PL-Tree: An Efficient Indexing Method for High-Dimensional Data

Jie Wang, Jian Lu, Zheng Fang, Tingjian Ge, and Cindy Chen

Department of Computer Science,
University of Massachusetts Lowell,
Olsen Hall, 198 Riverside St., Lowell, MA 01854, U.S.
http://cs.uml.edu

Abstract. The quest for processing data in high-dimensional space has resulted in a number of innovative indexing mechanisms. Choosing an appropriate indexing method for a given set of data requires careful consideration of data properties, data construction methods, and query types. We present a new indexing method to support efficient point queries, range queries, and k-nearest neighbor queries. Our method indexes objects dynamically using algebraic techniques, and it can substantially reduce the negative impacts of the "curse of dimensionality". In particular, our method partitions the data space recursively into hypercubes of certain capacity and labels each hypercube using the Cantor pairing function, so that all objects in the same hypercube have the same label. The bijective property and the computational efficiency of the Cantor pairing function make it possible to efficiently map between high-dimensional vectors and scalar labels. The partitioning and labeling process splits a subspace if the data items contained in it exceed its capacity. From the data structure point of view, our method constructs a tree where each parent node contains a number of labels and child pointers, and we call it a **PL-tree**. We compare our method with popular indexing algorithms including R*-tree, X-tree, quad-tree, and iDistance. Our numerical results show that the dynamic PL-tree indexing significantly outperforms the existing indexing mechanisms.

1 Introduction

Large-scale applications on high-dimensional data need efficient querying mechanisms to quickly retrieve information. These data objects may contain tens and even hundreds of dimensions. Reducing dimensionality is a standard approach, which has met with certain success in applications where the most significant information resides on a small number of dimensions. However, dimensionality cannot always be reduced without losing critical information; even if it can be reduced, the remaining data objects may still contain tens of dimensions. A number of indexing mechanisms have been developed for indexing multidimensional data. Among them, the R*-trees [2] and X-trees [4] which are evolved from R-trees [11], have become the dominating indexing methods because they are geometrically suited for spatial data. R-trees suffer from exponential blowups of MBRs as the dimensionality increases. Attempts at reducing overlapping bounded boxes have resulted in R*-trees; attempts at avoiding splits have resulted in

M.A. Nascimento et al. (Eds.): SSTD 2013, LNCS 8098, pp. 183–200, 2013.

X-trees. However, there are often significant overlaps among the sibling nodes in these indices, especially when the dimensionality is high. The Quadtree [22] is another commonly used indexing mechanism that is based on the hierarchical decomposition of data space. The Quadtree structure is overlap free and partitions the d-dimensional data space recursively into 2^d hypercubes.

We note that breaking the spell of the "curse of dimensionality" for indexing high-dimensional data is not impossible with a new line of thinking and our attempt is an algebraic approach. In particular, we present a new indexing method to support efficient point queries, range queries, and k-nearest neighbor (KNN) queries. Our method, called **PL-tree** (Partition and Label tree), is designed to substantially reduce the negative impacts of the "curse of dimensionality", with the following additional properties:

1. *Overlap free.* It partitions the data space into hypercubes and maps each hypercube to a unique label.
2. *Strong scalability.* It scales up well in terms of both dimensionality and data size, measured by the number of pages accessed and the total elapsed time for queries.
3. *Distribution insensitivity.* It works well regardless of the distribution of the data being indexed.

The main idea of the PL-tree is to partition the space recursively and map multiple dimensional vectors into scalar labels. It works as follows: (1) partition the original space into hypercubes, also called subspaces; (2) map objects in a subspace to a unique fixed point; (3) label each object in a subspace using the Cantor pairing function [6] on the fixed point of the subspace so that all objects in the same subspace have the same label and objects in different subspaces have different labels; (4) if the number of data objects contained in a subspace is greater than a pre-determined bound (e.g. the size of a database page), continue this process recursively. The structure of a PL-tree is a tree of labels with a number of children at each node, where each label uniquely identifies the set of objects contained in the same subspace of the node. PL-tree indexing cuts down searching redundancy substantially for range queries, and is especially suited for indexing large volumes of high-dimensional data. The uniqueness property of labeling plays a critical role in processing range queries efficiently (which cannot be achieved by hashing). Moreover, label generation using the Cantor pairing function is time efficient, and incurs little space overhead.

