PK-tree: A Spatial Index Structure for High Dimensional Point Data *

Wei Wang, Jiong Yang, and Richard Muntz
Department of Computer Science
University of California, Los Angeles
{weiwang,jyang,muntz}@cs.ucla.edu

Abstract

In this paper we present the PK-tree which is an index structure for high dimensional point data. The proposed indexing structure can be viewed as combining aspects of the PR-quad or K-D tree but where unnecessary nodes are eliminated. The unnecessary nodes are typically the result of skew in the point distribution and we show that by eliminating these nodes the performance of the resulting index is robust to skewed data distributions. The index structure is formally defined, efficiently updatable and bounds on the number of nodes and the mean height of the tree can be proved. Bounds on the expected height of the tree can be given under certain mild constraints on the spatial distribution of points. By appropriate choice of parameters the index structure can be allocated one node per page achieving acceptable storage utilization as well as low I/O counts per query. We also present a consistency control scheme which allows a large number of readers and writers to access the index structure concurrently. Empirical evidence both on real data sets and generated data sets with a range of distributions shows that the PK-tree outperforms the recently proposed spatial indexes based on the R-tree such as the SR-tree and X-tree by a wide margin. It is significant also that the relative performance advantage of the PK-tree grows with the dimensionality of the data set.

Keywords multi-dimensional index, spatial indexing, point data sets, image database.

1 Introduction

Large databases of objects are becoming more important in applications such as medical image archives, biology, CAD, etc. It is common in such databases for objects to be described by a vector of numeric attributes. Particularly in similarity search applications, e.g., in content based access in image databases, high dimensional spaces are common. Therefore, a variety of dynamic spatial indexing structures have been proposed in the past few years [Sam90] [Ber96] [Hen89] [Kat97] [Lom90]. Most index structures become less efficient with increasing dimensionality and this is where the challenge lies.

All the existing dynamic spatial indexing structures achieve some degree of success. However, there is still room for improvement. Many applications can benefit from the simple and easy tree traversal

*This is an extended version of [Wan97], which include pagination and concurrency control scheme.

property of spatial decomposition. We introduce a new spatial indexing structure based on regular spatial decomposition, the PK-tree (Pyramid K-instantiable tree). A PK-tree achieves a better bound on the height of the tree for skewed data distributions than other trees based on regular decomposition because it only instantiates non-leaf cells with at least a certain number of nonempty children. This restriction largely eliminates the problem of the height of the tree growing large due to skew in the spatial distribution of points. Updates to the PK-tree are inexpensive and the performance of the PK-tree is shown via experiments compared well with that of any existing variations of the R-tree for the most common types of queries especially for higher dimensional data. Moreover, a PK-tree is independent of the order of data insertions and deletions. Last but not least, the PK-tree is based on a solid theoretical foundation. Its uniqueness for any data set, a bound on the mean height, and bounds on the number of children in a node can all be proved formally [Yan97].

The remainder of the paper is organized as follows. Related work is introduced in Section 2. We formally define the PK-tree structure in Section 3. Then in Section 4, we introduce the properties of the PK-tree. The pagination and operations of PK-tree are introduced in Sections 5 and 6. Section 7 briefly discusses the concurrency control on PK-tree. In Section 8, we summarize the results of empirical studies comparing PK-tree performance with other spatial index methods. Finally, we summarize and discuss conclusions in Section 9.

2 Related Work

2.1 Index Structure Based on Spatial Decomposition

One of the major drawbacks of the PR quad-tree is that, for continuous coordinates, the height of the tree is dependent on the minimum Euclidean distance separating two data points rather than the total number of data points. Depending on the application this can lead to inefficient storage/search performance and severely unbalanced trees. The K-D tree [Ben75] removes some of the unnecessary nodes from PR quad tree. However, since data can be highly skew distributed over a space, the height of the K-D tree can be very large. To construct a height balanced K-D tree, Robinson proposed K-D-B-Tree [Rob81]. K-D-B tree combines the properties of K-D tree and B-tree.

K-D-B tree employs the same space division as K-D tree. But the height of the K-D-B tree is balanced. However, if a node is split, the split is propagated down to its children. This kind of forced split can cause empty or near-empty nodes. Pages are not guaranteed to be 50% full without very complicated record insertion and deletion algorithms [Sam90]. In [Rob81], 60% page occupancy is reported for uniformly distributed data sets at low dimensions. However, how to achieve high disk page utilization for large dimensionality with skewed data distributions remains a challenge for K-D-B tree [Kat97].

The hB-tree [Lom90] is a multi-dimensional point indexing method that guarantees good storage utilization. Its inner-node and growth processes are precisely analogous to the corresponding processes in B-trees. The efficiency of the hB-tree compared to R-tree family variations such as SR-tree and X-tree remains an open problem.

2.2 R-Tree family of Index Structure

The indexes discussed so far all use spatial division techniques, which guarantee that the regions associated with sibling nodes do not overlap. Gutman proposed the R-tree [Gut84] structure that allows overlap among siblings to maintain a balanced tree. The SR-tree [Kat97] and X-tree [Ber96] are two of the newest developments in R-tree family. The major drawback of the R-tree can be the overlap among sibling nodes. The SR-tree uses both a minimum bounding box and a minimum bounding sphere at each node. The region that corresponds to a node in an SR-tree is the intersection of the minimum bounding box and the minimum bounding sphere associated with that node. Thus, the overlapping area between two sibling nodes is reduced, particularly in high dimensions. Consequently, the search time improves significantly. However, the storage required for the SR-tree is quite large because it has to store both the minimum bounding boxes and minimum bounding spheres. For the same reason, the structure creation time and update time are impacted.

The X-tree [Ber96] is based on the R^*-tree [Bec90]. The major difference between these two index structures is that when a node needs to be split, the R*-tree always splits the node according to some heuristics. However, in the X-tree, if no good split can be found, (e.g., all splits result in relatively large overlap) then the node remains unsplit and it is called a supernode. Therefore, the overlapping area is reduced. In low dimensional space, an X-tree has similar performance as a R^*-tree. In high dimensional space, since the number of supernodes increases, the X-tree outperforms the R*-tree significantly. However, the number of children in a supernode is not bounded and the search time on a supernode is at least linear with the number of children it has. Therefore, the search time can be impacted. In Section 7, we present empirical data comparing the performance of the SR-tree and the X-tree with the PK-tree.

Another attempt [Kan98] to solve the problems incurred due to high dimensionality is to reduce dimensionality by transformation (e.g. Principle Component Analysis). The index structure is built on transformed data. However, this scheme can not guarantee the correctness of the results for K nearest neighbor and range queries. The precision varies from 30% to 90% depending on the dimensionality, database size, query specification, and so on. The reason is that due to the transformation, a certain amount of information is lost and the original distance between data points is not preserved precisely.

3 Definition of PK-tree

In this section, we formally define the structure of a PK-tree. Assume a given set of m-dimensional points such that all points are in a given rectangular area, C_0. (If the spatial area of interest is not a rectangular shape, then C_0 is the minimum bounding rectangle of the area of interest.[1]) Consider a recursive rectilinear[2] division of C_0 into successively smaller rectangular cells. We divide the cell at level 0 into several disjoint subcells at level 1, these cells at level 1 are each divided into disjoint subcells at level 2, and so on. The division is independent of the data set. The recursive subdivision used in constructing the PR-quadtree [Sam90] is an example of this kind of recursive division. Division on each level could be different. For example, one can divide on dimension 1 and 2 at level i and divide on dimension 3 at level $i+1$. For

[1] The development can be easily extended to the case that the spatial area of interest is dynamically defined as data points are added, but due to space limitations, we will not address this issue here.

[2] This is for simplicity of exposition. Many of the properties of the PK-tree are independent of this assumption.

the spatial area C_0, we thus have different sets of cells at different levels, with cells at level $i+1$ nested within cells at level i. The higher the level is, the smaller the cell size is. The length of the side of a cell along dimension j at level i is denoted by L_j^i, and $\forall i > 0$, $L_j^i = \frac{L_j^{i-1}}{r_j^i}$ where $r_j^i \geq 1$ is a constant integer. We call r_j^i the dividing ratio of level i along dimension j. For the same level, the dividing ratios along different dimensions may be different. For instance, if each cell at level i is trisected along dimension 1 but not dimension 2, then $r_1^i = 3$ and $r_2^i = 1$. Let $R^i = (r_1^i, r_2^i, \ldots, r_m^i)$ and $R = \cup_i \{R^i\}$ be the dividing ratio tuple at level i and the overall dividing ratio tuple respectively, where m is the number of dimensions. We also require that $\forall i > 0$, $r_1^i \times r_2^i \times \ldots \times r_m^i > 1$, i.e., a cell has to be partitioned on at least one dimension at each level. Let $r_i = r_1^i \times r_2^i \times \ldots \times r_m^i > 1$ be the fanout factor at level i. Then each cell at level i is always divided into r_i cells at level $i+1$. Let $r_{max} = \max_i(r_i)$ and $r_{min} = \min_i(r_i)$ be the maximum and minimum fanout factor over all levels, respectively. Since all $r_j^i > 1$ are constant integers, r_{max} and r_{min} are also constant integers and $r_{max} \geq r_{min} \geq 2$. The cells at two successive levels are illustrated in Figure 1 for the two dimensional case.

