k-d Range Search with Binary Patricia Tries

by

Qingxiu Shi and Bradford G.Nickerson

TR04-168, January 24, 2005

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

We use Patricia tries to represent textual and spatial data, and present a range search algorithm for reporting all k-d records from a set of size n intersecting a query rectangle. Data and queries include both textual and spatial data. Patricia tries are evaluated experimentally (for n up to 1,000,000) using uniform distributed random spatial data and textual data selected from the Canadian toponymy. We compared the performance of the Patricia trie for k-d points, k-d rectangles and k-d combined textual and spatial data to the k-d tree, R^*-tree, Ternary Search Trie and the naive method. Overall, our experiments show that Patricia tries are the best when $F \in [0, \log_2 n]$ (F is the number of data in range). The expected range search time for Patricia tries was determined theoretically, and found to agree with experimental results when $2 \leq k \leq 20$.

Contents

List of Tables

List of Figures

1 Introduction

Data structures to support efficient searching have been a fundamental research area of computer science for many years. The specific problem of searching we are concerned with is range search. Range search represents an important class of problems that occur in applications of databases, geographical information systems, computer graphics and computational geometry. Given a collection of records, each containing multidimensional attributes or keys, a range search asks for all records in the collection with key values each inside specified ranges.Over the past 30 years, more than 60 data structures for the range search problem have been presented [1] [6][21][27][35]. In this paper we present a research for k-d range search using Patricia tries, which is analyzed theoretically and experimentally.

1.1 Background and Related Work

We analyze a search structure by giving three cost functions of n (the number of points) and k (the number of dimensions): $P(n, k)$, the cost of preprocessing n points in k-space into a data structure; $S(n, k)$, the storage required by the data structure; $Q(n, k)$, the search time or query cost.

Bentley et al. [6][7] review several data structures for k-dimensional point range searching including sequential scan, projection, cells, k-d trees, range trees and k-ranges. The simplest approach to range searching is to store the n points in a sequential list. As each query arrives, all elements of the list are scanned and every record that satisfies the query is reported. Since all k keys of the n records must be stored and each k-key record is examined as the structure is built or searched, it is easy to see that the sequential scan structure has the properties $P(n, k) = O(kn)$, $S(n, k) = O(nk)$, $Q(n, k) = O(nk)$, and has the advantage of being trivial to implement on any storage medium. The projection technique is referred to as inverted lists by Knuth [27]. This technique was applied by Friedman, Baskett and Shustek [19] in their solution of the ”nearest neighbor” problem and by Lee, Chin, and Chang [30] to a number of database problems. $P(n, k) = O(kn \log n)$, $S(n, k) = O(kn)$ and $Q(n, k) = O(n^{1-1/k})$.

One of the most prominent data structures for solving the point range searching problem is called the k-d tree [4], a binary search tree that stores points of the k-dimensional space. At each intermediate node, the k-d tree divides the k-dimensional space in two parts by a (k-1)-dimensional hyperplane.

The direction of the hyperplane alternates between the k possibilities from one tree level to the next. Analysis of k-d trees for range searching has been considered by several researchers. The work required to construct a k-d tree and its storage requirements [5] are $P(n,k) = O(n \log n)$, $S(n,k) = O(nk)$. The search cost depends on the nature of the query. Lee and Wong [29] have shown that in the worst case $Q(n,k) = O(n^{1-1/k} + F)$, where F is the number of points found in the region. An improved version proposed in [20] is the adaptive k-d tree. When splitting, the adaptive k-d tree chooses a hyperplane that divides the space in two sub-spaces with equal number of points. All points are stored in the leaves. Devroye et al. [16] analyzed range search on squarish k-d trees and random k-d trees [10]. The k-d-B tree [33] combines properties of both the adaptive k-d tree and the B-tree [14]. Mehlhorn[31] discusses dd-trees which are very similar to k-d trees. At each step the data space is divided into three subsets, in contrast with two for the k-d tree.

The k-range is an efficient worst-case structure for range searching introduced by Bentley and Maurer [7]. They showed that one level k-ranges had $Q(n,k) = O(k \log n + F)$, $S(n,k) = P(n,k) = O(n^{2k-1})$, and developed two types of l-level k-ranges, overlapping and nonoverlapping. By appropriate choice of l for a given k and $\epsilon > 0$, the overlapping k-ranges can be made to have performance $S(n,k) = P(n,k) = O(n^{1+\epsilon})$ and $Q(n,k) = O(\log n + F)$, and nonoverlapping k-ranges require linear space, $P(n,k) = O(n \log n)$ and $Q(n,k) = O(n^{\epsilon})$. Range tree was introduced by Bentley [6], which achieved a good worst-case search time, but has relatively high preprocessing and storage costs. The k-dimensional range tree yield the performances $P(n,k) = O(n \log^{k-1} n)$, $S(n,k) = O(n \log^{k-1} n)$ and $Q(n,k) = O(log^k n + F)$. Lower bounds for range search were studied by Chazelle [13][12], who showed that a sequence of n operations for insertion, deletion, and reporting points in a given range costs $\Omega(n(\log n)^k)$. Chazelle [11] gives a comprehensive 3 overview of data structures for k-dimensional searching, including the description of a k-dimensional rectangle reporting algorithm (supporting dynamic operations) with $Q(n,k) = O(F(\log(\frac{2n}{F})^2) + \log^{k-1} n)$, which is close to the lower bound.

The problem of how to represent collections of rectangles arises in many applications. The most common example occurs when a rectangle is used to approximate other shapes for which it serves as the minimum enclosing object. The MX-CIF quadtree [25] associates each rectangle with the quadtree node corresponding to the smallest block that contains the rectangle in its

Table 1: Performance of data structures for k-dimensional range search.

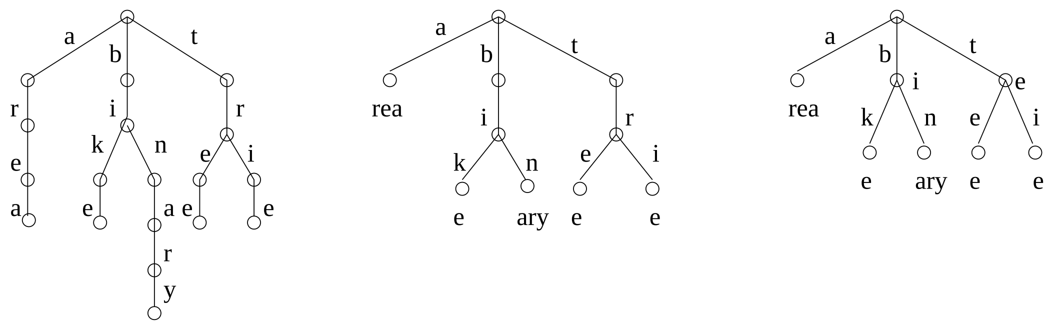
$DataStructure$	$P(n,k)$	$S(n,k)$	$Q(n,k)$
Sequential scan	$O(n)$	$O(n)$	$O(n)$
Projection	$O(n\log n)$	$O(n)$	$O(n^{1-1/k}+F)$
k-d tree	$O(n\log n)$	$O(n)$	$O(n^{1-1/k}+F)$
Range tree	$O(n\log^{k-1}n)$	$O(n\log^{k-1}n)$	$O(\log^{k}n+F)$
Overlapping k-ranges	$O(n^{1+\epsilon})$	$O(n^{1+\epsilon})$	$O(\log n+F)$
Non-overlapping k-ranges	$O(n\log n)$	$O(n)$	$O(n^{\epsilon}+F)$
d-fold tree	$O(n\log^{k-1}n)$	$O(n\log^{k-1}n)$	$O(\log^{k-1}n+F)$
R-tree	$O(n\log n)$	$O(n)$	$O(n+F)$
Priority R-tree	$O(n\log n)$	$O(n)$	$O(n^{1-1/k}+F)$

entirety. Building an MX-CIF quadtree of maximum depth h for n rectangles has a worst-case execution time of $O(hn)$, and the worst-case storage requirements are also $O(hn)$ and the excution time of range query is $O(hn^2)$.

The R-tree [23] is a hierarchical data structure that is derived from the B-tree, efficient indexing of multidimensional objects with spatial extent. A packed R-tree is proposed by Roussopoulos and Leifker [34], an R-tree that is built by successively applying a nearest neighbor relation to group objects in a node after the set of objects has been sorted according to a spatial criterion. Another alternative to the R-tree in dealing with rectangles is the R^+-tree [37], an extension of the k-d-B tree. The motivation for R^+-tree is to avoid overlap among the bounding rectangles. Several weakness of the original R-tree insertion algorithms stimulated Beckmann et al. [3] to work on an improved version of the R-tree, the R^*-tree, minimizing the overlap region between sibling nodes in the tree. For n rectangles, the construction time and space requirements of R-tree and R^+-tree are both $O(n)$ and $O(n^2)$ respectively. The Priority R-tree, or PR-tree was presented by Lars Arge et al.[2], and is provably asymptotically optimal and significantly better than other R-tree variants. Edelsbrunner [17] introduced the d-fold rectangle tree to support orthogonal range search on k-d hyper-rectangles with $S(n,k) = \Theta(n\log^{k-1}n)$, $P(n,k) = O(n\log^{k}n)$, and $Q(n,k) = O(\log^{2k-1}n + F)$. The comparison of the performance of data structures discussed above is depicted in Table 1.

1.2 Tries

Tries were introduced by Rene de la Briandais [28]. E. Fredkin [18] and Donald E. Knuth [27] developed them further. The trie is a simple order preserving data structure supporting fast retrieval of elements and efficient nearest neighbor and range searches. In general, a trie stores a set of, say n items, which are represented by keys from Σ^{∞} ,the set of all infinite sequences over a finite alphabet Σ. A trie is composed of branching nodes, also called internal nodes, and external nodes that stores the keys. The branching policy at any level, say d, is based on the d^{th} symbol of a key. Binary tries are data structures which use a binary representation of the key to store keys as a path in the trie, and follow the rule of direction determined by the d^{th} bit information of the key: branch left if 0 and branch right if 1. Trie(a) in Figure 1 is referred to as full trie [15]. There are two other tries. Trie(b) is an ordinary trie and trie(c) is a Patricia trie [32]. All three tries in Figure 1 are constructed from the same words: area, bike, binary, tree and trie.