In this paper we present algorithms to construct a PL-tree and carry out point queries, range queries, and KNN queries. For dynamic data sets we also present algorithms for inserting and deleting data. We carry out detailed performance evaluations through a large number of experiments and show that PL-trees on high-dimensional synthetic and real-world data outperform the popular indexing methods.

The rest of the paper is organized as follows. We provide a brief overview of the background and related work in Section 2. In Section 3, we describe the PL-tree and present algorithms for constructing PL-trees, carrying out queries, and inserting (deleting) data to (from) an existing PL-tree. We show experiment results and performance analysis in Section 4. We conclude the paper with final remarks in Section 5.

2 Related Work

In the past several decades, many indexing methods for multidimensional data have been proposed.

R-Tree-Based Methods. The R-tree family is the most popular indexing structure including many multidimensional indexing methods [1, 2, 4, 11, 15, 20, 25]. An R-tree is a dynamic, balanced indexing structure which models data partition using Minimal Bounding Rectangles (MBRs). A node in an R-tree is split into two nodes if it contains too many MBRs. The splitting method with different heuristic optimizations varies among different R-tree variants. R-trees [11] split an MBR by minimizing the areas of the resulting MBRs, while R*-trees [2] also consider the overlaps. Hilbert R-trees [15] group similar MBRs using a "good" ordering based on the Hilbert curve. PR-trees [1] use priority rectangles on bulk loaded data where rectangles are represented as 4-dimensional points. However, these R-tree variants are mainly for indexing low-dimensional data. Several R-tree-based structures such as TV-trees and X-trees are designed to handle high-dimensional data. TV-trees [20] reduce dimensionality by ordering dimensions on their importance so that only important information among data objects is stored. X-trees [4] introduce the concept of supernodes to minimize overlaps in high dimensional space which keeps the directory as hierarchical as possible and at the same time avoids splits in the directory.

Space-Partition Methods. In addition to R-tree-based methods using MBRs to model the space partition, many methods employ the regular-partitioning of multidimensional data space. The Quadtree [24] is such a method which recursively divides the d-dimensional data space into 2^d sub-spaces. The Grid File [21] partitions space into buckets for indexing k-dimensional data, using a directory which contains a k-dimensional array and k one-dimensional arrays. The VA-File [27] (Vector Approximation File) is an array of b-bit strings which divides the data space into 2^b rectangular cells and uses a b-bit string for each cell. The Pyramid technique [3] is proposed to support efficient range queries which is based on a special partitioning strategy and is optimized for high-dimensional data. However, Fonseca and Jorge [9] point out that if the database is not uniformly distributed, the efficiency of range queries using Pyramid Technique cannot be guaranteed.

Feature-Based Methods. The feature-based similarity search is also an important search paradigm in database applications. SS-trees [28] are proposed for this purpose, which use Minimum Bounding Spheres (MBSs) rather than MBRs as bounding regions. SR-trees [16] are proposed to retain the advantages of what MBSs and MBRs can offer, but require more storage space for storing information of both MBRs and MBSs. A-trees [23] apply Relative Approximation to the hierarchical structure of SR-trees by introducing the concept of VBRs (Virtual Bounding Rectangles) which contain approximated MBRs and data objects.

Metric-Based Methods. There are also metric-based indexing structures [29] [7]. The metric-based methods differ from other indexing mechanisms in that they are based only on the relative distance between the data points. The VP-tree [29] is a static indexing method using binary tree based on the omni search strategy, where data points

are indexed according to their distance to a set of vantage points. The M-tree [7] is the most efficient metric-based indexing structure known so far. It follows the idea of the Bisector Tree (BST) and the Geometric Near-Neighbor Access Tree (GNAT) to group data points around a set of representatives.