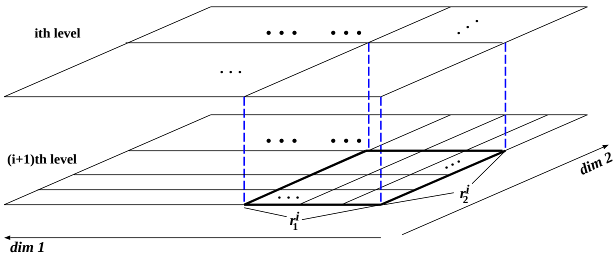

Figure 1: Multiple level division of the index space

Looking ahead, the PK-tree will have nodes, each of which corresponds to a cell at some level. The parent-child relationship corresponds naturally to the cell-subcell relationship. Let L_D be the minimum level such that all points in a set D are in separate cells. If all cells to level L_D are included in such a tree, the number of nodes is very large and the structure will tend to be inefficient. The PR-quadtree [Sam90] does not include subtrees of nodes that contain only one point (or less than some specified number of points). However, the non-leaf nodes that remain can have as little as one child. The "sparsely filled" nodes occupy space and increase the cost of queries unnecessarily. A PK-tree basically applies a simple rule to eliminate nodes with relatively few children from the tree in a such way that the result is an efficient tree structure with a number of desirable properties, including: uniqueness independent of the order of insertions and deletions, bounded number of children, non-overlapping of sibling cells and a bound on the average height.

Definition of K-instantiable Cells

The concept of K-instantiable cells is fundamental to the definition of PK-trees. We begin with some basic definitions which lead, by the end of this section, to the definition of K-instantiable cells and the PK-tree.

For any data point d, we use $(d_1, d_2, \ldots, d_m)$ to denote its location where d_j is the coordinate of d along dimension j. Let C_0 be the rectangular region containing all possible data points, $C_0 = \{d | \forall j (1 \leq j \leq m) :$

$l_j \leq d_j < u_j\}$ where l_j and u_j are the predefined minimum and maximum coordinates along dimension j. We call C_0 the *index space.*

Definition 3.1 *Given an index space C_0, a finite set of points D and dividing ratio tuple R. Let L_D denote the minimum level such that each cell at level L_D contains at most one point (or multiple collocated points). Each cell at level L_D which contains a point will be referred to as a* point cell.

Definition 3.2 *Let C be a cell in m dimensional space and $[C_1^{min}, C_1^{max}], [C_2^{min}, C_2^{max}], \ldots, [C_m^{min}, C_m^{max}]$ be the projection of this cell onto each dimension, where $C_j^{min} \leq C_j^{max}$, $\forall j = 1, 2, \ldots, m$. For any data point $d \in C_0$, we say $d \in C$ iff $C_j^{min} \leq d_j < C_j^{max}$, $\forall j = 1, 2, \ldots, m$. Otherwise, $d \notin C$.*

Each cell can be viewed as the set of points contained in it and set notation is used to express relationships among cells. For example, we say $C_1 \subseteq C_2$ iff $\forall d \in C_0 (d \in C_1 \Longrightarrow d \in C_2)$. We also say that C_1 is a sub-cell of C_2 (or C_2 is a super-cell of C_1) if $C_1 \subseteq C_2$ and C_1 is a proper sub-cell of C_2 (or C_2 is a proper super-cell of C_1) if $C_1 \subset C_2$. Let D be the set of all data points in an application, $D \subset C_0$. The cardinality of D is finite and denoted by N.

The following definition is not easy to grasp intuitively but an example is given immediately following the definition to illustrate the concept.

Definition 3.3 *Given a finite set of points $D \subset C_0$ and a dividing ratio tuple R, a cell C is K-instantiable $(K > 1)$ iff*

1. *C is a point cell, or*
2. *$\nexists$ $K-1$ K-instantiable cells $C_1, \ldots, C_{K-1} \subset C$, such that $\forall d \in D (d \in C \Longrightarrow d \in \bigcup_{i=1}^{K-1} C_i)$.*

Consider the example illustrated in Figure 2. In this case the space is two dimensional, the dividing ratios are $r_1^i = 2$ and $r_2^i = 2$ for all levels in Figure 2(d)(e), and we choose to set $K = 3$ in Figure 2(d) and $K = 5$ in FIgure 2(e). In Figure 2(a) $L_D = 3$ and the cells containing the "dots" are the point cells which, by definition, are all K-instantiable for any K. Of the cells at level two, A is not 3-instantiable since the points contained in A can be covered by only one subcell (namely $a2$) but B is 3-instantiable since the points it contains cannot be covered by less than three 3-instantiable subcells. In Figure 2(f), the tree for the same data set is built with $K = 5$ and the dividing ratios $r_1^i = 2$ and $r_2^i = 1$ when i is odd and $r_1^i = 1$ and $r_2^i = 2$ when i is even.

Theorem 3.1 *For a given area C_0, data set D, and dividing ratio tuple R, the set of K-instantiable cells is unique for any given value of K.*

Proof. Assume that there exist two distinct sets of K-instantiable cells, S_1 and S_2 ($S_1 \neq S_2$), for the same data set D. Let l be the highest (largest numbered) level at which the cells of S_1 do not match the

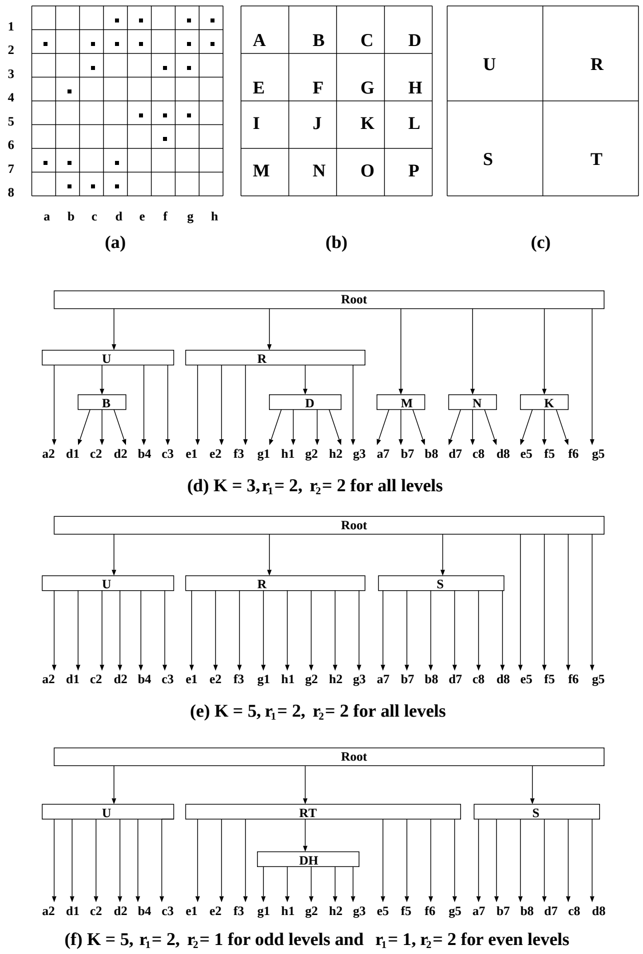

Figure 2: Examples of PK-trees

cells of S_2. Without loss of generality, let $C_i \in S_1$, and $C_i \notin S_2$. If C_i is a point cell, then $C_i \notin S_2$ is impossible by definition. Therefore assume C_i is a intermediate cell. Since C_i is K-instantiable ($C_i \in S_1$), $\nexists K-1$ or less K-instantiable cells at higher levels that cover all points in C_i. However, the same must be true for S_2 because S_2 has the same K-instantiable cells at levels higher than l. This is a contradiction and therefore the theorem holds. □

Definition 3.4 *Given a finite set of points D over index space C_0 and dividing ratio tuple R, a PK-tree of rank K ($K > 1$) is defined as follows.*

1. *The cell at level 0 (corresponding to C_0) is always instantiated and serves as the root of the PK-tree.*
2. *Except for the root, every node in the PK-tree is mapped one-to-one to a K-instantiable cell.*
3. *For any two nodes C_1 and C_2 in the PK-tree, C_1 is a (immediate) child of C_2 (or C_2 is the parent of C_1) iff*
   (a) *C_1 is a proper sub-cell of C_2, i.e., $C_1 \subset C_2$, and*
   (b) *$\nexists$ node C_3 in the PK-tree such that $C_1 \subset C_3$ and $C_3 \subset C_2$.*

Theorem 3.2 (Existence and Uniqueness) *For a given finite set of points D of cardinality N, in an index space C_0, and dividing ratio tuple R, there exists a unique PK-tree of rank K ($K > 1$).*

Proof. Given C_0, R, and D the set of cells in levels 0 through L_D is uniquely determined. Theorem 3.1 showed that the set of K-instantiable cells is unique. Given that the tree structure (including the root) is then determined by set inclusion of the associated cells, it is clear that the PK-tree is uniquely determined. □

4 PK-tree Properties

Theorem 4.1 (Bounds on Node Outdegree)*For a given index space C_0 and maximum fanout factor r_{max}, each intermediate node in a PK-tree of rank K ($K > 1$) has at least K and at most $(K-1) \times r_{max}$ children.*

Proof. The lower bound follows immediately from the definition of K-instantiable cells.

