)$. Since each leaf of T^1 corresponds to a block of T^0, each leaf has such a pointer, as well as some internal nodes.

This process is continued in the same fashion, creating additional tries $T^2, \ldots, T^h$, until creating a trie T^h that fits in a single block.

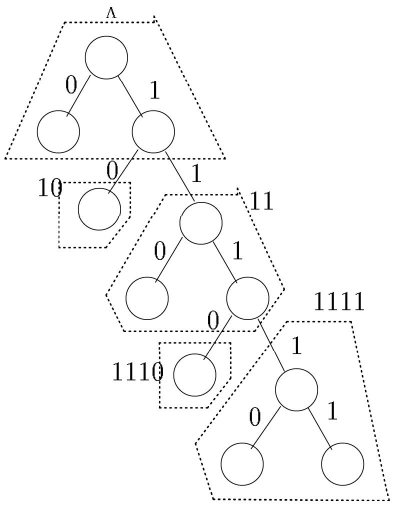


Figure 2: A partitioning a trie into blocks of size 3

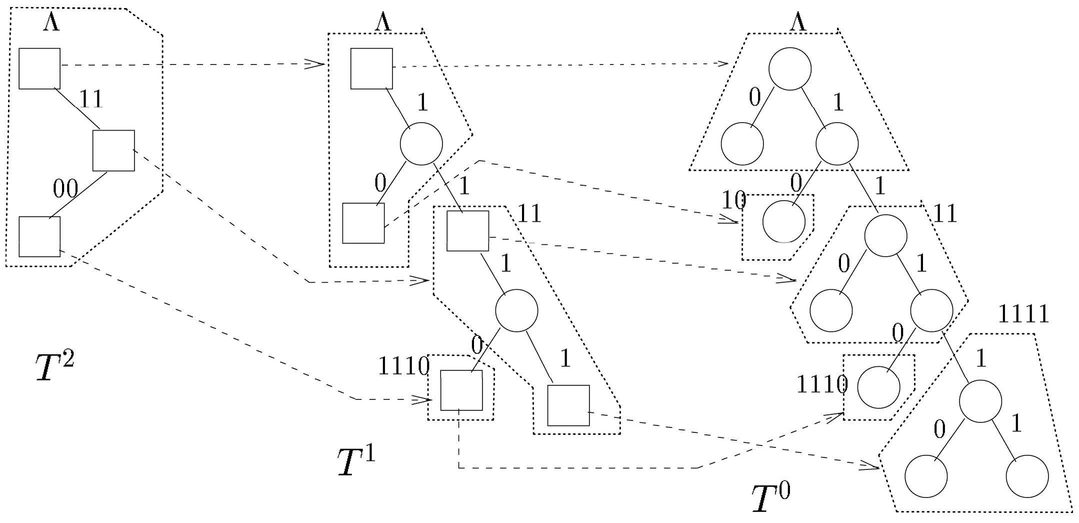


Figure 3: The tries T^0, T^1 and T^2.

Let n_i denote the number of out-pointers of T^i. Since each T^i is a compact trie, the number of vertices of T^i is at most $2n_i - 1$. If each block contains at least $\beta \geq 3$ nodes, $|T^{i+1}|$, the number of nodes of T^{i+1}, satisfies

$$\left|T^{i+1}\right| \leq \frac{2}{\beta}\left|T^i\right| .$$

Consequently, the number of levels h satisfies $h \leq \log_{\beta/2} n$.

The number of blocks for all the indices is

$$\sum_{i=0}^{h}\left|T^i\right| \leq \sum_{i=0}^{h}\left(\frac{2}{\beta}\right)^i\left|T^0\right| = \frac{1}{1-2/\beta}\left|T^0\right| = \frac{2n}{\beta-2} \approx \frac{2}{\beta}n .$$

Note that even storing only the pointers to the keys requires n/β blocks. Thus the index at most doubles the storage requirements. In Section 4 we will see how to reduce the amount of storage.

3.2 Searching

Introduce the following notation:

1. For a key x, let $||x||$ denote its length, $x[i]$ denote its i-th character and $x[i\,..\,j]$ the sequence $x[i]x[i+1]\cdots x[j]$. Thus, $x = x\,[0\,..\,||x||-1]$.
2. For $x, y \in \Sigma^*$, y is a *prefix* of x (denoted $y \sqsubset x$), if $y = x\,[0\,..\,||y||-1]$.
3. Let $x, y \in \Sigma^*$. Then $match(x,y)$ is the length of the longest common prefix of x and y, i.e., $x[0\,..\,j-1] = y[0\,..\,j-1]$ and if both $||x||$ and $||y|| > j$ then $x[j] \neq y[j]$.
4. Let $B^i(x)$ denote the block B of level i which maximizes $match(common(B), x)$.

Lemma 3.1 *Let $x \in \Sigma^*$, and let $v \in T^0$ be the deepest node such that $W(v) \sqsubseteq x$. Then*

1. *If x is a key of a record R in the database then $W(x) = v$ and v points to R.*
2. *$v \in B^0(x)$.*

Proof:

1. Immediate from the construction.
2. By definition $v \in T^0$. Suppose $v \in B' \neq B^0(x)$. By the definition of $B^0(x)$, $common(B')$ is a proper prefix of $common(B^0(x))$. Thus the path from $root(T^0)$ to v first passes through B' then to $B^0(x)$ and finally returns to B'. Thus the nodes of B' do not form a connected component, contrary to the way we partitioned T^0 into blocks.