Dimension-Reducing Methods. In iMinMax(θ) [22], a data point in d-dimension is mapped to a $1D$ line using its maximum or minimum value of all dimensions. A query to the d-dimensional data is then mapped into d subqueries, with one query for each dimension. The NB-trees [9] calculate the Euclidean norm of a n-dimensional data point and inserts it into a B^+-tree. The iDistance [14] index is also based on B^+-Tree. The iDistance [14] index improves the efficiency of kNN queries by reducing dimensionality. It uses a clustering algorithm to choose reference points, and calculates the distance between each point and its closest reference point. This distance and an additional scaling value is called iDistance. However, iDistance only works well with point data.

We have discussed many indexing methods above, some methods are only suited to the point data and some can handle both point and spatial objects. Surveys of common indexing methods can be found in [5, 10]. The index can be constructed in different ways: If data are mostly static such as GIS databases, an efficient index of good structure can be constructed using bulkloading algorithms. For instance, Kim and Patel [17] showed that for kNN queries, STR bulkloaded [19] R^*-trees outperform Quadtree, but the dynamic constructed R^*-trees perform worse than Quadtree. However, some database applications are highly dynamic, such as stock or moving object databases. These databases need to be constructed using a dynamic algorithm. Moreover, many previous works evaluate different indices on particular queries. Kothuri et al. [18] compare R-trees and Quadtrees using a variety of range queries on 2D GIS spatial data and show that R-trees outperform Quadtrees in general. Hoel and Samet [13] evaluate the performance of traditional spatial overlap join on various R-tree variants and the Quadtree, and show that R+trees and Quadtrees outperform R^*-trees using 2D GIS spatial data. Corral et al. [8] compare R^*-trees, X-trees, and VA-File, and show that R^*-trees outperform X-trees and VA-File on closet pair queries.

3 PL-Tree Indexing

In this section, we first introduce our dimensionality reduction method that maps multiple dimensional data into a scalar value. Secondly, we present PL-tree indexing algorithms on point data. Finally in Sect. 3.6, we explain how to construct PL-tree on spatial data and how to answer queries.

3.1 Multidimensional Space Mapping

Without loss of generality, we assume that data in a k-dimensional space are in the non-negative quadrant of a coordinate system; we refer to this coordinate system as the *home system*. Let $\mathbb{R}_0$ and $\mathbb{N}$ denote the sets of non-negative real numbers and integers, $D = (d_1, \ldots, d_k)$ be a point in the home system where $d_i \in \mathbb{R}_0$ for $i = 1, \ldots, k$. and $U = (u_1, \ldots, u_k)$ be a rescaling vector, where $u_i \in \mathbb{R}_0$ for $i = 1, \ldots, k$. A *scaling function* $S_U : \mathbb{R}_0^k \rightarrow \mathbb{N}^k$ is defined to map a real vector D to an integral vector:

$$S_U(D) = (\lfloor d_1/u_i \rfloor, \ldots, \lfloor d_k/u_i \rfloor), i = 1, \ldots, k.$$

A paring function is a bijection from integral k-dimensional points to integers and the Cantor paring function is a standard and commonly used paring function. We define the Cantor paring function $f_k : \mathbb{N}^k \to \mathbb{N}$ as follows:

$$f_k(i_1, \ldots, i_k) = \begin{cases} f_2(i_1, f_{k-1}(i_2, \ldots, i_k)), & \text{if } k > 2, \\ \frac{1}{2}(i_1 + i_2)(i_1 + i_2 + 1) + i_1, & \text{if } k = 2. \end{cases}$$

By using the *scaling function* and the *pairing function* we reduce a k-dimensional data point D to an integer by $L = f_k(S_U(D))$. Based on such reduction, we present the algorithms for PL-tree indexing in the following sections, and we also discuss the choice of the *re-scaling vector* U.