Secondly, we prove that each intermediate node in an PK-tree of rank K can have at most $(K-1)\times r_{max}$ children by contradiction. Assume that there exists an intermediate cell C that has more than $(K-1)\times r_{max}$ children. Then, by the Pigeonhole Theorem, there exists at least one uninstantiated proper sub-cell Cs of C such that at least K of C's children, say $C_1, \ldots, C_K$, are K-instantiable proper sub-cells of Cs. Since Cs is not K-instantiable, there exist at most $K-1$ K-instantiable proper sub-cells of Cs, namely $Cs_1, \ldots, Cs_{K-1}$ which cover the points in Cs. Then there must exist one of these sub-sells of Cs, say Cs_j $(1 \leq j \leq K-1)$ and two cells from $C_1, \ldots, C_K$, say C_{k1} and C_{k2} $(1 \leq k1, k2 \leq K)$, such that both $C_{k1} \subseteq Cs_j$ and $C_{k2} \subseteq Cs_j$. Since C_{k1} and C_{k2} are disjoint, we have $C_{k1} \subset Cs_j$ and $C_{k2} \subset Cs_j$. Then C_{k1} and C_{k2} can not be children of C because Cs_j is an K-instantiable proper sub-cell of C and $C_{k1} \subset Cs_j$, $C_{k2} \subset Cs_j$. This contradicts the assumption that C_{k1} and C_{k2} are children of C. Therefore, we have shown that each intermediate node in an PK-tree of rank K has at most $(K-1) \times r_{max}$ children. □

Theorem 4.2 (Bounded Storage Space) *Any PK-tree of rank K for a finite set of points D, of cardinality N, index space C_0, and maximum fanout factor r_{max} has at least $N + \frac{N-1}{(K-1)\times r_{max}-1}$ and at most $N + \frac{N+K-2}{K-1}$ nodes.*

Proof. Let X be the number of non-leaf nodes in a PK-tree. Any non-leaf node is a parent of some nodes and any non-root node is a child of some node. Moreover, according to Theorem 4.1, each parent node has at least K children and at most $(K-1) \times r_{max}$ children. Therefore, we obtain the following inequality:

$$(X-1) \times K + 1 \leq X + N - 1 \leq X \times (K-1) \times r_{max}$$

Solving this equation, we find the range of the total nodes $X+N$: $N + \frac{N-1}{(K-1)\times r_{max}-1} \leq X + N \leq N + \frac{N+K-2}{K-1}$. Thus, the theorem holds. □

Expected Height of a PK-tree

Let $D = \{x_1, x_2, \ldots, x_N\}$ be a random set of N points in a space C_0. $P_D(x_1, x_2, \ldots, x_N)$ is the probability distribution for D. We construct the PK-tree T from D and denote the height of T by $h(T)$. For a given C_0 and R, let $\wp(C_0, K, R)$ be the set of cells in the corresponding pyramid structure. Let C_1, C_2, ..., C_i, ... be a *cell sequence* where $C_1 \supset C_2 \supset \ldots \supset C_i \supset \ldots$ in which C_i is a cell at level i of $\wp(C_0, K, R)$. Let $P_i = P(d \in C_{i+1} \mid d \in C_i)$ be the probability that a random point $d \in D$ is in C_{i+1} given that d is in C_i. Given a *cell sequence* $C_1, C_2, \ldots$, the corresponding *path sequence* ρ is $C_1, C_2, \ldots, C_\pi$ where π is the smallest integer such that $P_\pi = 0$ if there exists such a π; otherwise, the corresponding path sequence is the cell sequence. Clearly, if $P(d \in C_{\pi+1} \mid d \in C_\pi) = 0$, then $C_{\pi+1}$ and all its subcells are empty and hence are not K-instantiable. Given a path sequence $\rho = C_1, C_2, \ldots$, the corresponding *sub-path sequence* ρ' is a subsequence[3] of ρ such that each C_i is an element in ρ' iff $P_i < 1$. The motivation for defining the sub-path sequence is to eliminate from the path the cells that would not be K-instantiable; any cell for which $P_i = 1$ only contains points in one subcell at the next level and is never K-instantiable. For a sub-path sequence ρ', let $\psi_{\rho'} = \max(P_i)$ for all cells C_i in ρ' be the probability bound on ρ'. Let $\Psi = \max(\psi_{\rho'})$ be the overall probability bound of all sub-path sequence ρ'. (For example, for a uniform spatial distribution of points, $\Psi = \frac{1}{r_{min}}$). To simplify the discussion here, we assume that there exists a small constant $\epsilon > 0$ such that $\Psi \leq 1 - \epsilon$. However, all results still hold without this assumption. The proof of the general case can be found in [Yan97].

Property .1 *Let D be a set of N points as defined above with parameter $\Psi \leq 1 - \epsilon$ and T be the PK-tree for D with rank K. Let $h(T)$ be a random variable equal to the height of T. Then $P(h > H) = O\left(\Psi^{\log^2 N}\right)$ where $H = \Omega(\log N)$, $H > 4 \log N$ and Ψ is the probability bound defined above.*

Property .2 *The expected height of the PK-tree for a set D of N data points with parameter $\Psi \leq 1 - \epsilon$ is $O(\log N)$.*

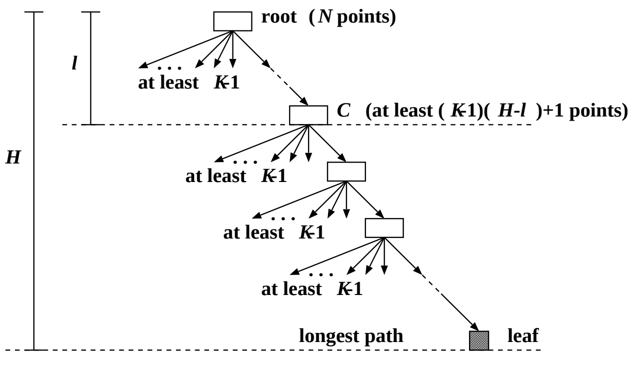

Figure 3: Expected Height of PK-tree

Intuitively, for any node C on the longest path of the PK-tree, if the height of the subtree rooted at C is $H - l$, then there are at least $(H - l)(K - 1) + 1$ leaf nodes in such a subtree. The reason is that

[3]Given a sequence, a subsequence contains a subset of elements from the original sequence in the same order, but not necessarily consecutive in the original sequence.

there are at least $(H-l)$ intermediate nodes in such a subtree and each intermediate node has at least K children. Then for each node on the longest path, there are at least $K-1$ out-branches besides the one in the longest path. Since each out-branch has at least one leaf node as its descendant, there must be at least $(H-l)(K-1)+1$ leaf nodes in this subtree. Therefore, at least $(H-l)(K-1)+1$ data points have to be located within C in order to make the height of the tree be H. This is illustrated in Figure 3.

Since there are at least l nodes on the path from the root to C and $\Psi \leq 1-\epsilon$ is the upper bound of the probability $P(d \in C_{k1} \mid d \in C_{k2})$ where $C_{k1} \subset C_{k2}$ are two consecutive cells in a sub-path sequence, the probability of a data point to be in C is at most Ψ^l. If we choose H and l such that $l > 3 \log N$ and $H - l \geq \log N$, by Chernoff Bound [Mot97], the probability that there exists such a cell C containing at least $(H-l)(K-1)+1$ points decreases exponentially as H increases.[4]

Sufficient for $h(T) = O(\log N)$

Since from the definition, it is not intuitive how general $\Psi \leq 1-\epsilon$ is, the above results can be difficult to apply directly in a real situation. However, we are developing simpler, sufficient conditions to guarantee $\Psi \leq 1-\epsilon$ (and therefore, $E(h) = O(\log N)$) that should be more easily verified in practice. Two examples are described next. Interested readers can refer to [Wan97] for more examples.

(1) M-Level Clustering Spatial Distributions

We begin by describing 0-level clustering spatial distributions. A distribution of a set of points over C_0 is said to satisfy 0-level clustering if each point in D is an independent sample from the uniform distribution over C_0. As mentioned previously, $\Psi = \frac{1}{r_{min}} \leq \frac{1}{2} = 1-\epsilon$ where $\epsilon = \frac{1}{2}$ in this case, thus the height of the PK-tree is $O(\log N)$ if the data set is from a 0-level clustering distribution. A distribution satisfying the 1-

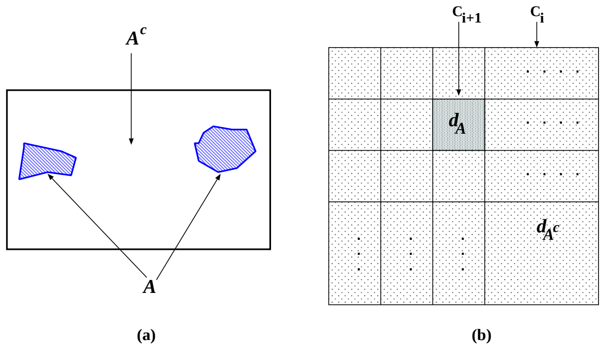

Figure 4: M-Level Clustering

level clustering constraint is defined as follows. Let $A \subset C_0$ be some subset of C_0 and let $A^c = C_0 - A$. The distribution for points in A and A^c satisfies 0-level clustering constraint, i.e., the points in each region are uniformly and independently distributed over the region. Let D_A and D_{A^c} be the subsets of points in regions of A and A^c, respectively. The cardinality of these two subsets are denoted by N_A and N_{A^c}, respectively