□

Since T^i is a trie over $K^i = \{common(B)\ : B \text{ is a block of } T^{i-1}\}$, we get:

Corollary 3.2 *For $i \geq 1$, T^i has a unique node $v \in B^i(x)$ that points to $B^{i-1}(x)$.*

Thus the search proceeds from $B^h = B^h(x)$ level by level. At level i, we search the sub-trie of $B^i(x)$ to find a node u which maximizes $match(x, W(u))$, and points to a block B of level $i-1$ and continue to $B^{i-1}(x) = B$.

```
DISK_ADDRESS Search(key $x$, Database $D$){
    $B$ = root block of $D$;
    while $B$ is not a leaf do
      $B$ =Search_in_Block($x$, $B$);
    return Search_in_Leaf($x, B$);
}
```

Program Search

To search a leaf we use

```
DISK_ADDRESS Search_in_Leaf(key $x$, block $L$){
    let $v = root(L)$;
    $i = depth[v]$;
    while there exists an edge $(v, v')$ labeled $x[i]$ do {
      $v = v'$;
      $i = i + 1$;
    }
    if $v.outptr \neq NULL$ then /* $W(v) == x$ */
      return $v.outptr$; /* FOUND */
    return $NULL$;     /* NOT FOUND */
}
```

Program Search leaf

At that level the search ends at a vertex $v \in T^h$, such that $W(v)$, the sequence of labels of the edges of the path from $root(T^h)$ to x, is equal to $x[0 .. ||W(v)|| - 1]$ and v has no child labeled $x\,[||W(v)||]$. We climb up the tree, until finding an ancestor u of v such that $u.outptr \neq NULL$. (Possibly $u = v$.) Let $B^{h-1} = u.outptr$.

```
DISK_ADDRESS Search_in_Block(key $x$, Block $B$){
    let $v = root(L)$;
    $i = depth[v]$;
    while there exits an edge $(v, v')$ labeled $x[i]$ do {
      $v = v'$;
      $i = i + 1$;
    } /* $v$ is the lowest vertex of $B.trie$ which satisfies $W(v) \sqsubseteq x$ */
    while $v.outptr == NULL$ do
      $v = parent(v)$;
      return $v.outptr$;
}
```

Program Search block

Corollary 3.3 *1. If x is in the database the search procedure above finds it.*

2. The search requires exactly one block per level.

3.3 Insertion

To insert a record R with key x, we add R to the database, and add x to T^0. To find the block to which to add x, we a conduct a search for x, passing through blocks $B^h(x), B^{h-1}(x), \ldots, B^0(x)$. Let $k = common(B^0(x))$. We add the remainder of x, i.e., $x[||k|| \,..\, ||x|| - 1]$, to the sub-trie of $B^0(x)$. Let v be the last node added, i.e., $W(v) = x$. Then we let v point to R.

```
Insert_to_Block(key $x$, Block $B$, Record_or_Block $R$){
    let $v = root(B)$;
    $i = ||common(B)||$;
    while there exits an edge $(v, v')$ labeled $x[i]$ do {
      $v = v'$;
      $i = i + 1$;
    }
    /* $v$ is the lowest vertex of $B.trie$ which satisfies $W(v) \sqsubseteq x$ */
    while $i \leq ||x||$ do {
      let $v'$ be a new node;
      add edge $(v, v')$ labeled $x[i]$;
      $v = v'$;
      $i = i + 1$;
    }
    $v.outpr = R$;
}
```

Program Insert to block

If block $B^0(x)$ *overflows*, we *split* the block to $B^0(x)$ and B' (see Section 3.4) and add $common(B')$ to $B^1(x)$. This process might continue up-to the root. If $B^h(x)$ splits, we add a new level, $h + 1$, with a single block $B^{h+1}(x)$ which points to $B^h(x)$ and to the new block of level h.

```
Insert(record $R$, Database $D$){
    Add $R$ to the database at location $v$;
    let $x = R.key$;
    Search for $x$ in $D$,
      suppose the search went through blocks $B^h(x), \ldots, B^0(x)$.
    $i = 0$;
    $y = x$;
    $A$ = address of $R$;
    while $i \leq h$ do {
      Insert_to_Block($y$, $B^i(x)$, $A$);
      if block $B^i$ did not overflow then
        return ;
      Split block $B^i(x)$ to get a new block $B'$;
      $y = common(B')$;
      $i = i + 1$;
      $A$ = address of $B'$;
    }
    Add a new block $B^{h+1}$ whose trie consists of $common(B^h)$ and $common(B')$.
    The appropriate nodes point to $B^h$ and $B'$. }
```

Program Insert

3.4 Splitting

When adding keys to the trie of a block B, the number of nodes of the trie might exceed the capacity of the block. In this case, a subtree rooted at some node $v \in T(v)$ is moved to a new block B'. In order to achieve logarithmic search time, we would want that each of the blocks B and B' hold at least half of the original nodes of B. However, this is not always possible: Let $0 < \alpha < 1$. Consider, for example, a block of size $|B| < |\Sigma|$. If $T(B)$ consists of a single node with $|\Sigma|$ children, there is no vertex v such that $\alpha \leq |T_v| / |B| \leq 1 - \alpha$. However, if $|\Sigma| = 2$, then there always exists a good partition.