(a)Full trie (b) Regular trie (c) Patricia trie

Figure 1: Tries example.

An ordinary trie is a pruned trie where all the leaf's parents will be the last bifurcating node in the corresponding full trie, and all the nodes between the leaves and the last bifurcating node have been removed. A Patricia trie and an ordinary trie store eliminated information in their leaves. A Patricia trie removes all the single descendant nodes. These skipped symbols are stored in the internal nodes or in the leaves.

The Patricia trie was discovered by D.R. Morrison [32]. All nodes in Patricia tries have degree greater than or equal to two by eliminating all one-child internal nodes. Patricia tries are well-balanced trees [38] in the sense that a random shape of Patricia tries resembles the shape of complete balanced trees. The Patricia trie has many applications, including lexicographical sorting, dynamic hashing algorithms, string matching, file systems, and most recently conflict resolution algorithms for broadcast communications [27]. The performance of Patricia tries is turned out to be superior to tries [26][27].

The ternary search trie (TST) is an alternative representation of the tries. In a TST, each node has a character and three links, corresponding to keys whose current digits are less than, equal to, or greater than the node's character [36]. TSTs provide an efficient implementation of string symbol tables and can quickly answer advanced queries. Advanced searching algorithms based on TSTs are likely to be useful in practical applications, and they present a number of interesting problems in the analysis of algorithms [8]. We construct a TST from the same word collection used in tries above and insert them in an order of bike, binary, area, tree and trie, shown in Figure 2. TSTs can be more efficient in space usage by putting keys in leaves at the point where they are distinguished and by eliminating one-way branching between internal nodes as in the Patricia trie. In our paper, we adapt TST to represent spatial data and combined textual and spatial data.

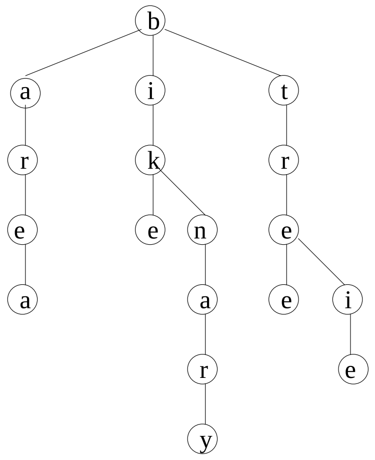


Figure 2: A TST for bike, binary, area, tree and trie.

1.3 Our Results

In Section 2 we use the binary Patricia tries to represent multidimensional points and rectangles, combined textual and spatial data, and present a range search algorithm for reporting all k-dimensional records from a size of n intersecting a random query rectangle. In Section 3 we theoretically analyze the average cost of the range search, which is proportional to the number of nodes in the trie visited during the range search. In Section 4 we present an extensive experimental study of the practical performance of the Patricia trie using uniform randomly generated spatial data and place names selected from the Canadian toponymy. We compare the performance of the Patricia trie to the k-d tree, TST and the naive method. Overall, our experiments show that Patricia tries take less time when $k \leq 10$, and the experimental results agree with the theoretical analysis. However, like the k-d tree, TST and R^*-tree, Patricia tries are limited by the curse of dimensionality meaning that the computing cost grows exponentially with the dimension of the data.

2 k-d Binary Patricia Tries

Binary tries are data structures which use a binary representation of the key to store keys as a path in the tree. Binary k-d tries use the principle of bit interleaving, e.g., a set of n k-dimensional keys:

$$\begin{gathered} P_1 = (P_{11}\,, P_{12}\,, \ldots, P_{1k}), \\ \vdots \\ P_n = (P_{n1}\,, P_{n2}\,, \ldots, P_{nk}), \end{gathered}$$

where P_{ij} can be spatial data or textual data, $1 \leq i \leq n$ and $1 \leq j \leq k$. Let $\tilde{P}_{ij}$ be the binary representation of P_{ij}, $\tilde{P}_{ij} \in \{0\,,1\}^\infty$, we produce one sequence $\tilde{P}_i \in \{0\,,1\}^\infty$ for each P_i by regular shuffling of the components $\tilde{P}_{i1}, \ldots, \tilde{P}_{ik}$, and use these new composite keys $\tilde{P}_1, \ldots, \tilde{P}_n$ to construct a trie. More precisely, if $\tilde{P}_{ij} = \tilde{P}^0_{ij}\tilde{P}^1_{ij}\tilde{P}^2_{ij}\cdots$, then $\tilde{P}_i = \tilde{P}^0_{i1}\cdots\tilde{P}^0_{ik}\tilde{P}^1_{i1}\cdots\tilde{P}^1_{ik}\tilde{P}^2_{i1}\cdots$, where the superscript indicates the bit position.

We denote by T the Patricia trie constructed by inserting n keys into an initially empty trie. There are altogether $n-1$ internal nodes and n leaves in T. The skipped bits are stored in an array SKIPSTR, and every leaf is associated with one key. Figure 3 is an algorithm to insert one key into T.

```
INSERT($T, P$)
 1  if $T = $ NIL
 2     then $T \leftarrow$ new $PatrieNode(\tilde{P}, P, k)$
 3     else $s \leftarrow \tilde{P}$
 4          $l = 0$
 5          while $l < \tilde{P}.length()$
 6          do $h \leftarrow$ LONGESTPREFIX($T.SKIPSTR, s$)
 7             if $h = \tilde{P}.length()$
 8                then $break$;
 9             $l \leftarrow l + h$
10             if $h = T.SKIPSTR.length()$ and $l < \tilde{P}.length()$
11                then if $\tilde{P}[l] = 1$
12                        then if $right[T] \neq$ NIL
13                                then $T \leftarrow right[T]$
14                                     $l \leftarrow l + 1; s \leftarrow s.substr(h + 1)$
15                        else if $left[T] \neq$ NIL
16                                then $T \leftarrow left[T]$
17                                     $l \leftarrow l + 1; s \leftarrow s.substr(h + 1)$
18                else $t \leftarrow T.SKIPSTR$
19                     $T.SKIPSTR \leftarrow t.substr(0, h)$
20                     $P \leftarrow$ new $PatrieNode(t.substr(h + 1), T.P, k)$
21                     $left[P] \leftarrow left[T]$
22                     $right[P] \leftarrow right[T]$
23                     $Q \leftarrow$ new $PatrieNode(s.substr(l + 1), P, k)$
24                     if $t[h] = 0$
25                        then $left[T] \leftarrow P$
26                             $right[T] \leftarrow Q$
27                        else $left[T] \leftarrow Q$
28                             $right[T] \leftarrow P$
29                     $break$
```

Figure 3: Pseudo-code for inserting a key P into a Patricia trie T. The function LONGESTPREFIX(a,b) returns the longest length of the same prefix of string ”a” and ”b”.

2.1 Spatial data

The k-d trie is a concise representation of k-dimensional space of bits. For the spatial data, we assume our search space is defined on the set of positive integers in k-dimensional space and the space is finite, limited by the number of bits B used to represent an integer. B is the number of bits used for representing a coordinate value in binary, $B = \log_2(MAX - MIN + 1)$, where MIN and MAX are the whole search space's lower and upper bounds. For example, if we have a set of three points in a 2-dimensional space of bits (as shown in Figure 4(a)): $P_1 = (2,5)$, $P_2 = (6,1)$ and $P_3 = (7,3)$. We assume $B = 3$, so we have:

$$\begin{aligned} P_1 = (010, 101) &\longrightarrow \tilde{P}_1 = 011001 \\ P_2 = (110, 001) &\longrightarrow \tilde{P}_2 = 101001 \\ P_3 = (111, 011) &\longrightarrow \tilde{P}_3 = 101111 \end{aligned}$$

The thick lines in Figure 4(a) represent partitions. The principle of partition is that each partition splits a space into two sub-spaces of equal size. k-d tries select the attributes to be split cyclically, i.e. $1, \cdots, k, 1, \cdots$. Figure 4(b) is a regular 2-d trie built up using the sequences $\tilde{P}_1, \tilde{P}_2$ and $\tilde{P}_3$. Removing the one-child internal nodes and storing the skipped information at the internal nodes, we get a 2-d Patricia trie (4(c)). In [9] a binary $2k$-d trie

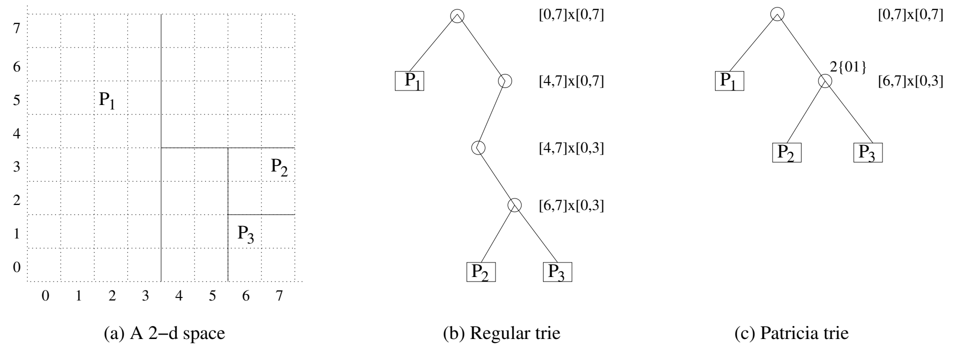


Figure 4: A 2-d space with three points and their corresponding tries.

data structure for k-d rectangles range search was investigated. A rectangle is represented as four coordinate values $(x^{\min}, x^{\max}, y^{\min}, y^{\max})$, which, after bit interleaving gives the bit string: $\tilde{P}^0_{x^{\min}} \tilde{P}^0_{x^{\max}} \tilde{P}^0_{y^{\min}}$ $\tilde{P}^0_{y^{\max}} \tilde{P}^1_{x^{\min}} \tilde{P}^1_{x^{\max}} \cdots \tilde{P}^{B-1}_{x^{\min}} \tilde{P}^{B-1}_{x^{\max}} \tilde{P}^{B-1}_{y^{\min}} \tilde{P}^{B-1}_{y^{\max}}$. Thus, a rectangle can be represented as a 4-d point in a binary trie. For example, there is a set of three 2-d rectangles $E = [1,3] \times [3,5]$, $F = [5,6] \times [5,7]$ and $G = [4,7] \times [1,3]$, shown in Figure 5(a). We assume B=3, so we have

$$\begin{aligned} E = (001, 011, 011, 101) &\longrightarrow \tilde{E} = 000101101111 \\ F = (101, 110, 101, 111) &\longrightarrow \tilde{F} = 111101011011 \\ G = (100, 111, 001, 011) &\longrightarrow \tilde{G} = 110001010111 \end{aligned}$$

Figure 5(b) is the corresponding Patricia trie.