[4]The complete formal proofs of the above properties are in [Yan97].

and $N = N_A + N_{A^c}$. Also denote the density in A and A^c by d_A and d_{A^c}, respectively. The 1-level clustering constraint requires that $\frac{Area(A)}{Area(A^c)}$ be independent of N. An 1-level clustering spatial distribution in 2 dimensional space is illustrated in Figure 4(a). The conditional probability $P(d \in C_{i+1} \mid d \in C_i)$ for $C_{i+1} \subset C_i$ is equal to the expected number of points in C_{i+1} divided by the expected number of points in C_i. Without loss of generality, assume $d_{A^c} < d_A$. The value of $P(d \in C_{i+1} \mid d \in C_i)$ is the greatest when i is the level with minimum fanout factor (i.e., a cell C_i of level i has r_{min} subcells at level $i+1$) and only one subcell C_{i+1} at level $i+1$ has density d_A while all the rest $r_{min} - 1$ subcells has density d_{A^c}. This scenario is illustrated in Figure 4(b) for 2 dimensional space. So $\Psi \leq \frac{d_A}{d_A + (r_{min}-1) \times d_{A^c}} \leq \frac{d_A}{d_A + d_{A^c}} = 1 - \epsilon$ where $\epsilon = \frac{d_{A^c}}{d_A + d_{A^c}} > 0$ when $d_{A^c} > 0$. Therefore, for $d_{A^c} > 0$, the height of the PK-tree is $O(\log N)$. For $d_{A^c} = 0$, we do not have such an easy bound on Ψ. Instead we resort to the device of adding N points uniformly distributed over A^c to the original data set D. Let D' be the new data set. The corresponding $\Psi' \leq \frac{d_A}{d_A + (r_{min}-1) \times d'_{A^c}}$ where $d'_{A^c} > 0$ is the new density in A^c and $\frac{d'_{A^c}}{d_A} = \frac{Area(A)}{Area(A^c)}$ which is independent of N. Thus the expected height of the PK-tree for D' is again $O(\log N)$. Since the height of a PK-tree never decreases when we insert points, the height of the PK-tree for D is at most the height of the PK-tree for D'. Therefore, the expected height of the PK-tree for a data set that satisfies the 1-level clustering constraint is $O(\log N)$.

The definition of M-level clustering distribution is a direct extension of the definition of 1-level clustering. So the expected height of a PK-tree for a data set which has M-level clusters is $O(\log N)$ if M is bounded by a constant.

(2) Bounded Intensity Constraint

Intensity function, $\lambda(B)$, where B is any subset of C_0, is defined as the probability that a point is in B [Sto94]. Data set D consists of N independent samples and $\lambda(B)$ is independent of N for all B. Let I_{glb} and I_{lub} be the greatest lower bound and the least upper bound of the intensity for all B. The bounded intensity constraint requires that $I_{glb} > 0$. In this case, $\Psi \leq \frac{I_{glb}}{I_{lub} + (r_{min}-1) \times I_{glb}} \leq \frac{I_{glb}}{I_{lub} + I_{glb}} = 1 - \epsilon$ where $\epsilon = \frac{I_{lub}}{I_{lub} + I_{glb}} > 0$ since $I_{lub} \geq I_{glb} > 0$. Therefore, the PK-tree for any data set that satisfies the bounded intensity constraint has an expected height equal to $O(\log N)$.

5 Pagination of the PK-tree

In this section we address the issue of pagination of the structure. First, there are several parameters that can be chosen to control the storage size of nodes and, not independently, the bounds on the outdegree of nodes. The storage required for a node is basically determined by the number of children the node has. As K increases, both the minimum and maximum number of children increases as well as the spread between the bounds. The product of the dividing ratios for each dimension gives the slope of the increase in the maximum value. For example with $r_1^i = 2$ and $r_2^i = 2$ for all levels i in 2 dimensional space the maximum number of children increases four times faster than the lower bound. If we want fixed size nodes, this would mean a growing amount of wasted space as the number of dimensions increased. There is however a technique which is used in the definition of the K-D tree which can be applied to advantage here. Instead of dividing the space along all dimensions at each successive level, a cell can be divided along a subset of dimensions at each level. By cycling through all dimensions we achieve basically the same effect in multiple levels that we previously achieved in one but we can dramatically reduce the maximum size of a node. For example, if the number of dimensions is m and $r^i_j = 2$ for $1 \leq j \leq m$ and all levels i, then the maximum number of children for a node is $2^m(K-1)$. If we only divide a cell along one dimension at a time at each successive level then the maximum number of children a node can have is $2(K-1)$. Note also that in this latter case the minimum number of children is K so the ratio of the maximum to the minimum is slightly less than 2 and approaches 2 as K increases.

We have considered two approaches to pagination of the PK-tree. The first is to pick the parameters to keep the nodes small and to then consider page packing heuristics. One problem with this approach is that there is no direct relationship between the structure of the PK-tree as defined and the links that cross page boundaries. The second approach is to pick the parameter K and the number of dimensions to split at each level such that the maximum size node is close to a page size and allocate one node to a page. In this case, the paginated structure is identical to the PK-tree and any properties we can prove about the PK-tree carry over in terms of pages, e.g., expected height measured in nodes is the expected height measured in pages. One concern is the possibility of wasting space if nodes only partially fill pages. To address this issue we choose to pick K large but to use spatial division on only one dimension at each level and with dividing ratio 2 at all levels. Then the number of children a node has is bounded by K on the low end and $2(K-1)$ on the high end. Letting ρ be the constant space overhead per node, τ be the storage cost per child link, and P be the page size, then we select $K = \lceil \frac{P-\rho}{\tau} \rceil$. The most serious memory overhead is due to some nodes having less than the maximum children. However, nodes are guaranteed to be nearly 50% occupied in the worst case. We will report on a large number of experiments later that indicate that space utilization is much better than 50% on average. In our experiments, the page size is 8KB and we use K varying from 18 (16 dimensions) to 86 (2 dimensions).

6 Operations on a PK-tree

Let d denote the data point that is to be inserted into T or deleted from T and let C_d be the point cell that contains d. The insertion of a data point d into the PK-tree T is achieved by inserting the corresponding leaf node C_d into T. This can be done in two phases: the first phase is to follow the path from the root T down to the leaf level to locate all (potential) ancestors of C_d while the second phase is from the leaf level back to the root along the same path to make all necessary changes (instantiate or deinstantiate cells) to maintain the tree as a PK-tree. The formal description of the insertion algorithm is in [Wan97]. The computational complexity of data point insertion is linearly proportional to the height of the PK-tree. (Since the number of children of each node is bounded, the complexity of the algorithm at each level is constant.) Since the average height of the PK-tree is $O(\log N)$, the average computational complexity is $O(\log N)$ and the worst case complexity is $O(N)$ where N is the number of data points in the PK-tree. The deletion procedure is similar to insertion. The detailed algorithm is presented in [Wan97].

K-nearest-neighbor query and range query are common types of queries operated on a spatial index. Since these query algorithms are similar that used in PR-Quad trees, we will not discuss them here. Both algorithms are in [Wan97]. In addition, the PK-tree can be generated from sorted data efficiently ($O(N)$ complexity). Due to the limit of space, we will not present this efficient loading algorithm in this paper. Interested readers can refer to [Yan97].

7 Concurrency Control

Here we present a scheme which allows multiple users to access the index concurrently even when multiple writers are active. The basic insertion and deletion algorithms were presented in the previous section. These operations involve two phases: searching down the tree to locate the leaf node for updating and then retracking the path to the root for instantiating or deinstantiating internal nodes. Nodes could be deinstantiated or split (i.e., some of its children become children of a newly instantiated child node) during updating. These nodes are referred as *invalid* nodes if a deinstantiation or split were performed since the last time the user obtained their addresses. Every time an invalid node is confronted during traversal (up or down), the user needs to release the lock he acquired on this invalid node and backtrack to the first "valid" ancestor (or the root) and traverse down again. Here we expand the algorithms to permit maximum concurrent access in updating or searching the tree. The scheme we present is a modified version of the one proposed by Kanth et al. [Kan97]. We assume the use of a separate reference counter on each node[5] to keep track of the number of pointers either in the PK-tree or held in local variables by a client process to the associated node. (This means that the client has to explicitly increment and decrement the counter as it copies a pointer into its local memory or "deletes" such a pointer, respectively.) Once the counter becomes 0, we can garbage collect the space occupied by the node. When a client reads a node and finds its children's addresses and determines which children's addresses he needs to record, he increases the reference counters for those children. After he finishes his work and is about to discard a reference to a node, he needs to decrease the reference counter associated with that node. The increment and decrement operations on the reference counters should be atomic. A page is not reused until garbage collected. The root will never be deinstantiated or moved. (The reference counter of the root is set to 1 initially by some external daemon that will never decrease it.)

In addition, there is a sequence number (sn) in each node and a global sequence number (gsn) which is always greater than or equal to the current maximum sn of any node in the tree at any instant. The sequence number is always maintained (even after the node is deinstantiated) until the space is garbage collected. Every time a node is deinstantiated or split, gsn is increased by 1 and the sn of the node(s) involved is set to be the value of gsn. This enables the user to tell whether the node has become invalid since the address was obtained in such a way that the identity of the node has to be recalculated. For example, in Figure 6(d) the sn of both C and D' are set to the new value of gsn while in Figure 6(g) the sn in C needs to be set to the new value of gsn.