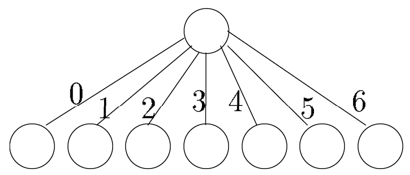
Lemma 3.4 *Let T be a binary tree of at least three nodes. Then there exists a node $v \in T$ such that*

$$\frac{1}{3} \leq \frac{|T_v|}{|T|} \leq \frac{2}{3} .$$

Thus if we want to ensure a good split, we embed the Σ-trie in a binary trie, i.e., we replace each character $c \in \Sigma$ by a binary word of length $\lceil \log |\Sigma| \rceil$. Thus a key of k characters is replaced by a binary key of length $k \lceil \log |\Sigma| \rceil$. We construct the binary trie for which we are guaranteed that a good split node exits. See Figure 4.

Implementation note

In practice we need not construct the binary trie explicitly, instead, we construct a binary trie only for the parent of the node v of Lemma 3.4.



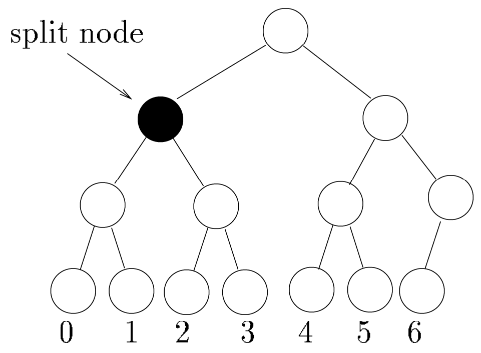


Figure 4: (a) A trie with no split node (b) A binary trie

4 A stratified index over a PATRICIA trie

In other tree data structures, such as B$^+$-trees, the keys require a large amount of space—a large key might be replicated at each level of the tree. In contrast, in a trie, each key appears at most once. Sometimes the size of the trie is smaller than the data, e.g. the keys $\{\underbrace{0\cdots0}_{m}0, \underbrace{0\cdots0}_{m}1, \ldots, \underbrace{0\cdots0}_{m}9\}$, require one path of length m that ends with a vertex with 10 children.

However, we go much further in reducing space by using *PATRICIA tries* (PT). To distinguish the PT from the tries discussed above will will refer to the latter as *full tries.*

4.1 PATRICIA tries

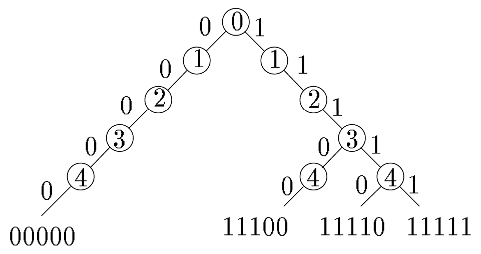
We start with a full trie for the keys $K_1, \ldots, K_n$ over the alphabet Σ. Thus n vertices of the full trie represent the keys. Each full trie vertex contains its *depth*—distance from the root—and $|\Sigma|$ pointers labeled $0, \ldots, |\Sigma| - 1$. Each such pointer can be NULL or be an outgoing edge leading to a child. The search path to a string follows the edge from the root labeled with the first character of the string. The path from the root to a full trie vertex v defines a string $W(v)$. Let u be an ancestor of v. Then the following *trie-invariant* holds:

$$W(u) = W(v)[0..\mathit{depth}(u)] \ . \tag{1}$$

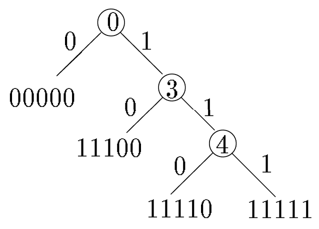
Thus, all data keys for which $W(u)$ is a prefix are reachable from descendents of u.

A PT is an index into the data, not the data itself. Thus we need to separately represent the keys $K_1, \ldots, K_n$ as strings in main memory or the disk. The PT is obtained from a full trie on the keys by repeatedly replacing a vertex v with only one child and its incident edges by an edge leading from v's parent to its only child. Consequently, all non-leaf vertices of the resultant tree (except perhaps the root) have at least two children. (See Figure 5.)

Each vertex v that corresponds to a key K has a pointer (called *out_pointer*) that points to the record with that key. Let v be a node of a PT at depth d, then all the keys descending from v agree on the first d characters. Denote this string $W(v)$. (Actually, $W(v)$ of a PT is equal to that of the corresponding full trie vertex.) The PT also maintains the trie



(a) A trie



(b) A Patricia trie (PT)

Figure 5: A full trie and PT on the keys 00000, 11100, 11110, 11111.

invariant—Eq. (1)—and as in the full trie, the path from u to v exits u by an edge labeled $W(v)[i]$.

4.1.1 Search

To search for a key P, one first descends the PT according to the *blind search* procedure described below. If P belongs to the PT, the blind search reaches a vertex v for which $W(v) = P$. Otherwise, the search reaches some vertex v for which $W(v) \neq P$. Thus to distinguish between a successful and unsuccessful search, check whether $W(v) = P$.