Each node in k-d tries covers part of the k-d space, that is, every node has a cover space defined as $NC = [\mathcal{L}(p), \mathcal{U}(p)]_{p=1}^{k}$. Arrays $\mathcal{L}$ and $\mathcal{U}$ store the lower and upper bounds of a node's cover space. In Figure 4(b) and (c), the list of tuples is the cover space NC of each internal node. The root of k-d tries covers the whole space and child nodes cover half of the search space volume of their parent. The nodes on level ℓ split attribute $p = (\ell \mod k) + 1$ (at the root, $\ell = 0$). If a node on level ℓ has cover space $[\mathcal{L}_1,\mathcal{U}_1] \times \cdots \times [\mathcal{L}_p\,,\mathcal{U}_p] \times \cdots \times [\mathcal{L}_k,\mathcal{U}_k]$, then its left child's cover space is $[\mathcal{L}_1,\mathcal{U}_1] \times \cdots \times [\mathcal{L}_p\,,(\mathcal{L}_p + \mathcal{U}_p)/2] \times \cdots \times [\mathcal{L}_k,\mathcal{U}_k]$, and its right child's cover space is $[\mathcal{L}_1,\mathcal{U}_1] \times \cdots \times ((\mathcal{L}_p + \mathcal{U}_p)/2, \mathcal{U}_p] \times \cdots \times [\mathcal{L}_k,\mathcal{U}_k]$. For k-d Patricia tries, ℓ is not the level of the trie, but the length of the path from root to the node plus the length of the skipped bits in the internal nodes along the path. The node cover space must take the skipped bit string stored in the nodes into consideration. For example, in Figure 4(c), the node cover space of the root of 2-d Patricia trie is $[0,7] \times [0,7]$ ($\ell = 0$), the node cover space of its right child (denoted by NC_r) is computed as follows: first, $p = \ell \mod 2 + 1 = 1$, $NC_r = [4,7] \times [0,7]$ and $\ell = 1$; then, the first bit of the skipped bits string is '0', which means a left child node has been removed, $p = \ell \mod 2 + 1 = 2$, $NC_r = [4,7] \times [0,3]$ and $\ell = 2$; the second and last bit of the skipped bit string is '1', which means a right child node has been removed, $p = \ell \mod 2 + 1 = 1$, $NC_r = [6,7] \times [0,3]$ and $\ell = 3$. So the node cover space of root's right child is $[6,7] \times [0,3]$, as shown in Figure 4(c).

2.2 Textual data

Definition 1 *(Numeric Mapping [24]) Assume strings are comprised of symbols drawn from an alphabet of size α, and each symbol is mapped to an integer in the range 0 to α-1. Let a string of length c be $s_1 s_2 \cdots s_c$, with each symbol s_i mapped to an integer t_i, the string s is mapped to $\frac{t_1}{\alpha} + \frac{t_2}{\alpha^2} + \frac{t_3}{\alpha^3} + \cdots + \frac{t_c}{\alpha^c}$, which is a one-to-one mapping.*

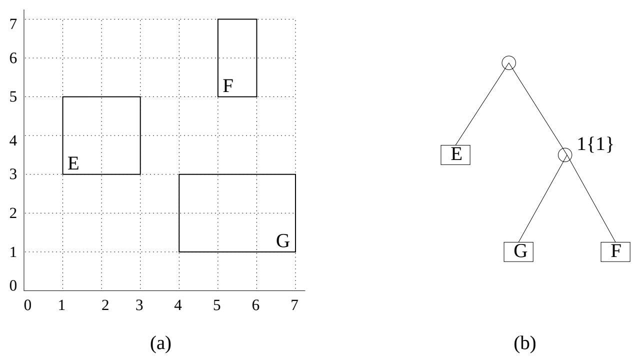


Figure 5: A 2-d space with three rectangles and their corresponding tries.

Though we can use numeric mapping technique to map strings to rational numbers and treat them as numeric data, a string may require a very large precision representation for the corresponding rational number, and the rational numbers require substantially more bits per symbol since α is likely to be much larger than 10. And we may need to keep the mapped numbers with high precision in order to support range searches. For an alphabet of size α, we assign a new decimal value in the range 0 to α-1 for each symbol, different from its decimal value in the *ASCII* table, and use $\lceil \log_2 \alpha \rceil$ bits to represent each symbol's decimal value. If a node's splitting attribute is p and it has cover space $[\mathcal{L}_1, \mathcal{U}_1] \times \cdots \times [\mathcal{L}_p, \mathcal{U}_p] \times \cdots \times [\mathcal{L}_k, \mathcal{U}_k]$, then its left child's cover space is $[\mathcal{L}_1, \mathcal{U}_1] \times \cdots \times [\mathcal{L}_p, (\mathcal{L}_p + \mathcal{U}_p)/2) \times \cdots \times [\mathcal{L}_k, \mathcal{U}_k]$ and its right child's cover space is $[\mathcal{L}_1, \mathcal{U}_1] \times \cdots \times [(\mathcal{L}_p + \mathcal{U}_p)/2, \mathcal{U}_p] \times \cdots \times [\mathcal{L}_k, \mathcal{U}_k]$, a little different from the numeric data.

For example, assume an alphabet with 4 characters $\{A, E, I, O\}$. We have $\alpha = 4$ and assign an integer from 0 to 3 to them respectively in sequence, and each symbol can be well represented by 2 bits. Consider the five strings AE, AO, E, I and O. We use the binary representation of each string to build up the trie. The mapping and the corresponding binary trie are shown in Figure 6.

2.3 Combined Textual and Spatial Data

We assume each of the coordinate values can be represented in B bits, and the symbols in strings are drawn from an alphabet of size α and each text symbol can be represented in $\lceil \log_2 \alpha \rceil$ bits. For simplicity, we assume the

	binary value	mapping
AE	0001	0/4+1/16=1/16
AO	0011	0/4+3/16=3/16
E	01	1/4
I	10	2/4
O	11	3/4

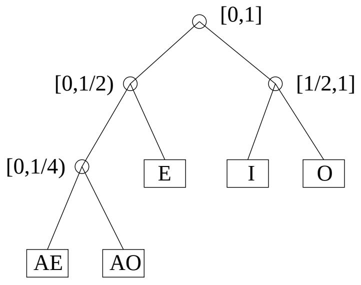


Figure 6: Mapping strings to rational numbers and the corresponding binary trie; the tuples besides the internal nodes are the node cover spaces.

first r dimensions in a k-dimensional key $P = (P_1, P_2, \cdots, P_k)$ are spatial data and the remainder are textual data, that is, $P_1, P_2, \cdots, P_r$ are numeric data and $P_{r+1}, \cdots, P_k$ are textual data, $0 \le r \le k$. First, we get the bit string of each dimension. If the length of the bit string of the textual data P_i is smaller than B, '0' is added at the end of the bit string to extend its length to B, $r+1 \le i \le k$. Then we can use the shuffle means mentioned above to get a sequence $\tilde{P}$ of P, and insert P using $\tilde{P}$ into the trie T using the algorithm in Figure 3.

Given a query rectangle $Q = [L_1, H_1] \times [L_2, H_2] \times \cdots \times [L_k, H_k]$, $L_i \le H_i$, a k-dimensional key $P = (P_1, P_2, \cdots, P_k)$ is in range iff $P_i \in [L_i, H_i]$, $\forall\, i \in \{1, 2, \cdots, k\}$. We obtain the query rectangles's cover space $QC = [L_i, H_i]^k$. On the p^{th} dimension, there are three types of relations of $QC(p)$ and $NC(p)$ ($QC(p)$ is the p-th component of the k-dimensional vector QC, and $NC(p)$ is the p-th component of the k-dimensional vector NC), which we call BLACK, GREY, and WHITE. Figure 7 illustrates the three colors for a node's cover on dimension p. Dashed lines are used for $QC(p)$ and solid lines for $NC(p)$. WHITE indicates when the trie can be pruned. BLACK relationships occurring k times contiguously indicates that all records in the subtree intersect Q. GREY indicates the trie must be searched further.