When a user traverses down the tree during a search, he always records the gsn value at the time when he obtains the address of a node. If an insertion or deletion is going to be performed, this user will store the pair (gsn, address) for each node visited during the downward traversal. This information provides a way to decide whether a node becomes invalid after its address was obtained. Basically, the user just needs to compare the value of gsn he recorded and the sn of the node. If the former is less than the latter, then this node is invalid. Otherwise, this node is valid. Every time after acquiring a shared mode lock on

[5] The pagination scheme is used such that there is one node per page.

a node, he checks whether the node is still valid or not. If so, the process continues. Otherwise, he needs to release this shared mode lock and backtrack to an valid ancestor (or the root) and traverse down from this ancestor again. This is always doable since the root is never deinstantiated. The worst case scenario is when the user has to go back to the root and traverse down again. Every time he finishes examining a node and obtains address(es) of its child(ren), he also records the value of gsn again before he releases the lock on this node. This procedure is repeated until a leaf node is reached. The number of shared mode locks a user holds at any time is at most 1. The use of gsn and sn avoids holding shared mode locks on multiple levels. Figure 5 shows an example of the locking sequence for data retrieval.

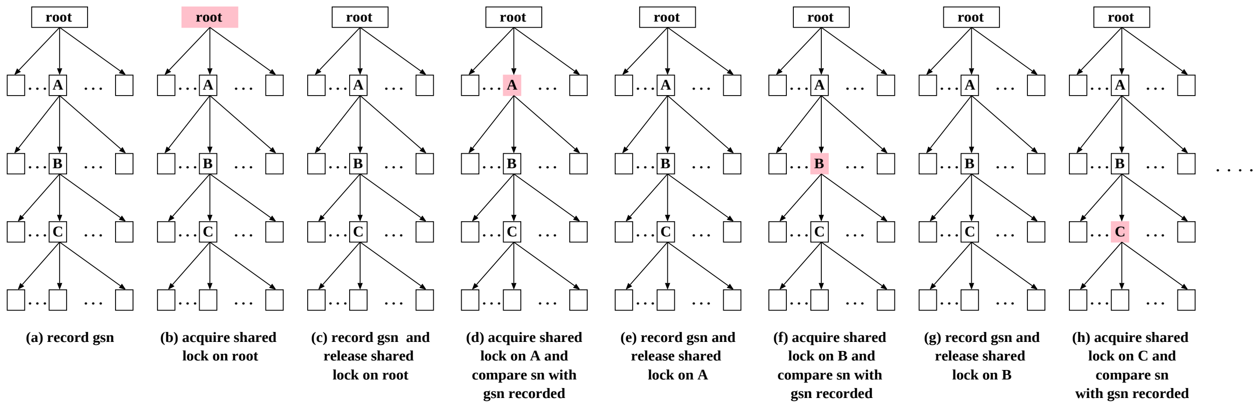

Figure 5: Consistency Control of Retrieval

Before traversing up the tree, the last shared mode lock acquired during the downwards traversal has to be released. During the upwards traversal, the updates caused by the insertion (or deletion) of a point is an alternating sequence of instantiating and deinstantiating nodes in a bottom-up manner which ends as soon as a node is reached which does not have to be modified. A node overflow will trigger instantiating a new node as its child and hence reduce the number of children of the overflowing node which, in turn, may only cause underflow of this node. This means that overflow and underflow occur alternately. Due to this fact, the user only needs to acquire exclusive mode locks on at most 2 nodes (a node and its parent) at any time. The reason is that since an overflow is followed by an underflow, there is always space to hold the pointer to the new child. Thus, splitting a node never needs to be postponed until splitting of its ancestors has been completed (as is required for example in a B-tree). For example, when instantiating D' (Figure 6(d)), part of C's children become children of D' and D' becomes C's child. It is obvious that C has fewer children after instantiating D'. So, making D' a new child of C will never cause C to overflow. The only consequence is that C could underflow. Every time a user finishes updating a child of a node X, he can release the lock on the child and then acquire an exclusive mode lock on X's parent. For example, the user releases lock on D' first and then acquire lock on B as shown in Figure 6. After the user acquires the exclusive mode lock on the parent, he also checks whether the parent is still valid. If the parent is invalid, the user needs to release the exclusive mode lock on the parent and find the first valid ancestor (or the root) and traverse down to locate the current parent (in the same way as the downwards traversal described previously in this section) as well as update the stored information (gsn, address). Otherwise, the process continues. Note that in the case that the parent became invalid, the node is still in the PK-tree since the user still holds an exclusive mode lock on this node. This means that the node must have a parent somewhere else in the PK-tree. What the user needs to do is to locate the new parent. Figure 6 shows an example of a locking sequence during update.

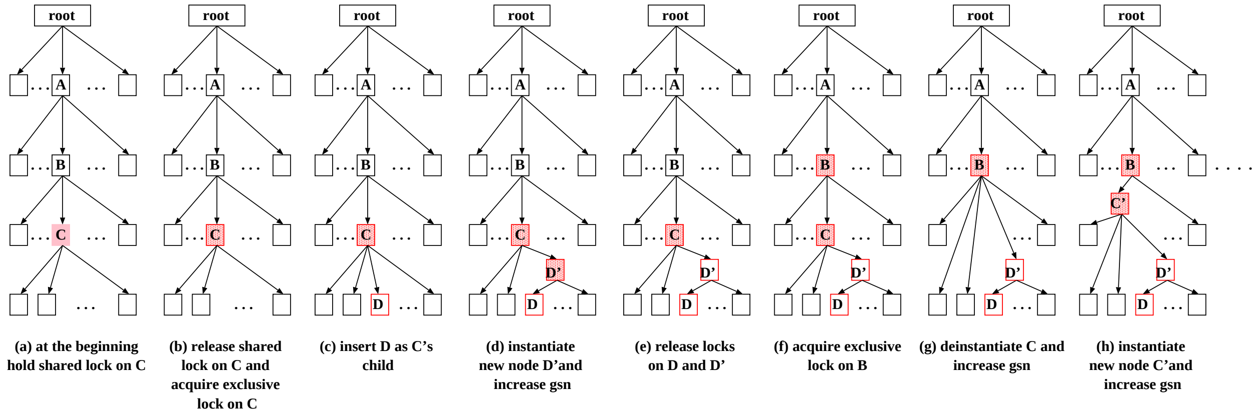

Figure 6: Consistency Control of Insertion

The shared mode lock and exclusive mode lock are used during different traversal directions. When traversing down the tree, a user holds at most one lock at a time, when traversing up the tree the user may hold up to 2 locks at a time. So, a user can hold at most 2 locks at any time. Thus this lock scheme allows a large number of users to concurrently access the same index structure.

In addition, this scheme avoids deadlock which can be shown as follows. Assume that we are building a wait-for graph. When a user traverses down the tree, he or she always releases a shared mode lock before acquiring another shared mode lock. So, during a traversal down the tree a user creates no arcs in the wait-for graph. When traversing up the tree, the exclusive mode locks on nodes are acquired in bottom-up order. All arcs added to the wait-for graph are in one direction: from a node to its ancestors. Therefore, no cycles can exist in the wait-for graph and thus this scheme is deadlock free.

8 Performance Comparison of PK-tree, SR-tree, and X-tree

8.1 Experimental Setup

All the performance tests in this section were done on a SPARC-10 workstation (SunOS 5.5) with 208 MB main memory and a local disk with 9GByte capacity. We chose to compare PK-tree with X-tree and SR-tree because the X-tree and SR-tree are among the most recent proposed dynamic spatial indexing structures and they are reported to have significant performance improvement over R*-tree, K-D-B tree, and so on [Kat97][Ber96]. For the K-Nearest-Neighbor query and Range query, we choose to use no cache for PK-tree, SR-tree, X-tree. We use the version of X-tree source code given to us by the authors of [Ber96] recently, and the SR-tree source code from the authors of [Kat97].

For the PK-tree, in order to fit one node in one 8 KB disk block and keep the utilization level high, we fix the dividing ratio to be 2 at all levels, and round-robin through the dimensions, one dimension at each level. We vary the rank of the PK-tree for different dimensionality so that the maximum number of children, i.e. $2(K-1)$, will fill an 8 KB page as tightly as possible. In these performance tests, we are mainly concerned with K-Nearest-Neighbor (KNN) and Range Queries. We choose $K = 25$ for all KNN queries while we vary the distance in the range query so that around 10 points will be included.

It is known that performance of a spatial index structure can vary dramatically as a function of data distribution. Therefore, we used four set of data for the performance test. There are three synthetic data sets and one real data set. The synthetic data sets include: uniform distribution, two kinds of clustered data distribution. Each synthetic data set consists of one hundred thousand data points. The results of the performance tests are discussed in the following sections.