3.2 PL-Tree Index Structure

A leaf node of a PL-tree index contains a bounding box B and the multidimensional data identifiers included in B. Directory (non-leaf) nodes are in the form of (U, B, E), where U is a re-scaling vector; B is the bounding box of the hypercube represented by the node; and E is a list of entries of form (L, ptr), sorted by the value of L (L is an integer label and ptr points to node). In PL-tree nodes, the following processes may be operated in the home system of the corresponding hyperspace.

Partitioning. When the number of objects contained in a hypercube exceeds the limit (i.e., a page's capacity), we partition it into smaller sub-hypercubes according to the *re-scaling vector* U, so that the side length of a sub-hypercube is u_i units on the *i-th* dimension. Then we re-scale the sub-hypercubes into unit hypercubes consisting of 1 unit on each side in a new coordinate system. We refer to this new coordinate system as a U-system.

Mapping. Let $C_l = (l_1, \ldots, l_k)$ be the lowest corner point of a hypercube C in the U-system. In the home system, for any point $(p_1, \ldots, p_k)$ in C, we define *non-upper-boundary* points as points satisfying $u_i \cdot l_i \leq p_i < u_i \cdot (l_i + 1)$. Let $D = (d_1, \ldots, d_k)$ be a *non-upper-boundary* point, by the *scaling function* we obtain that $C_l = S_U(D)$. It follows that all *non-upper-boundary* points in C can be mapped to the same fixed point C_l. In Fig. 1, the 3 points in the middle cube are mapped to the point(3,3) which is $C_l = (1, 1)$ in the U-system.

Labeling. We define $C_L = \{D \mid S_U(D) = C_l\}$ as the set of *non-upper-boundary* points in C. According to the pairing function, we can label all points in C_L with a label $L = f_k(C_l)$, then the point set C_L becomes the child node with the label of L. Since f_k is a bijection, it is straightforward to show that L is unique with respect to C_L.

Re-coordination. When we move from a hypercube into its sub-hypercubes, we consider the sub-hypercube as a new home-system, by a re-coordinating process. For any D in a sub-home with a *re-scaling vector* U, we re-coordinate it to: $D_r = (d_1 - d'_1 \cdot u_1, \ldots, d_k - d'_k \cdot u_k)$, where $d'_i = \lfloor d_i/u_i \rfloor = l_i$, for $i = 1, \ldots, k$. For example, in Fig. 1 the 3 points in the middle are re-coordinated to (1,2), (2.5,1.5), and (1.2,0.5) in the home system of the middle sub-square.

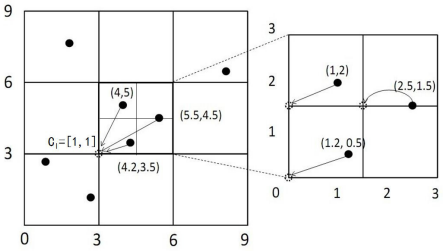


Fig. 1. PL-tree partition. The figure illustrates a PL-tree of 7 points in a 2D space, where each node can contain at most 2 points. In the home system of the root node $U = (3.0, 3.0)$ is used, and the home system of the middle square uses $U = (1.5, 1.5)$.

3.3 Point Data Indexing

Static Data Indexing

Given a dataset, we take the procedure of partitioning, mapping, labeling, and re-coordinating on the data space. The data with the same label become the children of the parent node. The procedure is repeated recursively on each hypercube until all the points of a sub-hypercube can fit in a page. Fig. 1 demonstrates the PL-tree partitioning and one of its sub-hypercube as a child node in a 2D space.

Dynamic Data Indexing

Searching The search algorithm traverses the tree from the root using Alg.1. In the worst case, the number of node accesses for a single search is $O(h)$ where h is the expected height of the tree (see Sect. 3.5). Note that the labels at a node are compact and sorted. Thus, a particular label can be located very fast through a binary search.