Definition 2 *If, on all k dimensions, the cover space relationship satisfies $QC(p) \cap NC(p) = BLACK$, $\forall p \in \{1, 2, \cdots, k\}$, then the node in the trie is black. If the cover space relationship satisfies $\exists p \in \{1, 2, \cdots, k\}$, such that $QC(p) \cap NC(p) = WHITE$, then the node in the trie is white. All other nodes are grey nodes.*

The RANGESEARCH algorithm (see Figure 8) is used to perform a range

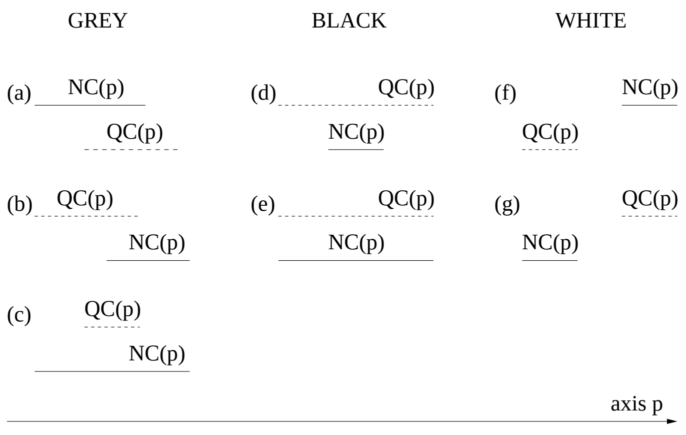


Figure 7: GREY((a),(b), and (c)), BLACK ((d) and (e)), WHITE ((f) and (g)) relationships of a trie node cover space NC to a query rectangle cover space QC in dimension p (from [9]).

search on T. The search proceeds from the root ($\ell = 0$) to the leaves, accounting for possible skipped bits stored at internal nodes. $\mathcal{L}$ and $\mathcal{U}$ are the lower and upper limits of the node's cover space. If $p = \ell \mod k$ is a numeric data dimension, the InRange function determines the value of $\mathcal{RI}[p]$; if p is a textual data dimension, function StringInRange (Figure 9) determines $\mathcal{RI}[p]$. Array $\mathcal{RI}$ of size k (initialized to store all GREY values) keeps track of the color of the $NC(p)$ to $QC(p)$ relationship for T and ancestors of T. Array *Flag* (initialized to contain all 2s) of size $k - r$ is used to track the state of textual data string bits being in range. Define t as the bit string traversed from the root to the current node T. Function StringInRange determines if the text part of a query rectangle Q intersects the trie node T. on the p^{th} dimension, $Flag[p] = 0$ indicates $t \leq SH.substr(0, t.length())$; $Flag[p] = 1$ indicates $t \geq SL.substr(0, t.length())$; $Flag[p] = 2$ indicates $t = SL.substr(0, t.length())$ and $t = SH.substr(0, t.length())$. If all ranges fall within Q at some node T, then all points in the subtree attached to T are in Q and are collected by the Collect function into a *List* for reporting. Function Color($\mathcal{RI}$) determines the color (black, grey or white) of node t. When we reach a leaf node, we determine whether it is in range using the CheckNode function; if so, it is added to *List*.

```
RANGESEARCH($T, \ell, \mathcal{L}, \mathcal{U}, \mathcal{RI}, Q, List, Flag$)
if $T$ is a leaf node
   then CHECKNODE($T, Q, List$)
   else $i \leftarrow 0$
        while $i < T.SKIPSTR.length()$
        do $p \leftarrow \ell \bmod k$
           if $p$ is a numeric data dimension
              then if $T.SKIPSTR[i] = 0$
                      then $\mathcal{U}[p] \leftarrow (\mathcal{L}[p] + \mathcal{U}[p])/2$
                      else $\mathcal{L}[p] \leftarrow (\mathcal{L}[p] + \mathcal{U}[p])/2 + \epsilon$
                   $\mathcal{RI}[p] \leftarrow$ INRANGE($\mathcal{L}[p], \mathcal{U}[p], p, Q$)
              else $q \leftarrow \ell/k + p - k$
                   STRINGINRANGE($T.SKIPSTR[i], p, q, \mathcal{RI}[p], Q, Flag[p]$)
           $i \leftarrow i + 1$
           $\ell \leftarrow \ell + 1$
        $C \leftarrow$ COLOR($RI$)
        if $C$ is grey
           then $p \leftarrow \ell \bmod k$
                if $left[T] \neq$ NIL
                   then if $p$ is a numeric data dimension
                           then $\mathcal{U}[p] \leftarrow (\mathcal{L}[p] + \mathcal{U}[p])/2$
                                $\mathcal{RI}[p] \leftarrow$ INRANGE($\mathcal{L}[p], \mathcal{U}[p], p, Q$)
                           else $q \leftarrow \ell/k + p - k$
                                STRINGINRANGE($0, p, q, \mathcal{RI}[p], Q, Flag[p]$)
                        RANGESEARCH($left[T], \ell + 1, \mathcal{L}, \mathcal{U}, \mathcal{RI}, Q, List, Flag$)
                if $right[T] \neq$ NIL
                   then if $p$ is a numeric data dimension
                           then $\mathcal{L}[p] \leftarrow (\mathcal{L}[p] + \mathcal{U}[p])/2 + \epsilon$
                                $\mathcal{RI}[p] \leftarrow$ INRANGE($\mathcal{L}[p], \mathcal{U}[p], p, Q$)
                           else $q \leftarrow \ell/k + p - k$
                                STRINGINRANGE($1, p, q, \mathcal{RI}[p], Q, Flag[p]$)
                        RANGESEARCH($right[T], \ell + 1, \mathcal{L}, \mathcal{U}, \mathcal{RI}, Q, List, Flag$)
           else if $C$ is black
                   then COLLECT($T, List$)
```

Figure 8: Range search algorithm to find combined text and points in a k-d+t trie T intersecting query rectangle Q. $T.SKIPSTR[i]$ is the $(i+1)^{st}$ bit of bit strings stored at compressed nodes of the Patricia trie T.

```
StringInRange($bit, p, q, \mathcal{RI}[p], Q, flag$)
1  $SL \leftarrow BinaryValue(Q.L[p]); SH \leftarrow BinaryValue(Q.H[p])$
2  if $bit = 0$
3     then switch
4              case $flag = 0$ :
5                  if $SL[q] = 1$
6                     then $\mathcal{RI}[p] = WHITE$
7              case $flag = 1$ :
8                  if $SH[q] = 1$
9                     then $\mathcal{RI}[p] = BLACK$
10             case $flag = 2$ :
11                 if $SL[q] = 1$ and $SH[q] = 1$
12                    then $\mathcal{RI}[p] = WHITE$
13                 if $SL[q] = 0$ and $SH[q] = 1$
14                    then $flag = 0$
15    else  switch
16             case $flag = 0$ :
17                 if $SL[q] = 0$
18                    then $\mathcal{RI}[p] = BLACK$
19             case $flag = 1$ :
20                 if $SH[q] = 0$
21                    then $\mathcal{RI}[p] = WHITE$
22             case $flag = 2$ :
23                 if $SL[q] = 0$ and $SH[q] = 0$
24                    then $\mathcal{RI}[p] = BLACK$
25                 if $SL[q] = 0$ and $SH[q] = 1$
26                    then $flag = 1$
```

Figure 9: StringInRange algorithm to determine if the text part of query rectangle Q intersects the trie text .

3 Range Search Cost

We adapt the approach used in [10] to analyze the range search cost of k-d Patricia tries, using Theorem P of [26]. Without loss of generality, the following discussions are all based on unit space $[0, 1]^k$, and we assume the input data and the query rectangle Q are drawn from a uniform random distribution.

3.1 Partial Match Queries with Patricia Tries

A partial match query asks for all records whose attributes are either specified or not. Given a query $q = (q_1, q_2, \cdots, q_k)$ when each q_j can be specified or unspecified (a so-called wild card, denoted by *), return all records whose attributes coincide with the specified attributes of q. If, e.g., $q = (17, *, *, 30)$, we look for all records whose first attribute is 17, fourth attribute is 30; the second and third attributes are left unspecified. The specification pattern ω of q is a word in $\{S, *\}^k$ where $\omega_j = S$ if q_j is specified and $\omega_j = *$ if q_j is unspecified; in our example we have the specification pattern $S{*}{*}S$. Partial match queries make sense if at least one attribute of the query is specified and at least one attribute is not. The analysis of the average cost of partial match queries in k-d Patricia Tries was addressed by P. Kirschenhofer and H. Prodinger [26]. We restate their theorem as follows:

Theorem 1 *Given a Patricia trie T built from n k-dimensional data and a partial match query of specification pattern ω, let $S \subset \{1, 2, \cdots, k\}$ be the set of specified coordinates, the average cost of partial match query measured by the number of nodes traversed in T is*

$$Q_S(n) = n^{1-\frac{s}{k}} \cdot \{\frac{(\frac{s}{k}+1)(1-2^{-s/k})}{k \log 2} \cdot \frac{\Gamma(\frac{s}{k})}{1-\frac{s}{k}} \cdot \Sigma_{j=0}^{k-1} (\delta_1\, \delta_2\, \cdots\, \delta_j) 2^{-j(1-s/k)} \\ + \delta(\log_2 n^{1/k})\}$$

where s is the number of specified attributes in ω, $\delta_j = 1$, if the $\jmath^{th}$ attribute of q is specified, and $\delta_j = 2$ if it is unspecified, and $\delta(x)$ a continuous periodic function of mean zero with small amplitude.

The following proposition relates the performance of range searches with the performance of patrial match queries.

Proposition 2 *Given a Patricia trie T built from n k-dimensional data and a partial match query of specification pattern ω, let $S \subset \{1, 2, \cdots, k\}$ be the* *set of specified coordinates, the average cost of partial match query measured by the number of nodes traversed in T is*

$$Q_S(n) = E\{\Sigma_{t=1}^{2n-1} \prod_{p \in S} |NC_t(p)|\},$$

where $|NC_t(p)|$, $1 \le p \le k$ are the cover spaces of node t in T.