Dimension	2	4	8	16	32	64
PK-tree (u)	4	4	5	6	7	9
PK-tree (c1)	5	7	7	6	7	8
PK-tree (c2)	7	7	6	7	8	9
X-tree	4	4	4	4	5	6
SR-tree	4	4	5	5	6	7

Table 1: The Height (maximum levels) of Indexes with 100,000 points

Dimension	2			4			8			16		
	u	c1	c2	u	c1	c2	u	c1	c2	u	c1	c2
PK-tree	1.8	1.9	1.9	2.8	2.8	2.8	4.9	4.8	4.9	9.4	9.3	9.4
X-tree	1.8	1.8	1.8	3.0	3.0	3.0	5.6	5.5	5.6	10.7	10.4	10.6
SR-tree	69	70	70	74	73	74	74	74	75	90	91	92

Table 2: The size of Indexes in Megabytes with 100,000 points

8.2 Uniformly Distributed Data Set

First we generated uniformly distributed data sets for 2, 4, 8, 16, 32, and 64 dimensions. Each data set consists of 100,000 data points. The height of the indexes up to 64 dimensions are shown in Table 1. (Uniformly distributed data is denoted by "u".) Since PK-trees are not height balanced, they are a little bit taller than the other two. But there is no overlap among any sibling nodes in a PK-tree, which reduces search time dramatically. X-tree is shorter than the SR-tree because the X-tree has larger fan-out than SR-tree [Kat97]. The size of the index structures are given in Table 2. Moreover, PK-tree has the smallest structure size and a simple generation rule, the generation time of PK-tree is much smaller than SR-tree and X-tree. SR-tree has the largest size due to the fact that it stores more information than the other two indexes. Figure 7 illustrates the average CPU time and the average number of I/O for KNN queries for different dimensionalities. For each dimensionality, we randomly chose 20 points and executed the KNN query on the same set of 20 points for all three data structures. For low dimensionalities, the number of nodes fetched for a KNN query is very small for all three indexes. For high dimensionalities, since the data is uniformly distributed, then for a random point d, there is a large number of points approximately the same distance from d. Therefore, a large percentage of the index needs to be searched. For example, with 16 dimensions, about 80% to 90% of the disk blocks in the PK-tree and the X-tree are fetched while about 60% of the disk blocks of the SR-tree are fetched. Since the SR-tree occupies a much larger amount of space, the overall number of I/O in a SR-tree is much larger than the other two. X-tree has a similar number of I/O as PK-tree, but the CPU time used by the X-tree is much larger because the number of supernodes in X-tree increases with the dimensionality [Ber96] and the number of children in a supernode is

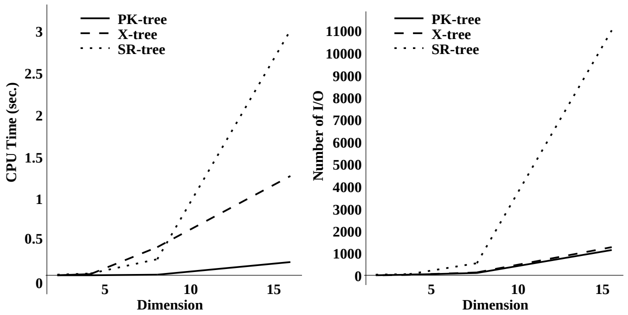

Figure 7: KNN Query Performance for Uniform Data Distribution

not bounded. Therefore, the CPU time consumed in X-tree search for appropriate children nodes increases at a much faster pace than the PK-tree.

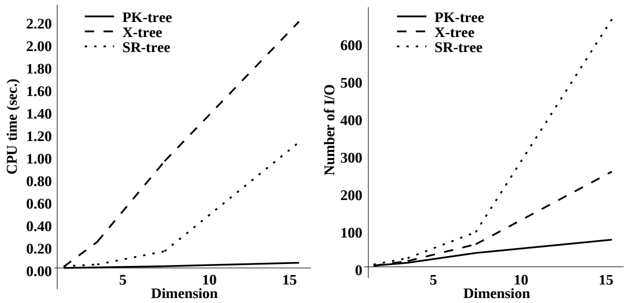

Figure 8: Range Query Performance for Uniform Data Distribution

Figure 8 shows the average CPU time and the expected number of I/O for a range query in each dimensionality. We also randomly choose 20 points at each dimensionality, and the average performance is reported. Katayama and Satoh have shown that fan-out is a problem in the SR-tree [Kat97]. For example, for sixteen dimensions, an internal node of a SR-tree has 20 children on average while the PK-tree has an average of 30 children in an internal node (for uniform distribution). Therefore, the number of I/O is large even in the case that only a small percentage of the structure is fetched. On the other hand, since the overlap among siblings in an X-tree grows with the dimensionality, the average number of I/O increases because multiple paths have to be searched.

8.3 Clustered Distribution

Skewed data distributions are common place in many real applications. To explore this issue, we generate two sets of skewed data: flat cluster (c1) and sharp cluster (c2).

- c1: 20% of data (20,000 points) are uniformly distributed in space and 80% of data (80,000 points) are distributed in three disjoint clusters. Within each cluster, the coordinate of data points along each dimension is normally distributed with the mean at the center of the cluster and the standard deviation 0.05.

- c2: 20% of data are uniformly distributed in space and 80% of data are distributed in fifteen disjoint clusters. Within each cluster, the coordinate of data points along each dimension is normally distributed with the mean at the center of the cluster and the standard deviation 0.025.

The heights of the PK-tree, X-tree, and SR-tree are listed in Table 1 while the size of each index is listed in Table 2. From Table 1, it is clear that with different data distribution, the height of a PK-tree does not vary significantly. The PK-tree only instantiates the set of nodes with at least K children, and hence it is guaranteed that each disk block has at least 50% utilization when the dividing ration on each level is chosen as 2. (The average disk utilization level was actually observed to be well above 80% for the three synthetic and one real data sets.) Thus, the index structure size does not vary significantly with the data distribution. For the KNN and range queries, we found that the average CPU time and the expected

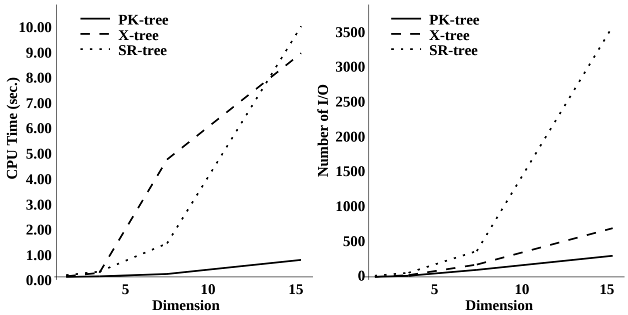

Figure 9: KNN Query Performance for Clustered Data Distribution

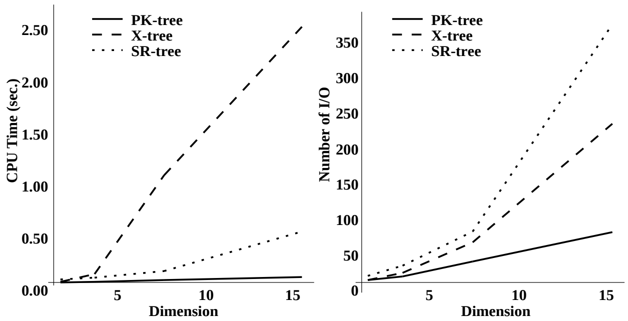

Figure 10: Range Query Performance for Clustered Data Distribution

number of I/O are similar for the c1 and c2 data sets. Due to space limitations, we only present the performance on the c2 data set. Figures 9 and 10 show the performance of the KNN and range queries on the c2 data set, respectively. If a KNN query is performed on a point within a cluster, then many points outside the cluster are quickly eliminated. If a KNN query is performed on a point outside any cluster, then depending on the data location, the performance may vary significantly. We randomly chose 20 points for each query. The average number of I/O and CPU time are low, for all three indexes, compared with uniformly distributed data set. Since the PK-tree employs disjoint spatial division, it can take most advantage of this kind of clustered data set and it has the lowest expected number of I/O and average CPU time.

8.4 Real Data Set

We chose the NASA Sky Telescope data as an example real data set. (We have run similar test on real data sets such as the Sequoia 2000 benchmarks [Sto93]. The results were similar to those for the NASA data set.) The NASA data set consists of 200,000 two-dimensional points (they are the coordinates of crater locations on the surface of Mars). Table 3 shows the average tree height, index structure size, the average CPU time, the expected number of I/O of KNN and Range query. Based on the test results presented

	height	size	KNN CPU	KNN I/O	RAN CPU	RAN I/O
PK-tree	5	3.7MB	4ms	4	3ms	4
X-tree	4	5.7 MB	90ms	4	10ms	4
SR-tree	5	120MB	28ms	8	14ms	6

Table 3: The Performance on NASA Sky Telescope Data Set

above, we can see that PK-tree outperforms the X-tree and SR-tree by a large margin, especially in high dimensionalities because the overlap among sibling nodes in an X-tree or an SR-tree increases significantly with dimensionality. On the other hand, there is no overlap among any sibling nodes in a PK-tree.

9 Conclusion

In this paper, we propose a new dynamic spatial indexing structure, called the PK-tree. In some sense, it can be conceived as a variation of PR quad-trees. However, it differs from all existing trees by employing a unique set of constraints to eliminate unnecessary nodes that can result from a skewed spatial distribution of objects. This ensures that the total number of nodes in a PK-tree is $O(N)$ and the average height of a PK-tree is $O(\log N)$ under some general conditions.