Blind search

To search for a pattern P, start from the root of the PT and follow the labels according to the depths. I.e., when arriving at vertex v, follow the edge labeled $P[depth[v]]$ if it exists. If no such edge exists, the search terminates.

```
node blind_search(key $P$, PT $T$){
        $v$ = root of $T$;
        while $depth[v] \leq ||P||$ do {
            $i = depth[v]$;
            if $v$ has no outgoing edge labeled $P[i]$ then
                return $v$;
            $v$ = child of $v$ pointed to by the edge labeled $P[i]$;
        return $v$;
        }
}
```

Program Blind Search

For example, to search the key $P = 11100$ in the PT of Figure 5. start from the root (depth 0) and follow the edge labeled $1 = P[0]$ to reach the vertex at depth 3. Since $P[3] = 0$ we take the edge labeled 0, to reach the desired key. If P does not belong to the PT, the search might lead us to a key $K \neq P$, so one has to check if $K = P$. For example, when searching the key $P = 10010$, one follows the edge labeled $1 = P[0]$ to reach a vertex at depth 3, then the edge labeled $1 = P[3]$ to reach a vertex at depth 4, and finally the edge labeled $0 = P[4]$ to reach key $K = 11110$.

```
node PT_search(key $P$, PT $T$){
    $v$ =blind_search($P$, $T$);
    **if** $Key(v) = P$ **then**
        **return** "SUCCESS at $v$";
    **else return** "FAIL at $v$";
}
```

Program PT Search

4.1.2 Insertion

To insert a record R with a new key P, we first find its PT-parent, i.e., a PT-vertex y that points to the new record. Let u be the deepest vertex for which $Key(u) \sqsubseteq P$, if $P\,[depth[u]] = NULL$ then $y = u$. Otherwise we need to add a new vertex y between u and its child in direction $P\,[depth[u]]$.

```
PT-node PT-parent(key $P$, PT $T$){
    $v = blind\_search(P, T)$;
    **if** $Key(v) \sqsubseteq P$ **then return** $v$;
    **let** $i$ = the first position where $P[i] \neq Key(v)[i]$;
    $u = v$;
    **while** $depth(u) > i$ **do**
        $u = parent(u)$;
    **let** $y = u \to out\_edge\,[P[depth[u]]]$;
    **if** $y \neq NULL$ **then** {
        $y$ = **new** vertex;
        $depth[y] = i$;
        $y \to out\_edge\,[Key(v)[i]] = u \to out\_edge\,[P[depth[u]]]$;
        $u \to out\_edge\,[P[depth[u]]] = y$;
    }
    **return** $y$;
}
```

Program PT-parent

Once the parent is found, insertion is straightforward.

```
PT-insert(key $P$, PT $T$){
    $v$ = PT-parent($P$, $T$);
    **if** $W(v) = P$ **then return** ;
    Allocate $P$ in block $R$.
    Add an edge from $v$ to $R$, labeled $P\,[depth[v]]$;
    **return** ;
}
```

Program PT-insert

4.2 Implementing block indexes as PT

We saw that a stratified index over a full trie was somewhat wasteful in space. To overcome this problem, we replace the full tries $T^0, \ldots, T^h$ by PATRICIA tries.

The out-pointers of T^0 point to blocks that contain the records. Let ${B^i}_0, \ldots {B^i}_m$ be the blocks of T^i. Then the keys of T^{i+1} are $common({B^i}_0), \ldots common({B^i}_m)$, and the out-pointers of T^{i+1} lead to the blocks of T^i. To perform the searches efficiently, each block contains the common prefix of all its records. (Each record maintains its full key.) See Figure 6 for a two-level index ($h = 1$).

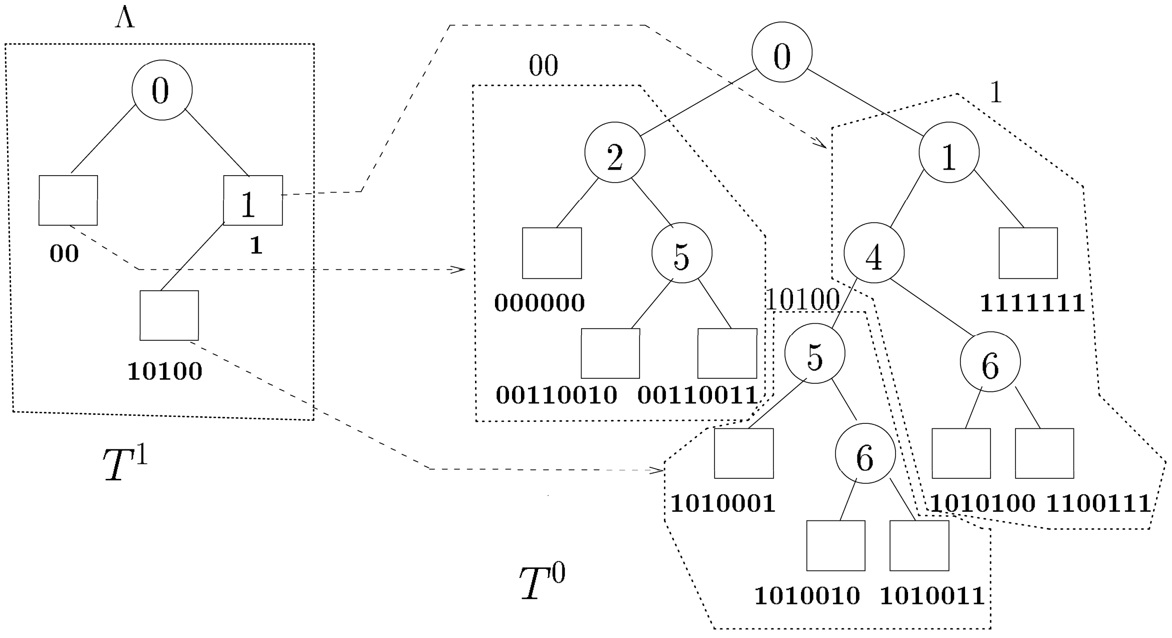


Figure 6:

Since a block does not contain the keys of the blocks it points to, the stratified index over PT is space efficient. This leads to better space utilization of each level, i.e., each block contains less key information and more outgoing pointers. Thus using PT decreases the number of levels. However, this savings comes at a price. Since the blind-search procedure for key P might lead to a vertex v such that $Key(v) \neq P$, the search might involve reading superfluous blocks. We will address these problems in the next section.

4.3 Search

In principle, we would like to perform a search of the stratified indexes over PT the same way we searched stratified indexes with full keys. However, since T^{i+1}, the PT of level $i+1$, consists of the *common prefixes* of the blocks of T^i, not the keys of T^i, the PT-search in block $B^{i+1}(x)$ might not lead to $B^i(x)$. The PT-search is guaranteed to succeed only on keys in the PT. and if $x \neq common(B^i(x))$, the search might result in a failure.

Example 4.1: Consider a database consisting of the keys 000000, 00110010, 00110011, 1010001, 1010010, 1010011, 1010100, 110111, 1111111 (Figure 6. Searching for $x = 1010100$ in T^1 leads to the block whose common prefix is 10100, instead of to the block with common prefix 1. □

Therefore, for sparse keys we replace the procedure Search_in_Block by Search_sparse_Block below. This is essentially the PT-parent procedure that was used for insertion into a PT.

Searching a key P in a non-leaf block B^i of level i should lead to a block B^{i-1} of level $i-1$ such that $common(B^{i-1}) = common(root(B^{i-1})) \sqsubseteq P$. Starting from $root(B)$ at the general step we are at vertex v of the PT, and move to its child u such that (v, u) is labeled $P[depth[v]]$. If there is no such edge, we move up the PT until we find an ancestor v' such that $v'.outptr \neq NULL$. (It is possible that $v = v'$.)

We now move to block $B' = v'.outptr$ and check whether $common(B') \sqsubseteq P$. If the answer is positive, then B' is indeed the block from which we should continue the search. Otherwise, let j be the first index of the mismatch, i.e., $common(B')[0..j-1] = P[0..j-1]$ and $common(B')[j] \neq P[j]$. We return to block B to the vertex v' from which we moved to B' and climb up the path from $root(B)$ to v' to find the lowest ancestor v'' of v' whose depth is less than j and has an pointer to a block B'' of level $i-1$. (See Figure 7).

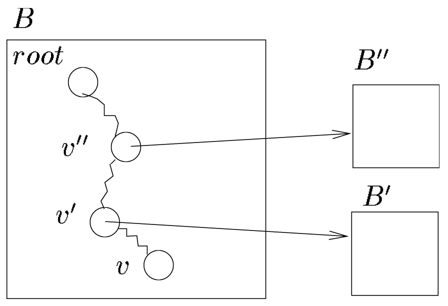


Figure 7:

We summarize the search procedure by Program Search_Sparse_Block.

```
Search_Sparse_Block(key $P$, block $B$){
    $v$ = blind_search($P$, $root(B)$);
    $v' = v$;
    while $v'.outptr == NULL$ do
        $v' = v'.parent$;
    if $key(v'.outptr) \sqsubseteq P$ then
        return $v'.outptr$;
    $j = match(v'.outptr[j'], P)$;
    $v'' = parent(v')$;
    while $depth[v''] > j$ or $v''.outptr \neq NULL$ do
        $v'' = parent(v'')$;
    if $v''.outptr \neq NULL$ then
        return $v''.outptr$;
    return FAIL;
}
```

Program Search sparse block

Lemma 4.2 *Let $\hat{B}^i(x)$ be the block of level i chosen by the procedure Search(P, D) Then $\hat{B}^i(x) = B^i(P)$.*

Proof: We show by induction on j that $\hat{B}^{h-j}(x) = B^{h-j}(P)$.
Base $j = 0$:
Level h consists of a single block B^h. The requirements hold since $common(B^h) = \Lambda \sqsubseteq P$.
Induction step we assume that $\hat{B}^{h-j}(x) = B^{h-j}$ and show that the $\hat{B}^{h-j-1}(x) = B^{h-j-1}$.
By definition B^{h-j} contains a vertex u that points to B^{h-j-1}. Let v be the deepest node of B reached by the search, v' the lowest ancestor of v (in the PT) that contains a pointer to a block of level $h-j-1$, and v'' the lowest ancestor of v' (in the PT) that contains a pointer to a block of level $h-j-1$ and in addition $key(v''.outptr) \sqsubseteq P$. $u \in B^{h-j}(P)$ is the node that points to $B^{h-j-1}(P)$. (These nodes need not be distinct.)