Proof. If a node is visited, $q_p \in NC(p) = [\mathcal{L}_p, \mathcal{U}_p], \forall p \in S$. The probability that a node in trie T will be visited is determined by the volume of every node's cover space in the space $[0, 1]$. $\square$

3.2 Orthogonal Range Search Using Patricia Tries

We use the probabilistic model of random range queries introduced in [10]. A range query is a k-dimensional rectangle $Q = [L_1, H_1] \times [L_2, H_2] \times \cdots \times [L_k, H_k]$ with $0 \le L_i \le H_i \le 1$, for $1 \le i \le k$. To get the color types for a node in the trie, we compare all the k ranges of QC with NC. In our algorithm, the range search proceeds from the root to the leaves. On each level, we do at least one comparison of the k ranges and store the color as the node's state. If all k ranges are BLACK, the node is black; if one range is WHITE, the node is white; all the other conditions indicate the node is a grey node. Traversing stops on paths when we meet with black or white nodes and continues when grey nodes are encountered and continues collection black nodes in the subtree of the black nodes we first meet. The time complexity of range search is proportional to the number of grey nodes (GN) and black nodes (BN) visited in the trie built from the input data. We have the following equation:

$$Q(n, k) = \Sigma_{t=1}^{2n-1} 1_{[node_t \in GN \cup BN]},$$

where we use $1_{[A]}$ as the characteristic function of the event A. The formula counts the number of grey nodes, which, apart from the black nodes nodes traversed to report the in-range data, represents the time complexity of the range search algorithm.

Lemma 3 $E\{\Sigma_{t=1}^{2n-1} \prod_{p=1}^{k} |NC_t(p)|\} \le\ 1 + \log_2 n.$

Proof. We denote the volume of the node t in Patricia trie T as $|NC_t|$, and $|NC_t| = \prod_{p=1}^{k} |NC_t(p)|$. If there is no skipped bits in the root, $|NC_t| = 1$; otherwise, assume the size of the skipped bit string in the root is c, then

$|NC_t| = \frac{1}{2^c}$. Assume the size of the skipped bits in its left child node is c, then its left child node cover space's volume is $\frac{|NC_t|}{2^{1+c}}$. As the level ℓ of T's increases, the value of $|NC_t|$ decreases. Assume $n = 2^h$, $h \geq 0$, the value of $\Sigma_{t=1}^{2n-1} \prod_{p=1}^{k} |NC_t(p)|$ is maximal when T coincides with a complete binary tree, and there is no skipped bit string in any node. In this case, $\Sigma_{t=1}^{2n-1} \prod_{p=1}^{k} |NC_t(p)| = 1 + 2 \times \frac{1}{2} + 4 \times \frac{1}{4} + \cdots + 2^h \frac{1}{2^h} = 1 + \log_2 n$. $\square$

Proposition 4 $E\{\Sigma_{t=1}^{2n-1} 1_{[\exists p \in \{1,2,\cdots,k\}: |NC_t(p)| \geq \frac{1}{2}]}\} \leq C$, *where* $C = 4^k - 1$.

Proof. The number of nodes with a cover space's size $\geq \frac{1}{2}$ is maximum when the Patricia trie coincides with a complete binary tree from level 0 to level $2k$-1, and there is no skipped bit string stored in nodes in these levels, assume $k \ll n$. In this case, the maximum number is $1+2+2^2+\cdots+2^{2k-1} = 2^{2k}-1$. So we have $E\{\Sigma_{t=1}^{2n-1} 1_{[\exists p \in \{1,2,\cdots,k\}: |NC_t(p)| \geq \frac{1}{2}]}\} \leq 2^{2k} - 1$. $\square$

Theorem 5 *Given a Patricia trie T built from n k-dimensional data, consider a random range search with query rectangle Q, the expected range search time measured by the number of nodes traversed in T is*

$$\beta' \leq \frac{E\{Q(n)\}}{n \prod_{p=1}^{k} |QC(p)| + \Sigma_{S \subset \{1,\cdots,k\}} (\prod_{p \notin S} |QC(p)|) \gamma(s) n^{1-\frac{s}{k}} + \log_2 n} \leq \beta,$$

where β and β' are constants depending on k only, and $0 < s = |S| < k$ *and*

$$\gamma(s) = \frac{(\frac{s}{k}+1)(1-2^{-s/k})}{k \log 2} \cdot \frac{\Gamma(\frac{s}{k})}{1-\frac{s}{k}} \cdot \Sigma_{j=0}^{k-1} (\delta_1\, \delta_2\, \cdots\, \delta_j) 2^{-j(1-s/k)}$$

with $\delta_j = 1$ *if* $\delta_j \in S$ *and* $\delta_j = 2$ *if* $\delta_j \notin S$.

Proof. $E\{Q(n,k)\} = E\{\Sigma_{t=1}^{2n-1} 1_{[node_{t \in GN \cup BN}]}\}$. This calculation includes the reporting time for collection of the subtree of black nodes which arises during the traversal. The probability that a node is black or grey is given as:

$$Pr[node_i \in GN \cup BN] \leq \prod_{p=1}^{k} (|NC(p)| + |QC(p)|).$$

We have

$$\begin{aligned}
E\{Q(n)\} &\leq E\{\Sigma_{t=1}^{2n-1}\prod_{p=1}^{k}(|QC(p)|+|NC_t(p)|)\} \\
&= \Sigma_{S\subseteq\{1,\cdots,k\}}(\prod_{p\notin S}|QC(p)|)E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|\} \\
&= \Sigma_{S=\emptyset}(\prod_{p=1}^{k}|QC(p)|)E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|\} \\
&\quad +\Sigma_{S\subset\{1,\cdots,k\},0<|S|}(\prod_{p\notin S}|QC(p)|)E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|\} \\
&\quad +\Sigma_{S=\{1,\cdots,k\}}(\prod_{p\notin S}|QC(p)|)E\{\Sigma_{t=1}^{2n-1}\prod_{p=1}^{k}|NC_t(p)|\}.
\end{aligned}$$

For the lower bound notice that,

$$\begin{aligned}
E\{Q(n)\} &\geq E\{\Sigma_{t=1}^{2n-1}1_{[node_t\in GN\cup BN]}1_{[\forall p\in\{1,\cdots,k\}:|NC_t(p)|<\frac{1}{2}]}\} \\
&\geq E\{\Sigma_{t=1}^{2n-1}\prod_{p=1}^{k}(|NC_t(p)|+\frac{|QC(p)|}{2})1_{[\forall p\in\{1,\cdots,k\}:|NC_t(p)|<\frac{1}{2}]}\} \\
&= E\{\Sigma_{t=1}^{2n-1}\prod_{p=1}^{k}(|NC_t(p)|+\frac{|QC(p)|}{2})\} \\
&\quad -E\{\Sigma_{t=1}^{2n-1}\prod_{p=1}^{k}(|NC_t(p)|+\frac{|QC(p)|}{2})1_{\exists p\in\{1,\cdots,k\}:|NC_t(p)|\geq\frac{1}{2}]}\} \\
&= \Sigma_{S\subseteq\{1,\cdots,k\}}\prod_{p\notin S}\frac{|QC(p)|}{2}E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|\} \\
&\quad -\Sigma_{S\subseteq\{1,\cdots,k\}}\prod_{p\notin S}\frac{|QC(p)|}{2}E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|1_{[\exists p\in\{1,\cdots,k\}:|NC_t(p)|\geq\frac{1}{2}]}\}.
\end{aligned}$$

From Proposition 4, we can bound the second item as:

$$\begin{aligned}
&\Sigma_{S\subseteq\{1,\cdots,k\}}\prod_{p\notin S}\frac{|QC(p)|}{2}E\{\Sigma_{t=1}^{2n-1}\prod_{p\in S}|NC_t(p)|1_{[\exists p\in\{1,\cdots,k\}:|NC_t(p)|\geq\frac{1}{2}]}\} \\
\leq\ & E\{\Sigma_{t=1}^{2n-1}1_{[\exists p\in\{1,\cdots,k\}:|NC_t(p)|\geq\frac{1}{2}]}\} \\
\leq\ & C.
\end{aligned}$$

The results follow by Theorem 1, Proposition 2 and Lemma 3. $\square$

Put differently,

$$E\{Q(n)\} \leq$$
$$\beta(n\prod_{p=1}^{k}|QC(p)| + \Sigma_{S\subset\{1,\cdots,k\}}(\prod_{p\notin S}|QC(p)|)\gamma(s)n^{1-s/k} + \log_2 n),$$

The first term accounts for the data returned by the range search. The third term arises from the height of the trie which is unavoidable. The second term dominates, and arises from the number of grey nodes checked to determine intersection with Q.

4 Experiments

In this section we describe the results of our experimental study of the performance of the Patricia tries. We compare Patricia to several other multidimensional data structures: k-d tree, ternary search tree (TST) and a naive method. For the naive approach, an array of size n is used to store the data; when searching, a $O(n)$ time scan of the entire array is made. We adapt TSTs to represent k-dimensional data as follows: after regular shuffling of the k components of each point i as discussed in Section 2, each character is represented by k of Bk bits of the composite key $\tilde{P}_i$. For rectangles, each character is represented by $2k$ of the $2Bk$ bits. Figure 10(a) and (b) are TSTs representing the data in Figure 4 and 5 respectively.

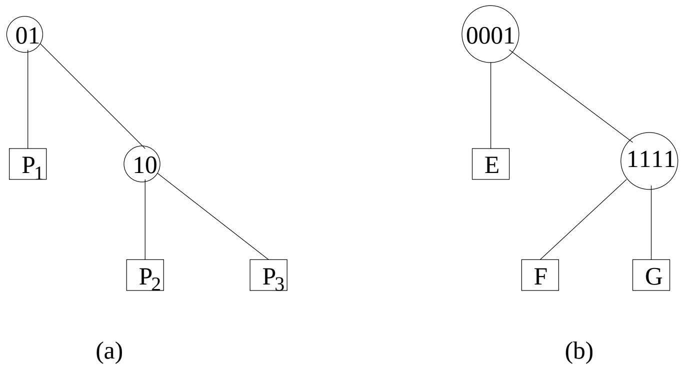


Figure 10: TSTs for (a) the points in Figure 4 and (b) the rectangles in Figure 5.