We also proved a set of properties of PK-trees. These properties include: non-overlapping of sibling nodes, uniqueness of the PK-tree for a given set of data points, and so on. Empirical studies shown that the PK-tree outperforms SR-tree and X-tree by a wide margin. Therefore, we see the PK-tree as a preferable indexing method.

References

[Abr95] M. Abrash. BSP Trees. *Dr. Dobbs Sourcebook*, 20(14), 49-52, May/June 1995.

[Bec90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD Conf. on Management of Data*, 322-331, 1990.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of ACM*, vol. 18, no. 9, pp. 509-517, 1975.

[Ber96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : an index structure for high-dimensional data. *Proc. 22nd Intl. Conf. on Very Large Data Bases (VLDB)*, 28-39, 1996.

[Cia97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: an efficient access method for similarity search in metric spaces. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 426-435, 1997.

[Fal88] Christos Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. on Software Engineering*, 14(10):1381-1393, 1988.

[Fal89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. *Proc. 8th ACM SIGART-SIGMOD Sym. on Principles of Database Systems (PODS)*, 247-252, 1989.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD Conf. on Management of Data*, 47-57, 1984.

[Jag90] H. V. Jagadish. Linear clustering of objects with multiple attributes. *Proc. ACM SIGMOD Conf. on Management of Data*, 332-342, 1990.

[Hen89] Andreas Henrich, Hanas-Werner Six, and Peter Widmayer. The LSD tree: spatial access to multi-dimensional point and non-point objects. *Proc. 15th Intl. Conf. on Very Large Data Bases (VLDB)*, 45-54, 1989.

[Kam94] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: an improved R-tree using fractals. *Proc. 20th Intl. Conf. on Very Large Data Bases (VLDB)*, 500-509, 1994.

[Kan97] K. V. Ravi Kanth, David Serena, and Ambuj K. Singh. Improved concurrency control techniques for multi-dimensional index structures. *Technical Report, Univ. of California at Santa Barbara*, 1997.

[Kan98] K. V. Ravi Kanth, Divyakant Agrawal, and Ambuj K. Singh. Dimensionality reduction for similarity searching in dynamic databases. *Proc. ACM SIGMOD Conf. on Management of Data*, 166-176, 1998.

[Kat97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. *Proc. ACM SIGMOD Conf. on Management of Data*, 369-380, 1997.

[Lin94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: an index structure for high-dimensional data. *VLDB Journal*, 3(4):517-542, 1994.

[Lom90] D. Lomet and B. Salzberg. The hB-tree: a multiattribute index method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625-658, 1990.

[Mot97] Rajeev Motwani. *Randomized Algorithms*, Cambridge University Press, 1997.

[Ore84] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. *Proc. 3rd ACM SIGART-SIGMOD Sym. on Principles of Database Systems (PODS)*, 181-190, 1984.

[Ore86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD Conf. on Management of Data*, 326-336, 1986.

[Ohs83] Yutaka Ohsawa and Masao Sakauchi. Multidimensional data management structure with Efficient Dynamic Characteristics. *Systems Computers Controls*, vol. 14, no.5, 1983.

[Rob81] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD Conf. on Management of Data*, 10-18, 1981.

[Sam90] Hanan Samet. The design and analysis of spatial data structures. *Addison-Wesley Publishing Company*, 1990.

[Sam90a] H. Samet. Efficient processing of window queries in the pyramid data structure. *9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 265-272, 1993.

[Sel87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: a dynamic index for multi-dimensional objects. *Proc. 13th Intl. Conf. on Very Large Data Bases (VLDB)*, 507-518, 1987.

[Sel97] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. Multidimensional access methods: trees have grown everywhere. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 13-14, 1997.

[Sto93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 storage benchmark. *Proc. 1993 ACM SIGMOD Int. Conf. Management of Data*, pp. 2-11, 1993.

[Sto94] D. Stoyan and H. Stoyan. *Fractals, Random Shapes and Point Fields.* John Wiley & Sons, 1994.

[Wan97] Wei Wang, Jiong Yang, and Richard Muntz. PK-tree: a dynamic spatial index structure for large data sets. *UCLA Computer Science Department Technical Report #970039*, 1997. http://dml.cs.ucla.edu/~weiwang/paper/TR-97039.ps

[Yan97] Jiong Yang, Wei Wang, and Richard Muntz. Yet another spatial indexing structure. *UCLA Computer Science Department Technical Report #970040*, 1997. http://dml.cs.ucla.edu/~weiwang/paper/TR-97040.ps

A Bound on the Height of PK-tree

Expected Height of PK-tree: **Property .1** Let D be a set of N points over an index space C_0 with distribution $P_D(x_1, x_2, \ldots, x_N)$ and parameter $\Psi \leq 1-\epsilon$ and T be the PK-tree for D with rank K Let $h(T)$ be a random variable equal to the height of T. Then $P(h > H) = O\left(\Psi^{\log^2 N}\right)$ where $H = \Omega(\log N)$, $H > 4 \log N$ and Ψ is the overall probability bound.

Proof. For any node C in a PK-tree, let h_C be the height of the subtree T_C rooted at C. If C is a point cell, then $h_C = 0$. Let Y_i and Y_l be the number of intermediate nodes and leaves in T_C, respectively. Then, $Y_i \geq h_C$. The total number of nodes in T_C is at least $Y_i \times K + 1$ since each intermediate node has at least K children. We have $Y_l + Y_i \geq Y_i \times K + 1$ and hence $Y_l \geq Y_i \times (K-1) + 1 \geq h_C \times (K-1) + 1$. Therefore, there are at least $h_C \times (K-1) + 1$ data points located in C. Now we consider the probability that the height of the PK-tree is greater than H. For a sub-path sequence ρ' and an integer $l > 0$, the lth element of ρ' is denoted by $C(\rho', l)$. If $C(\rho', l)$ is a node of the longest path of the PK-tree T, then the height of the subtree rooted at $C(\rho', l)$ has to be at least $H - l$ in order to make the height of the whole tree greater than H. This means that there must be at least $(H-l)(K-1)+1$ data points located in $C(\rho', l)$. Since we have $P(d \in C(\rho', i+1) \mid C(\rho', i)) \leq \Psi$, $\forall i \geq 0$ by definition, the probability that a data point d is located in $C(\rho', l)$ is at most Ψ^l. Let X be the number of data points in $C(\rho', l)$ and $p[X > (K-1) \times (H-l)]$ be the probability that at least $(K-1)\times(H-l)+1$ data points are in $C(\rho',l)$. Let μ be the expected number of points in $C(\rho',l)$. We choose l such that $l > 3\log N$ and $H-l \geq \log N$. Let $\delta = \frac{(K-1)\times(H-l)}{\mu} - 1$, so $p[X > (K-1)\times(H-l)] = p[X > (1+\delta)\times\mu]$. By Chernoff Bound [Mot97], we have the following equality: Since $\mu \leq N \times \Psi^l = N \times \Psi^{(3\log N)} \simeq \frac{c_2}{N^2}$ for some constant c_2 when $\Psi < 1$. Therefore we have

$$
\begin{array}{ccccc}
\left(\frac{e}{1+\delta}\right)^{\mu\delta} & = & \left(\frac{e\times\mu}{(K-1)\times(H-l)}\right)^{(K-1)(H-l)} & < & \left(\frac{e\times N\times\Psi^{3\log N}}{\log N}\right)^{\log N} \\
 & < & \left(\frac{\Psi^{\log N}}{\log N}\right)^{\log N} & < & \frac{\Psi^{\log^2 N}}{N}
\end{array}
$$

Then the probability that there exists a $C(\rho',l)$ that has more than $(K-1)\times(H-l)$ data points for any sub-path sequence ρ is $P < c' \times N \times p[X > (K-1)\times(H-l)] = O\left(\Psi^{\log^2 N}\right)$. Therefore, $P(h > H) = O\left(\Psi^{\log^2 N}\right)$. □

Property .2 The expected height of the PK-tree for a set D of N data points with parameter Ψ is $O(\log N)$.