Claim 4.2.1 *u is an ancestor of v.*

Proof: Suppose the claim is false. Then there are two cases:
(1) u is a proper descendant of v: Since $key(u) \sqsubseteq P$, v has an edge labeled $P[depth[v]]$ leading to u. When the search reached v, we would have continued to that edge and not have stopped at v.
(2) u is neither an ancestor or a descendant of v: Let w be the deepest common ancestor of u and v. Since w is an ancestor of u and $key(u) \sqsubseteq P$, $key(w) \sqsubseteq P$. Moreover, the edge that leads from w to u is labeled $P\,[depth[w]]$, hence, the search would have gone to u and not to v. □

Claim 4.2.2 *u is an ancestor of v'.*

Proof: Suppose u were a proper descendant of v'. While climbing up the path $root(B)$ to v we checked for each node w whether $key(w.outptr) \sqsubseteq P$ and hence would have stopped at u. □

If $key(v') \sqsubseteq P$ then $v' = u$ and $\hat{B}^{h-j-1}(x) = v'.outptr = u.outptr = B^{h-j-1}(P)$.

Claim 4.2.3 *If $key(v') \not\sqsubseteq P$ then $u = v''$.*

Proof: $k = match(P, key(v')) < depth[v']$. Since u is an ancestor of v', $key(v'') \sqsubseteq P$, and by construction v'' is the lowest node whose outptr points to a block whose key is a prefix of P. In this case, $v'' = u$ and $\hat{B}^{h-j-1}(x) = v''.outptr = u.outptr = B^{h-j-1}(P)$. □

□

To search a key P in a leaf L for which $key(K) \sqsubseteq P$, we conduct a PT_ search in L's PT. If the PT_search fails, then P does not belong to the database.

Remark 4.3: To save space, the keys $common(B)$ will be maintained incrementally. Let $B^h, \ldots, B^0$ ($B^i \in T^i$) be the set of blocks that lead to block B^0, and $d^i = depth[root(B^i)]$. Then let $\Delta(B^i) = common(B^i)[d^{i-1} \mathinner{..} d^i - 1]$, the characters of $common(B^i)$ that do not appear in $common(B^{i-1}$. Thus $\Delta(B^h) = \Lambda$ and $common(B^i) = \Delta(B^{h-1}) \cdots \Delta(B^i)$. At each block B^i we compare the next characters of the key to $\Delta(B^i)$. In the record we can also just keep the increment.

Remark 4.4: At each level we might have to read two blocks $B' = v'.outptr$ and $B'' = v''.outptr$, only on one of which the search is continued. Thus at most $2h + 1$ blocks are examined. If the two topmost levels reside in memory, we need to read $2h - 1$ blocks.

4.4 Insertions

Insertions with sparse keys is similar to insertions with full keys. First, we find the block $B^0(P)$, insert P to it using procedure Insert_to_Sparse_Trie below. Then if the block overflows, split it, and insert the key of the new block to $B^1(P)$. Thus the insertion process might continue up to $B^h = B^h(P)$.

```
Insert_to_Sparse_Trie(key $P$, Sparse_Trie $S$){
    Add a new record $R$ whose key is $P$.
    Proceed as as search sparse block,
       to find the lowest node $u$ such that $key(u) \sqsubseteq P$.
    If $depth[u] < ||P||$ then
       Add the edge $(u, R)$ and label it $P[depth[u]]$.
    else {
       let $(u, u')$ be the edge labeled $P[depth[u]]$;
       let $j = match(P, key(u'))$;
       add a new node $u''$ of depth $j$;
       add the edge $(u, u'')$ and label it $P[depth[u]]$;
       add the edge $(u'', u')$ and label it $key(u')[j]$;
       add the edge $(u'', R)$ and label it $P[j]$;
    }
}
```

Program Insert to PT

4.5 Sequential processing

The sparse key implementation is well suited to processing the file sequentially by increasing key value. Consider first the full trie. Because of the nature of the full trie, when processing the leaves in inorder from left to right, the keys are encountered by increasing lexicographical order. The crucial point is that the left-to-right order of the leaves in the PT is identical to that of the full trie. Hence also in the PT implementation, processing the leaves from left to right implies that the file in processed by increasing order of the keys. Therefore, in sequentially processing the file, after exhausting all the links from the index block, it can be discarded, hence at every instance we need to keep only one index block of each level and each index block need be read only once.

5 String B-trees

Recently Ferragina and Grossi [4] proposed a similar data structure—the *String B-tree*. However, there are important differences: The space they require is at least twice as large as ours, and mainly, the time complexity to search a database of n records, and page size B is $O\left(\log_B{}^2 n\right)$. Moreover, the constant hidden in the "big O" is much larger than ours.

As in stratified indexes over PT, they construct an index, and each index node is organized as a PT. They start with a B-tree. However, each parent node contains *two* pointers to each of its children. One labeled by the minimum key reachable from the child node and the second labeled by the maximum key reachable from the child. Each internal node is organized as a PT on the maximum and minimum keys of its children. Thus String B-trees require twice as many pointers as stratified indexes over PT, incurring a waste of space.

A stratified indexes over PT reflects the structure of the trie, while String B-tree reflect a B-tree structure, i.e., keys with no common prefix might be reachable from the same (non-root) node, while, keys with a large common prefix are reachable from different nodes. Thus String B-trees do not maintain the common prefix of internal nodes.

Consequently, when searching the String B-tree the error might be detected only when reaching the records themselves. When an error occurs at a block at level i the correction procedure leads us to the correct block at level $i-1$. Since, a search might involve an error at each level, the search complexity for a database of n keys and page size B is $\theta\left(\log_B{}^2 n\right)$.