Insertion. The insertion algorithm (see Alg. 2) is similar to searching. It inserts a data entry D into a leaf node if D is found in a leaf, or it creates a new leaf including D if the label of D is not found in a directory node. There is also an overflow treatment when a leaf is full.

Deletion. To remove a data record from a PL-tree, a similar searching algorithm is invoked; once the record is found in a leaf node, it is removed from the leaf node. We remove the leaf node and its label from its parent node if the left node becomes empty. A merge process may be executed after a deletion.

Query Processing

A point query on PL-tree is straightforward according to the search algorithm (see Alg. 1). For range queries which ask for a particular range (or hyper cuboid) of data points, let Q denote the query range. Hypercubes in the home system that intersect with Q can be classified into *innerblocks* and *outerblocks*. An $innerblock$ is a hypercube that is fully contained in Q, while an $outerblock$ partially intersects with Q. We devise a procedure RQP (Algorithm. 3) to carry out range queries. Given a query range Q, RQP identifies the innerblocks and outerblocks by checking the coordinates of each

Algorithm 1. Search(PLTNode N, Data D)

```
1: if N is a leaf node then
2:    Compare all entries N.E with D
3: else
4:    L ← N.Label(D)
5:    pos ← N. Search_in_children(L)
6:    if L is found then
7:       re-coordinate D
8:       return Search( N.E[pos].ptr, D )
9:    else
10:      return False
11:   end if
12: end if
```

label in the U-system. For each innerblock, RQP returns all the data points in it; For each outerblock, RQP casts the intersected portion as a range query on the corresponding sub-hypercube and processes the query recursively.

Given a query point p and a value k, a kNN query returns k data points which are closet to the query point p based on a distance function. We use a similar algorithm described in [12] to carry out kNN queries in PL-trees. The algorithm in [12] is implemented using R-trees; we adapted it to use PL-trees. To find the k-nearest neighbors for a query point p, PL-trees maintain a priority queue consisting of objects sorted based on their MINDIST from p. When the next object is retrieved from the priority queue, if it is the bounding box of a node, it is expanded by pushing its children into the priority queue; if it is a data entry, the data entry is reported as the next nearest neighbor to p. This process is repeated until k data entries are reported. According to the analysis in [12], the expected number of leaf node accesses is $O(k+\sqrt{k})$ and the expected number of objects in the priority queue is $O(\sqrt{k})$.

3.4 Compact PL-Tree Storage

In our insertion algorithm, while splitting a leaf node, if the data in the leaf are evenly distributed, the algorithm will create many new leaves with each leaf node containing only a few data points. These new "sparse" leaves seriously affect the efficiency of range queries, because too many "sparse" pages are accessed during the query. The problem becomes worse with increasing dimensionality. Hence we propose a compact storage which significantly reduces such "sparse" effect.

The basic idea is to store many leaf nodes in the same page. As a result, we need to store labels of the hypercubes corresponding to these nodes to identify each node. We refer to the labels that identify nodes as *home labels*, which indicates the location of the node with an offset value. We refer to the original labels with child pointers as *child labels*. Fig.2 illustrates the compact node structure, there are two arrays of entries, child label entries and home label entries. In a practical implementation, the two arrays of directory nodes grow respectively from the two ends of the page. Fig.3 is an example of a PL-tree and the corresponding compact pages, the value of offset_i indicate the location

Algorithm 2. Insert(PLTNode N, Data D)

```
1: if N is a leaf node then
2:    Insert D.id into N
3:    if N.full() then
4:       N.U = U-Calculator()
5:       for all data d in N do
6:          re-coordinate d and Insert(N, d)
7:       end for
8:    end if
9: else
10:   L ← N.Label(D)
11:   pos ← N. Search_in_children(L)
12:   if L is found then
13:      re-coordinate D
14:      Insert(N.E[pos].ptr,D)
15:   else
16:      create NewChild with L in N
17:      Insert(NewChild, D)
18:   end if
19: end if
```