Proof. Let h be the height of the PK-tree. By Property .1, $P(h > H) = O\left(\Psi^{\log^2 N}\right)$ where $H = \Omega(\log N)$ and $H > 4\log N$. We choose $H = c_h \log N$, where $c_h > 4$ is a constant. Because the height of a PK-tree with N leaves is at most N, the expected height is

$$
\begin{aligned}
E(h) &\leq (c_h \log N)\times(1 - P(h > c_h\log N)) + N \times P(h > c_h \log N) \\
&\leq c_h \log N \times \left(1 - O\left(\Psi^{\log^2 N}\right)\right) + N \times O\left(\Psi^{\log^2 N}\right) \\
&\leq c_h \log N + N \times O\left(\Psi^{\log^2 N}\right) \\
&= O(\log N + N \times \Psi^{\log^2 N}) \\
&= O(\log N)
\end{aligned}
$$

since $\Psi^{\log^2 N} = \left(\Psi^{\log N}\right)^{\log N} = O\left(\frac{\log N}{N}\right)$ when $\Psi < 1-\epsilon$. □

B Insertion Algorithm

Algorithm B.1 *Data Point Insertion*

DATAINSERT(T,d)
/* *insert data point* d *in the PK-tree rooted at* T */
{
  instantiate the corresponding point cell C_d $(d \in C_d)$
  INSERT(T,T,C_d)

}
INSERT(T, C_W, C_d)
/* *insert point cell* C_d *into the subtree rooted at* C_W */
{
    if $\exists$ *a child* C *of* C_W *such that* $C_d \subset C$
        then *INSERT*(T, C, C_d)
        else
            make C_d *be a child of* C_W
            CHECKINSTANTIABLE(T, C_W)
}
CHECKINSTANTIABLE(T, C_W)
/* *check instantiability of* C_W *and all its ancestors* */
{
    $ChildrenSet \leftarrow$ *set of all children of* C_W
    if $|\ ChildrenSet\ | < K$ *and* $C_W \neq T$ /* *K is the rank of the PK-tree* */
        then
            $P \leftarrow$ *parent of* C_W
            make all nodes in $ChildrenSet$ *be children of* P
            deinstantiate C_W /* *free the space occupied my* C_W */
            CHECKINSTANTIABLE(T, P)
        else if $(\exists C' \subset C_W$ *and* $C'_1, \ldots, C'_K \in ChildrenSet$
                    such that $C'_1 \subset C', \ldots, C'_K \subset C')$ *and*
                    $(\nexists C'' \subset C'$ *and* $C''_1, \ldots, C''_K \in ChildrenSet$
                    such that $C''_1 \subset C'', \ldots, C''_K \subset C'')$
            then
                instantiate C' /* *allocate space for* C' *and assign a node ID* */
                make C' *child of* C_W
                for *each node* C_i *in* $ChildrenSet$
                    if $C_i \subset C'$
                        then *make* C_i *be a child of* C'
                CHECKINSTANTIABLE(T, C_W)
}

C Query Algorithms

Range Queries

Given a location d and $range \geq 0$, $RANGE(T, d, range)$ will return the set of data points within *range* distance from d. The idea is to begin from the root of the PK-tree and recursively traverse down all the nodes that have nonzero intersection with the sphere centered at d with radius *range* until we reach the leaf level. If an intermediate node C' is entirely enclosed by the sphere, then we will call another function RETRIEVE-CELL(C', *Result*) to add every data point in C' to the result set without further checking. The formal description of the algorithm is in Appendix C.

When *range* is small, the number of data points returned can be regarded as constant. Then, the computational complexity is $O(h)$ where h is the height of the PK-tree. However, if *range* is large and the number of data points returned is linearly proportional to N, then the computational complexity becomes $O(N)$ where N is the cardinality of the data set upon which the PK-tree is built. Single point retrieval is a special case of range queries where $range = 0$.

Algorithm C.1 *Range Queries*

RANGE($T, d, range$)
{
  $Result \leftarrow \emptyset$
  RANGE-SUBTREE($T, d, range, Result$)
  return $Result$
}

RANGE-SUBTREE($C, d, range, Result$)
{
  for *each child* C' *of* C
    if *ENCLOSE*($C', d, range$)
    then *RETRIEVE-CELL*($C', Result$)
    else if *OVERLAP*($C', d, range$)
      then *RANGE-SUBTREE*($C', d, range, Result$)
}

RETRIEVE-CELL($C, Result$)
{
  if C *is a point cell*
  then $Result \leftarrow Result \cup \{C.data\}$
  else for *each child* C' *of* C
    RETRIEVE-CELL($C', Result$)
}

K Nearst Neighbor Queries

Given a location $d = (d_1, d_2, \ldots, d_m)$ and a constant k, K-N-N(T, d, k) returns the k nearest neighbors in the PK-tree. The variable *bound* is used to store the range within which there are at least k data points. We utilize two sets $Candidates$ and $Result$. $Candidates$ contains the set of (non-leaf) cells that need be examined further. They are actually those non-leaf cells that overlap with the sphere centered at d with radius *bound*. The cells in $Result$ are those cells enclosed by this sphere. At the beginning, *bound* is set to ∞. All children of the root T are added to $Result$ and all nonleaf children are added to $Candidates$. After this, $UPDATE(k, Result, Candidates, bound)$ is invoked to set *bound*. Each time, a cell C in $Candidates$ is chosen to be examined, it is removed from $Candidates$ and $Result$. There are many ways to choose a cell from the $Candidates$ set. Here we always choose the closest cell to d according to Euclidean distance. The $Candidates$ set is maintained by using a heap to facilitate this process. All children of C that are contained within the sphere centered at d with radius *bound* are added to $Result$ and all non-leaf children overlapping with the sphere are added to $Candidates$. Then $UPDATE(k, Result, Candidates, bound)$ shrinks *bound* to be the smallest distance such that there are at least k data points in the cells that are contained within the sphere centered at d with radius *bound*, and removes all "unqualified" elements according to the new *bound* from *Candidates* and *Result*. This procedure continues until *Candidates* is empty. At this point, the elements in *Result* are the k nearest neighbors of d and hence are returned. The complete algorithm is presented in Appendix C. If we regard k as a constant, then the complexity is $O(h)$ on average where h is the height of the PK-tree.

Algorithm C.2 *K Nearest Neighbor Queries*

K-N-N(T, d, k)
{
  $Candidates \leftarrow \emptyset$
  $Result \leftarrow \emptyset$
  $bound \leftarrow \infty$
  for *each child* C *of* T
    GET-DISTANCE$(C, dis_{min}, dis_{max}, d)$
    if C *is not a point cell*
    then
      $Candidates \leftarrow Candidates \cup \{(C, dis_{min}, dis_{max})\}$
    $Result \leftarrow Result \cup \{(C, dis_{min}, dis_{max})\}$
  UPDATE$(k, Result, Candidates, bound)$
  while $Candidates \neq \emptyset$
    $C \leftarrow$ *GET-NEAREST*$(Candidates)$
    for *each child* C' *of* C
      GET-DISTANCE$(C', dis'_{min}, dis'_{max}, d)$
      if $dis'_{min} \leq bound$ *and* C' *is not a point cell*
      then
        $Candidates \leftarrow Candidates \cup \{(C', dis'_{min}, dis'_{max})\}$
      if $dis'_{max} \leq bound$
      then $Result \leftarrow Result \cup \{(C', dis'_{min}, dis'_{max})\}$
    UPDATE$(k, Result, Candidates, bound)$
  return *Result*
}

GET-DISTANCE$(C, dis_{min}, dis_{max}, d)$
{
  if C *is a point cell*
  then
    $dis_{min} \leftarrow \sqrt{(C.d_1 - d_1)^2 + (C.d_2 - d_2)^2 + \ldots + (C.d_m - d_m)^2}$
        /* m *is the number of dimensions* */
    $dis_{max} \leftarrow dis_{min}$
  else
    $dis_{min} \leftarrow 0$
    $dis_{max} \leftarrow 0$
    for $j \leftarrow 1$ **to** m
    $dis_{min} \leftarrow dis_{x_{min}} + \left(\min\{\mid C.d_j - d_j \mid - L_j^{C.level}/2, 0\}\right)^2$
    $dis_{max} \leftarrow dis_{x_{max}} + \left(\mid C.d_j - xd_j \mid + L_j^{C.level}/2\right)^2$

$dis_{min} \leftarrow \sqrt{dis_{min}}$
$dis_{max} \leftarrow \sqrt{dis_{max}}$
}

```
UPDATE(k, Result, Candidates, bound)
{
   datanum ← 0
   sort elements (C', dis'_min, dis'_max) of Result in ascending order of dis'_max into a list
      (C^1, dis^1_min, dis^1_max), ..., (C^j, dis^j_min, dis^j_max)
   i ← 0
   while datanum < k and i ≤ j
      i ← i + 1
      datanum ← datanum + C^i.datanum
   bound ← dis^i_max
   while i < j
      i ← i + 1
      if dis^i_max > bound
      then Result ← Result − {(C^i, dis^i_min, dis^i_max)}
      for each element (C, dis_min, dis_max) in Candidates
         if dis^i_min > bound
         then Candidates ← Candidates − {(C^i, dis^i_min, dis^i_max)}
}
```

D PK-tree Generation Algorithm

Algorithm D.1 *Build PK-tree*

```
BUILDPKTREE(D, N, K, R)
{
   if D is not already in PK-order
   then
      D ← PK_SORT(D, N, R) /* sort D into PK-order */
   a stack S is initiated to be ∅
   for j ← 1 to N
      d ← D[j]
      instantiate the corresponding leaf node C_d
      BUILD-SUB-TREE(S, C_d, K, R)
   pop(S, C)
   if C is at level 0
   then T ← C
   else
      instantiate root T
      make C a child of T
      while S ≠ ∅
         pop(S, C)
```

```
            make C a child of T
    return T
}

BUILD-SUB-TREE(S, C_d, K, R)
{
    i ← 1
    while i ≤ K and S ≠ ∅
        pop(S, Cells[i])
        i ← i + 1
    if i = K + 1
    then
        C ← FIND-MIN-CELL(Cells, K, R)
                /* get the smallest common supercell of all cells in Cells */
        if C_d ⊂ C
        then
            for k ← K downto 1
                push(S, Cells[k])
        else
            instantiate C
            for k ← 1 to K
                make Cells[k] a child of C
            pop(S, C')
            while C' ⊂ C
                make C' a child of C
                pop(S, C')
            push(S, C')
            BUILD-SUB-TREE(S, C_d, K, R)  }
```