Each String B-tree pointer contains a pointer to a record the keys of these records might be needed to correct the errors of the blind-search. This incurs an additional overhead. See [5] for a complete counterexample.

6 Evaluation

The number of disk accesses depends on several factors:

N - the number of records in the file,

B - block size (in bytes),

K - key size (in bytes),

R - record size (in bytes)

P - pointer size,

P_B - average number of pointers in a block.

R_B - average number of nodes in a block.

α - average density of a block, the average number of nodes per block divided by the maximum number of nodes that fit in a block. nodes.

The number of blocks required by the records of the file is $N_0 = N/R_B$, this is typically more than the size of the file $N \times R$ divided by the block size, since not all blocks are necessarily full.

The level 1 index requires one pointer for each block of the file thus the number of blocks is $N_1 = N_0/P_B = N$. The next level requires $N_2 = N_1/P_B = N_1/P_B^2$ blocks, and level i index $N_i = N/(R_B P_B^i)$ blocks. Thus the number of levels is $h = \lceil \log_{P_B} N/R_B \rceil$.

We shall assume that levels h and $h-1$—the root and its children—reside permanently in main memory, thus to access a record, we need to pass through levels $0, \ldots, h-2$, i.e., $h-1$ levels.

In the worst case, each level accessed entails a miss, thus the number of block accesses is $2(h-1)$. However, experience indicates that this is very pessimistic, in reality the number of accesses is much closer to $h-1$. Moreover, when the file becomes larger, the keys become denser, and the path to a leaf contains more levels, hence the probability that a level is skipped is reduced, thereby reducing the probability of a miss.

The average number of pointers pointing out of a block, depends on the block size, the size of the trie node the density of the block, and β—the proportion of nodes that point to other blocks. Since in the PT each internal node has degree at least 2, the number of pointers to other blocks is greater or equal to the number of nodes. i.e., $\beta \geq 1$. Consequently,

$$P_B \geq \frac{\alpha B}{node\_size}$$

Suppose a pointer to a block requires 4 bytes, in addition with each pointer we need one more byte for the label, hence we need 5 bytes per pointer. In addition to the pointers, we need space for the nodes this space is independent of the number of children of the node. For a large file, the nodes near the root of the trie have many children, thus we may ignore the overhead needed for storing the node itself.

Under these hypotheses, a block of size B = 8KB can accommodate 8192/5 = 1638 pointers. For $\alpha = \ln 2 = 0.69$ we get that $P_B = 1140$ [1]. Even if we bring into account the space required for the nodes, $P_B \geq 1000$. Thus an index of height 3 can accommodate over ${P_B}^3 \geq 10^9$ records. Thus even for very large files no more than three index levels are needed. Since the first two levels reside in main memory, only one level of the index resides on the disk.

[1]This choice of α is customary, since it arises in the analysis of the insertion process of B-trees [1] and similar structures.

There are no disk misses at index levels that reside in main memory, and none for the records themselves, hence about only two disk accesses are required (one for the index and one for the record).

7 Other stratified indexes

We now wish to consider the stratified indexes over PT as a particular implementation of a more general construction. This construction allows to abstract several known data structures such as B-trees, and extendible hashing [2] as well as many new ones.

We start with a *partitioned* search structure—a data structure consisting of m blocks of data. Each block has a *representative key* from some domain $\mathcal{K}$. The representative keys suffice for search, i.e., given a key $K \in \mathcal{K}$ one can find the block that contains the record with K by searching the representative keys. More formally, there exists a *search function* $S : \mathcal{K} \times \mathcal{K}^m \mapsto 1, \ldots, m$, such that given a search key $K \in \mathcal{K}$ then $S(K, (K_1, ..., K_m))$ is the block of key K.

Example 7.1: The partitioned search structure is a sorted sequence of records partitioned into blocks of 3 records each. In Fig. 8 the representative keys are $R = (17, 32, 120, 420)$. The search function is $S(K, R) = \max\{i : R_i \leq K\}$. □

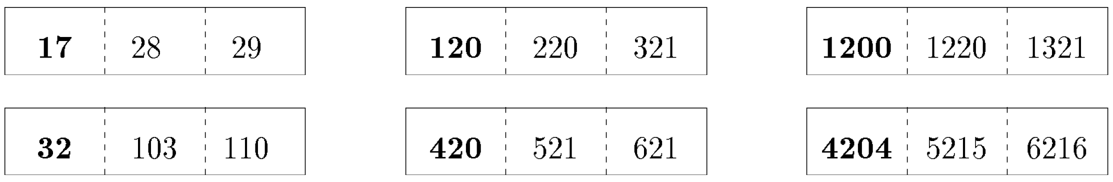


Figure 8: A partitioned search structure of 12 records.

In many cases the keys are sorted and the representative key is the minimum key of each block. The search function consists of a sequential search of the representative keys.

Given a partitioned search structure, we can construct an *index* to facilitate the search. The index consists of *levels*. The first level is a partitioned search structure of the representative keys, and each successive level is a partitioned search structure of the representative keys of previous level. This procedure is repeated until the level fits in a single block. Thus we get levels $I^h, I^{h-1}, \ldots, I^0$ = the original partitioned search structure. Each key of I^i is a representative key of I^{i-1}. The index contains pointers from the keys of I^i to the corresponding blocks of I^{i-1}. See Fig. 9 for the index to the search structure of Example 7.