Experimental validation of our approach was performed using uniform random distributed textual and numeric data for $2 \leq k \leq 20$ and n up to 1,000,000. We assume B=30. Experiments were run on a Sun Microsystems V880 with four 1.2 GHz UltraSPARC III processors, 16 GB of main memory, running Solaris 8. Each experimental point in the following graphs was done with an average of 300 test cases.

4.1 Query Squares with Fixed Volume

The k-dimensional points were uniformly and randomly generated. We compared the experimental results of Patricia tries to the theoretical results, and found that they are consistent when $2 \leq k \leq 20$ and $n = 1,000,000$ (see Figure 11). We show the results of experiments with k-dimensional square window queries with volumes that range from 0.01% to 1% of the total space for Patricia tries for k-dimensional points in Figures 12 and 13 respectively, in comparison with the k-d tree, TST and naive method.

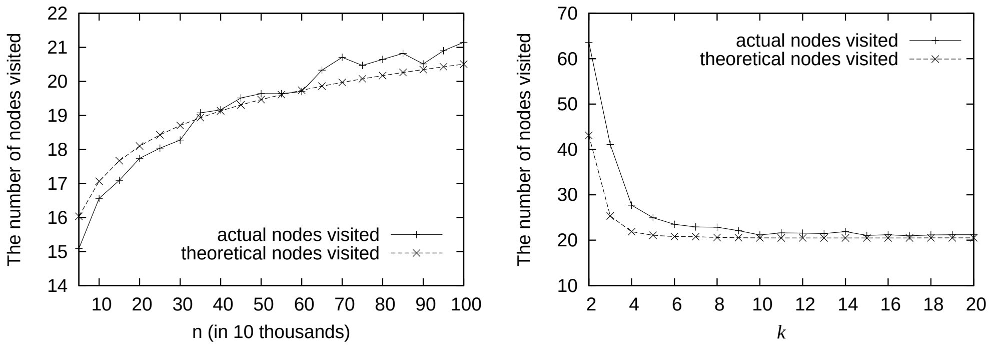
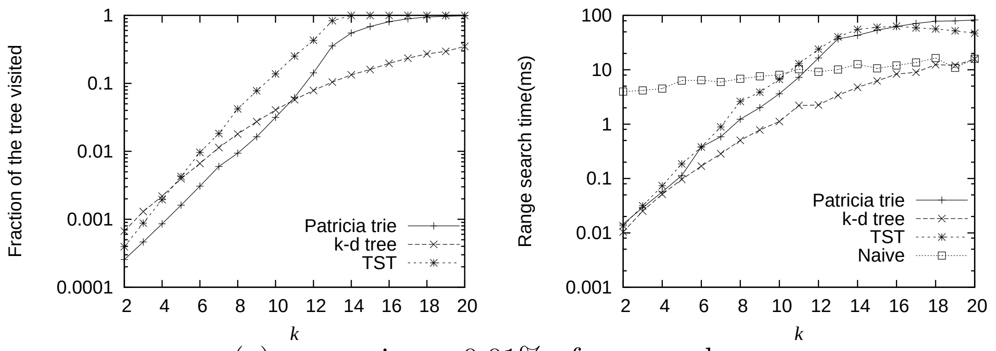
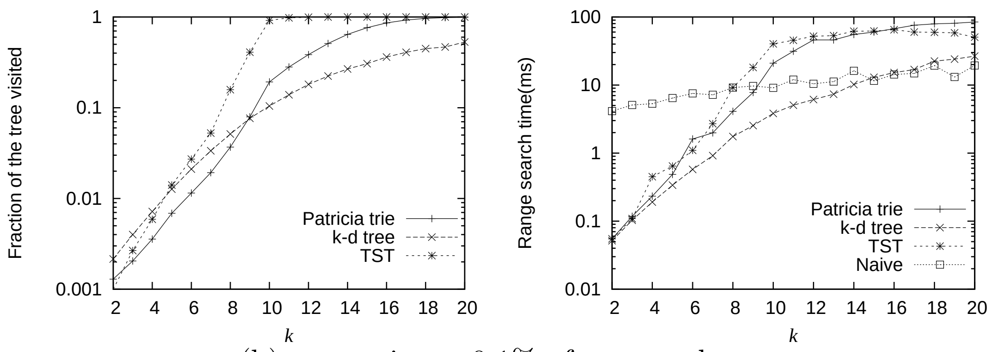


Figure 11: The experimental and theoretical number of nodes visited for Patricia trie range search with k-d query squares size of 0.01^k of total space volume for (left) $k = 10$ and $50,000 \leq n \leq 1,000,000$, (right) $2 \leq k \leq 20$ and $n = 1,000,000$.

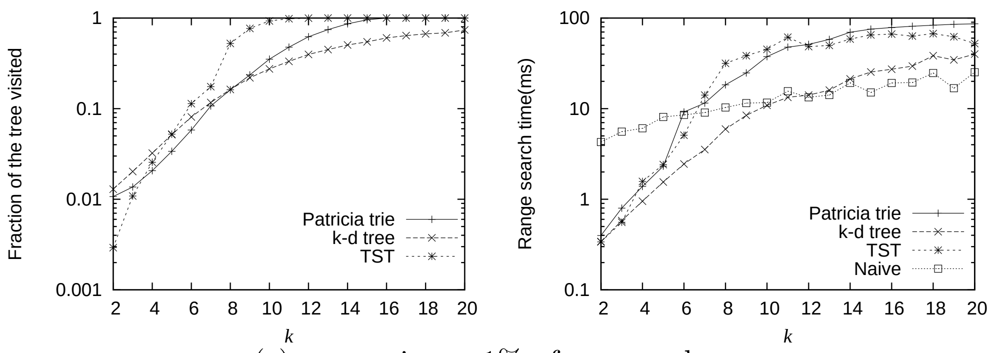
The rectangle centers were uniformly distributed and the lengths of their sides uniformly and independently distributed between 0 and $maxsize$. Range search reports the rectangles which intersect with the query rectangle. We show the results of experiments with k-dimensional square window queries with volumes that range from 0.01% to 1% of the total space for Patricia tries for k-dimensional rectangles in Figures 14 and 15, in comparison with the



(a) query size = 0.01% of space volume

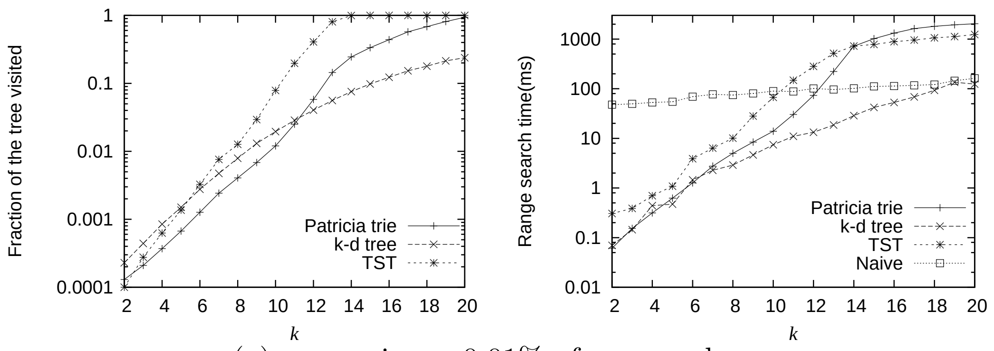


(b) query size = 0.1% of space volume

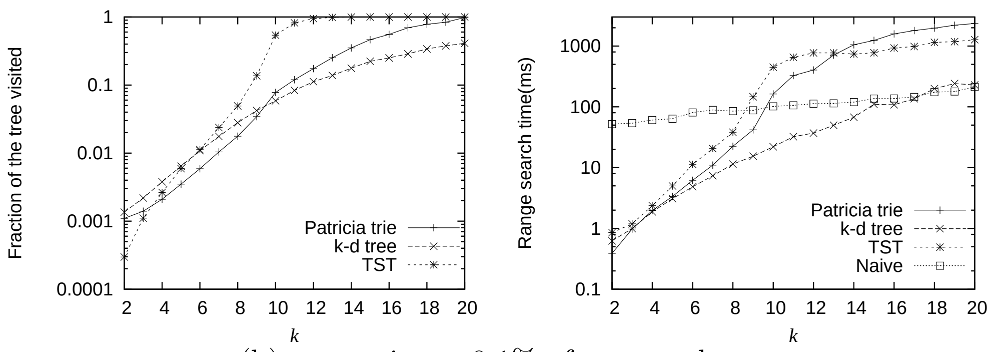


(c) query size = 1% of space volume

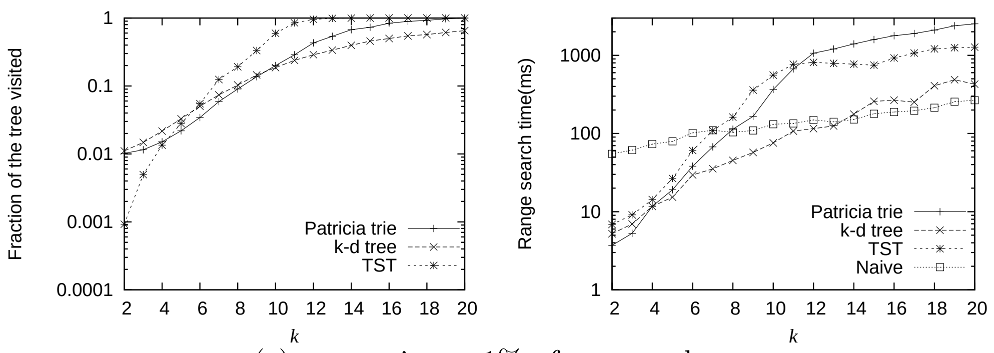
Figure 12: The fraction of the tree visited (left column) and range search time in milliseconds (right column) for the k-d tree, Patricia trie, TST and naive range search with k-d query square sizes of (a) 0.01%, (b) 0.1 % and (c) 1% of total space volume ($n = 100000$ and $2 \leq k \leq 20$, the average number of points in range is (a) 10, (b) 100 and (c) 1000).