7.1 Examples

The resultant data structure depends on what was the base partitioned search structure.

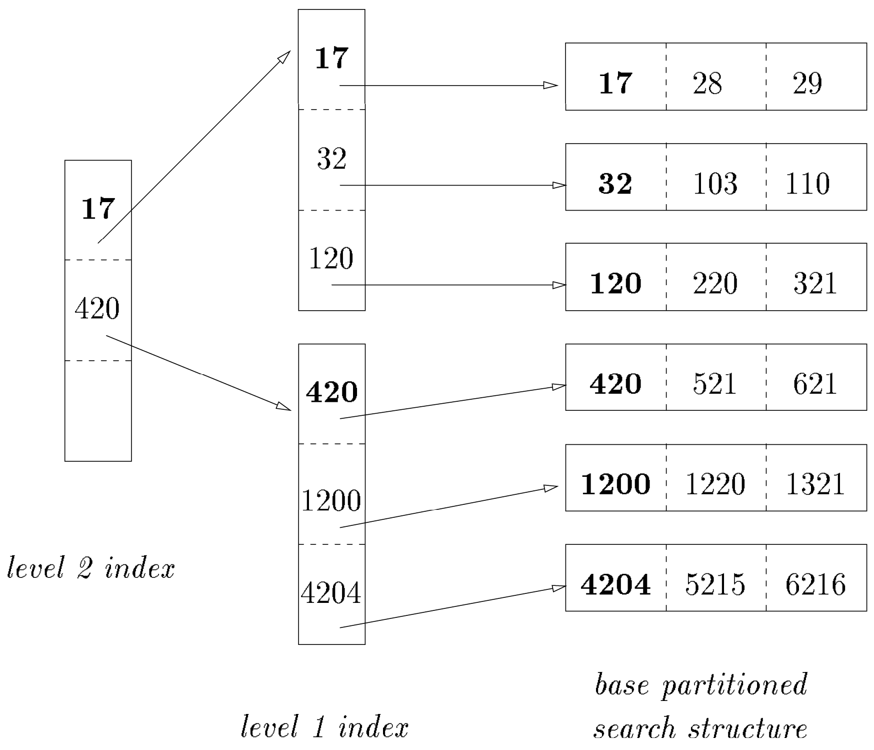


Figure 9: A partitioned search structure and 2-level index.

B-tree The base partitioned search structure is a sorted array partitioned into (at least half full) blocks.

Skip list The base partitioned search structure is a linked list.

Extendible-hash The base partitioned search structure is a hash table, and each level, requires one less bit.

Stratified indexes over full tries The base partitioned search structure is a trie, partitioned into blocks, such that each block is a tree.

Note that the base partitioned search structure need not be the same as the search structure used in the indexes, thus giving rise to hybrid systems such as BD-files [7, 8].

7.2 Search

To search for a key K, we search for K in the last level index—I^h, to reach block $B^{h-1}(K) \in I^{h-1}$. Then this block is searched as part of the level $h-1$ index to reach block $B^{h-2}(K) \in I^{h-2}$

To search for $K = 1321$ in the indexed search structure of Fig. 9, we first search for K in the single block of I^2, and follow the pointer from 420 to $B^1(K)$—the second block of I^1. We continue the search in $B^1(K)$ and follow the pointer from 1200 to $B^2(K)$—the sixth block of I^0 (the base partitioned block structure).

Each search involves accessing only one block at each level, thus the number of blocks accessed is equal to the number of levels. If the topmost two levels (level h and $h-1$) reside in main memory, and the remaining levels on disk, then each search requires $h-1$ disk accesses. In general, if each block contains at least β keys then an index for N keys consists of $h \leq \log_\beta N)$ levels.

7.3 Space

If each block contains at least β keys, the size of the level i index is N/β^i, and the additional space required for the index is bounded by

$$\frac{\beta}{1-1/\beta}N = \frac{N}{\beta-1} \ .$$

Thus for large blocks ($\beta \gg 1$) and small keys, the space required by the index is negligible.

7.4 Insertions

Insertions follow the B-tree paradigm: to insert a new record, we first insert it into the appropriate level 0 block. If this block overflows, we *split* it into two, obtaining a new representative key, which should be inserted into the next level index.

References

[1] Eisenbarth, Ziviani, Gonnet, K. Mehlhorn, and D. Wood. The theory of fringe analysis and its application to 2-3 trees and B-trees. *Inf. and Control*, 55:125–174, 1982.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—A fast access method for dynamic files. *ACM Journal on Database Systems*, 4:315–344, 1979.

[3] Ferragina and Grossi. The String B-tree: A new data structure for string search in external memory. *JACM*, 46:236–280, 1999.

[4] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *ACM Symposium on the Theory of Computing*, pages 693–702, 1995.

[5] A. Itai. The String B-tree by Ferragina and Grossi. Technical report, CS Dept., Technion, 1999.

[6] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[7] W. Litwin and D. Lomet. The bounded disorder access method. In *Proc. IEEE Conf. on Data Engineering*, pages 38–48, 1985.

[8] W. Litwin and D. Lomet. A new method for fast data searches with keys. *IEEE Software*, 4:16–24, 1987.

[9] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database system concepts.* McGraw-Hill, 3 edition, 1997.