(a) query size = 0.01% of space volume



(b) query size = 0.1% of space volume



(c) query size = 1% of space volume

Figure 13: The fraction of the tree visited (left column) and range search time in milliseconds (right column) for k-d tree, Patricia trie, TST and naive range search with the k-d query square sizes of (a) 0.01%, (b) 0.1 % and (c) 1% of total space volume (n = 1,000,000 and $2 \leq k \leq 20$, the average number of points in range is (a) 100, (b) 1000 and (c) 10000)).

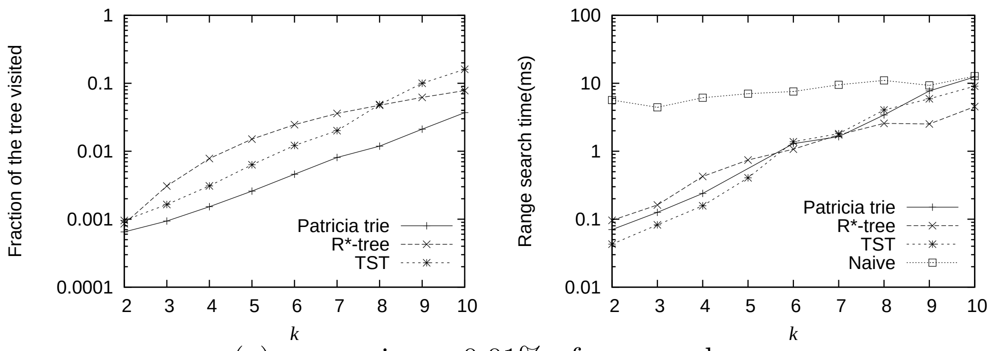
R^*-tree (the maximum number of children M=3), TST and naive method. The average number of rectangles in range is shown in Table 2.

Table 2: The average number of rectangles in range (n=100,000).

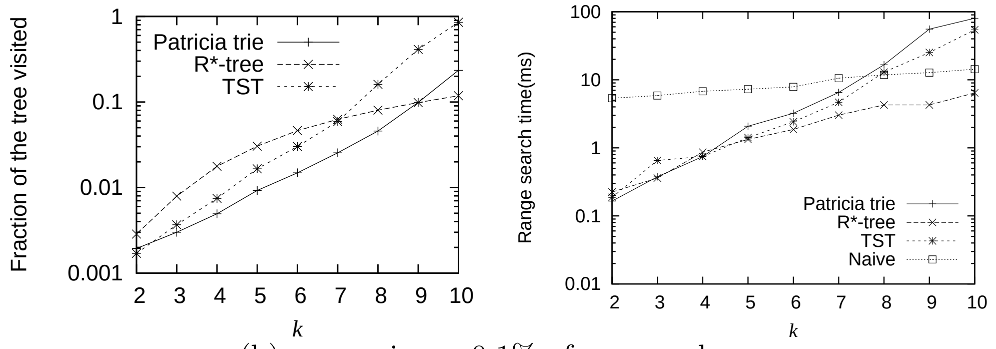
maxsize	0.001			0.01			0.1		
query volume	0.01%	0.1%	1%	0.01%	0.1%	1%	0.01%	0.1%	1%
k=2	11	106	1021	39	175	1232	1217	1783	4343
3	10	103	1015	17	136	1177	339	923	3766
4	10	102	1014	15	127	1175	193	756	3971
5	9	102	1018	13	126	1188	154	745	4387
6	9	102	1014	13	126	1200	142	786	4979
7	9	102	1021	13	128	1223	145	859	5666
8	10	102	1023	13	129	1238	159	976	6494
9	9	102	1021	13	133	1269	173	1081	7388
10	9	102	1025	13	134	1291	193	1277	8530

4.2 Random Query Rectangles

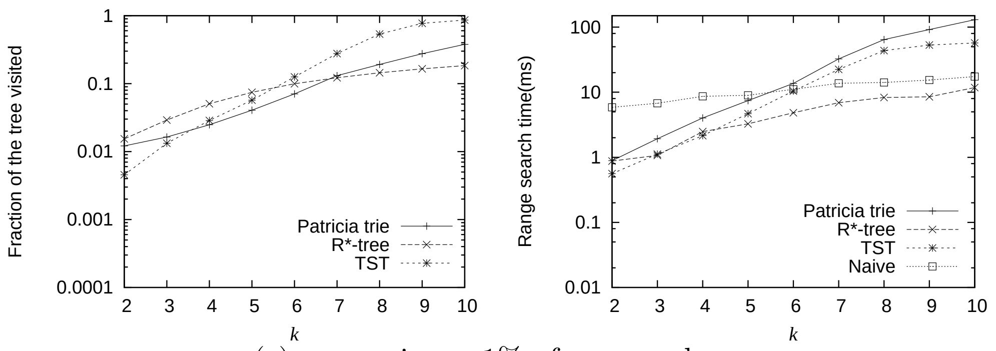
The experimental results of for Patricia tries for k-dimensional points were shown in Figures 16, in comparison with the k-d tree and TST with $F \in [0, \log_2 n]$, where F denotes the number of data in range. The results of experiments for Patricia tries for k-dimensional rectangles in Figures 17, in comparison with the R^*-tree (the maximum number of children M=3) and TST with $F \in [0, \log_2 n]$. We tested the Patricia tries with real-life textual data (names randomly chosen from the Canadian toponymy [22]). Experimental results are shown in Figures 18 for the fraction of the nodes visited in the k-d tree, Patricia tries and TST for $r = k - 2$ (two text strings per point) with $F \in [0, \log_2 n]$.



(a) query size = 0.01% of space volume

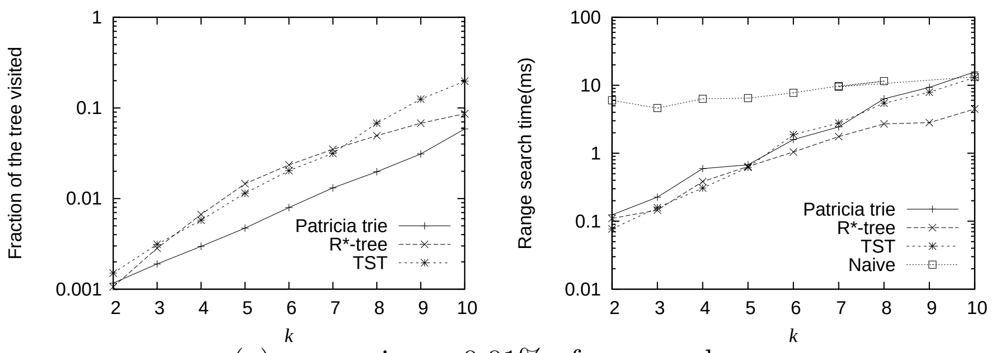


(b) query size = 0.1% of space volume

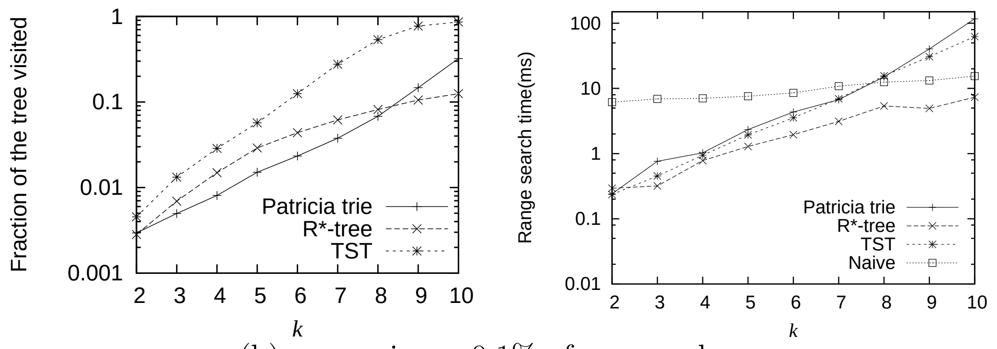


(c) query size = 1% of space volume

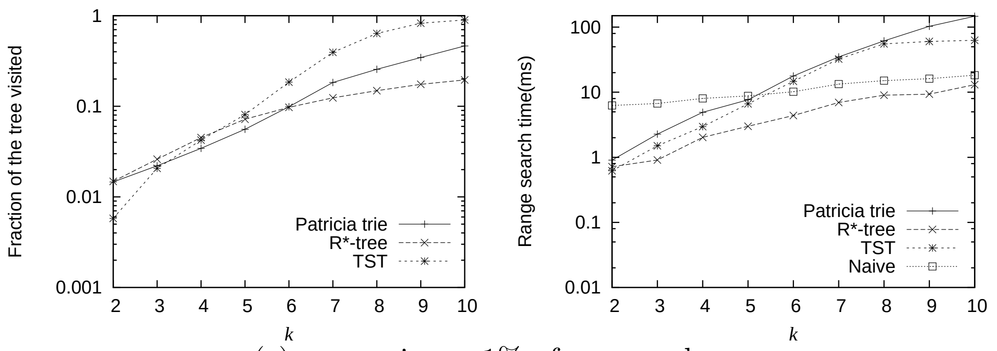
Figure 14: The fraction of the tree visited(left column) and range search time in milliseconds(right column) for the R^*-tree, Patricia trie, TST and naive range search for k-d query square sizes of (a) 0.01% (b) 0.1% and (c) 1% of total space volume (n = 100,000, $2 \leq k \leq 10$ and $maxsize$=0.001).



(a) query size = 0.01% of space volume



(b) query size = 0.1% of space volume



(c) query size = 1% of space volume

Figure 15: The fraction of the tree visited(left column) and range search time in milliseconds(right column) for the R^*-tree, Patricia trie, TST and naive range search for k-d query square sizes of (a) 0.01%, (b) 0.1% and (c) 1% of total space volume (n = 100,000, $2 \leq k \leq 10$ and $maxsize$=0.01).

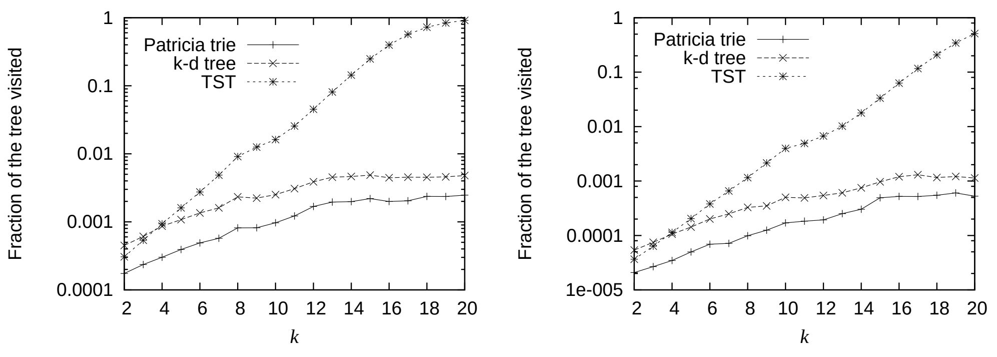


Figure 16: The fraction of the tree visited for the k-d tree, Patricia trie, and TST range search for (left) n=100,000 and (right) n=1,000,000 k-dimensional points ($F \in [0, \log_2 n]$ and $2 \leq k \leq 20$).

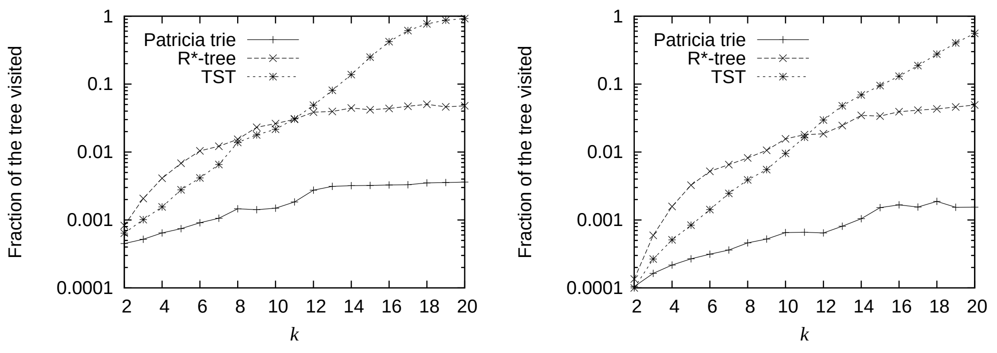


Figure 17: The fraction of the tree visited for the R^*-tree, Patricia trie, and TST range search for (left) n=100,000 and (right) n=1,000,000 random k-dimensional rectangles ($F \in [0, \log_2 n]$, $2 \leq k \leq 20$ and $maxsize$=0.01).

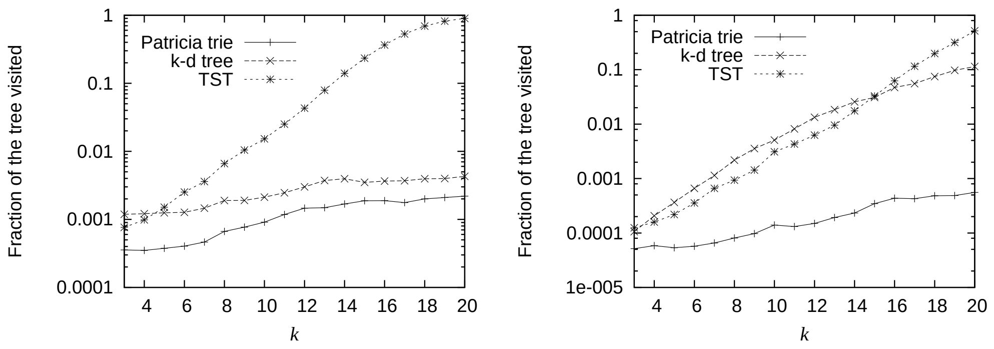


Figure 18: The fraction of the tree visited for the k-d tree, Patricia trie, and TST range search for (left) n=100,000 and (right) n=1,000,000 k-dimensional combined textual and spatial data ($F \in [0, \log_2 n]$, $r = k-2$ and $3 \leq k \leq 20$. The textual data are chosen from Canadian toponymy).

5 Conclusions and Future work

A range search algorithm for Patricia tries for k-d textual and spatial data was presented, theoretically analyzed and experimentally compared to the k-d tree, R^*-tree, TST and the naive method. Our expected time analysis of range search for k-d Patricia trie compares well with the experimental results. The experimental results show that the Patricia tries outperform k-d tree, R^*-tree and TST when $F \in [0, \log_2 n]$, and when n increases, the fraction of nodes visited during range search in Patricia tries decreases. However, when k is increasing, like the k-d tree, R^*-tree, and TST, Patricia tries are limited by the curse of dimension. Table 3 shows the average height of Patricia tries, where height+skips means the height of trie plus the length of skipped strings stored in the internal nodes along the path from the root to the leaf node. We can see that the height+skips of the trie doesn't change with the increase of k. So when k=2, the space of each dimension can be divided into halves near 17 times averagely from the root to the leaves, the nodes can be pruned quickly during range search; but when k=20, the space of each dimension only is divided into halves less than twice averagely, which results in much more nodes visited during range search. What's more, when the query size of the query rectangle $|QC|$ is fixed for the same n, for example, $|QC|$=0.01, when k=2, $|QC(1)|$=$|QC(2)|$=0.1; when k=20, $|QC(p)| = (0.01)^{1/20} \approx 0.794$, $1 \leq p \leq 20$. So the number of nodes visited and the range search time grows exponentially with the increase of k.

Table 3: The average height of Patricia tries (n=100,000).

	k=2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
height	22	23	22	22	23	22	22	23	22	23	22	22	22	22	22	23	22	22	22
height+skips	35	34	34	34	34	34	33	34	34	34	34	34	34	35	34	34	34	34	34

References

[1] P. Agarwal. *Handbook of Discrete and Computational Geometry*, chapter Range Searching, pages 575–598. CRC Press LLC, Boca Raton, FL, 1997.

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *Proc. of the 2004 ACM SIGMOD international conference on Management of data*, pages 347–358, 2004.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, May 23-25 1990.

[4] J. Bentley. Multidimensional binary search trees for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.*, 5(4):333–340, 1979.

[6] J. Bentley and J. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[7] J. Bentley and H. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[8] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of the eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, January 1997.

[9] L. Bu and B. Nickerson. Multidimensional orthogonal range search using tries. In *Canadian Conference on Computational Geometry*, pages 161–165, Halifax, N.S., August 2003.

[10] P. Chanzy, L. Devroye, and C.Zamora-Cura. Analysis of range search for random k-d trees. *Acta Informatica*, 37(4/5):355–383, 2001.

[11] B. Chazelle. A functional approach for data structure and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, June 1988.

[12] B. Chazelle. Lower bounds for orthogonal range search: II. the arithmatic

model. *Journal of the ACM*, 37(3):39–463, July 1990.

[13] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, April 1990.

[14] D. Comer. The ubiquitous b-tree. *Computing Surveys*, 11(2):121–137, 1979.

[15] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):377–387, 1970.

[16] L. Devroye, J. Jabbour, and C. Zamora-Cura. Squarish k-d trees. *SIAM Journal of Computing*, 30(5):678–700, 2000.

[17] H. Edelsbrunner. A new approach to rectangle intersection part i. *Int. J. Computer Mathematics*, 13:209–219, 1983.

[18] E. Fredkin. Trie memory. *Communiations of the ACM*, 3:490–500, 1960.

[19] J. Friedman, F. Baskett, and L. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. Comput.*, 24(10):1000–1006, 1975.

[20] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[21] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

[22] GeoBase. Homepage: http://www.geobase.ca, 2004.

[23] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, Boston, MA, June 18-21 1984.

[24] H. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 403–414, Dallas, Texas, USA, May 2000.

[25] G. Kedem. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proc. ACM/IEEE 19th Design Automation Conf.*, pages 352–357, Las Vegas, NV, June 1982.

[26] P. Kirschenhofer and H. Prodinger. Multidimensional digital searching-alternative data structures. *Random Structures and Algorithms*, 5(1):123–134, 1994.

[27] D. Knuth. *The art of computer programming: sorting and searching*, volume 3, pages 492–512. Addison-Wesley, Reading, Mass., 2 edition, 1998.

[28] R. la Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, volume 15, pages 295–298, San Francisco, 1959.

[29] D. Lee and C. Wong. Quintary trees: a file structure for multidimensional database systems. *ACM Transaction on Database Systems*, 5:339–353, 1980.

[30] R. Lee, Y. Chin, and S. Chang. Application of principal component analysis to multi-key searching. *IEEE Trans. Softw. Eng.*, 2(3):185–193, 1976.

[31] K. Mehlhorn. *Data structures and algorithms 3: Multidimensional tree struc-*

tures and computational geometry. Springer-Verlag, Berlin, 1984.

[32] D. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 14(4):514–534, October 1968.

[33] J. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, Ann Arbor, Michigan, April 29 - May 1 1981.

[34] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 17–31, Austin, TX, May 28-31 1985.

[35] H. Samet. *The design and analysis of spatial data strutures.* Addison-Wesley, Reading, MA, 1990.

[36] R. Sedgewick. *Algorithms in C++*, chapter Radix Search, pages 623–668. Addison-Wesley, Reading, Mass., 2001.

[37] T. Sellis. Efficiently supporting procedures in relational database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 278–291, San Francisco, CA, May 27-29 1987.

[38] W. Szpankowski. Patricia tries again revisited. *Journal of the ACM*, 37(4):691–711, 1990.