Algorithm 3. RQP(PLTNode N, Range Q)

```
1: Res ← empty set
2: if N is a leaf node then
3:    check all data in N and add the intersected ones into Res
4: else
5:    for all E in N.E do
6:       C_l ← N. DeLabel(E.L)
7:       let B be unit hypercube based on C_l
8:       if B is in Q in U-system then
9:          Res = Res∪ FetchAll(E.ptr)
10:      else
11:         if B intersects Q in U-system then
12:            Res = Res∪ RQP(E.ptr, Q)
13:         end if
14:      end if
15:   end for
16: end if
17: RETURN Res
```

of the first entry of the node with the home label L_i. In leaf nodes, we store the *home label* for each data entry.

The compact structure slightly complicates the PL-tree algorithms. For example, if we search for $D = O6$ in the PL-tree of Fig.3, we need one more parameter of home label for each search. Starting from the root node using Search(root, D, 0), the Search algorithm calculates the label 3 of D in root. Then Search(directory-node, D, 3) is called, and we calculate label 10 and find it from [5,10,97]. In the leaf node, we compare D with the objects with home label of 10 ($O6$-$O9$) and finally find $O6$ as the result. Nonetheless, the compact structure significantly improves the performance of queries (see Section 4.6).

3.5 Re-scaling Vector and Algorithm Performance

An interesting parameter of PL-trees is the *re-scaling vector* U. Our goal is to choose U such that the PL-tree is as height balanced as possible.

Height Estimation of PL-trees. Considering the points in a k-dimensional space, we assume that the root space is a hypercube with edges of the same length L and the smallest distance between any two points is l. If each time a PL-tree partitions hypercubes by splitting each dimension into d parts, then for a node of depth h, the diagonal of the node should be larger than l since a node must contain at least one point. Thus, we have $\sqrt{k(\frac{L}{d^{h-1}})^2} \geq l$ and it follows that $h \leq \log_d \frac{\sqrt{k}L}{l} + 1$, which is $O(\log L/l)$ if L/l is much larger than k. Hence in the worst case, the height of the PL-tree is logarithmic to the ratio of L to l. That is, the height of PL-trees is related to the density of the data.

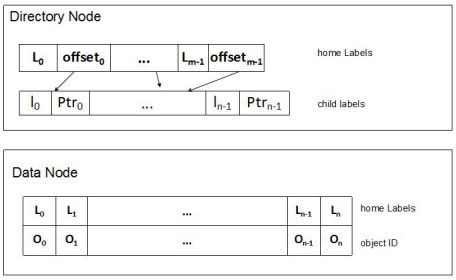


Fig. 2. Structure of PL-Tree Nodes

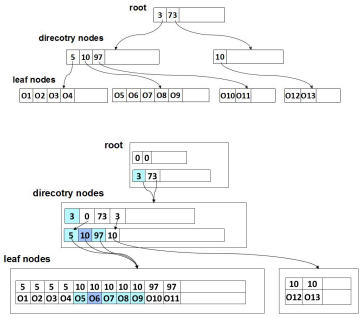


Fig. 3. Compact PL-tree with 7 nodes

Binwidth Optimization. If we partition the space randomly in PL-Trees, the hypercubes with dense data will be of deep height. Also, there are many blank areas in the hyperspace for non-uniformly distributed data. To make PL-trees as height balanced as possible, we choose U to partition the data of the hypercube as evenly as possible. Our partition strategy is, by choosing a proper U, the hypercube is partitioned into sub-hypercubes such that the sub-hypercubes have about the same number of data entries. Considering the data projections onto each dimension, choosing u_i for the i-th dimension is a *binwidth optimization* problem. Let $X = x_1, \ldots, x_n$ be a data set, the binwidth optimization is to find an optimal Δ such that the range of X is divided into N bins of width Δ. Let k_i be the number of elements in X that fall in the i-th bin. We first calculate the value of N that minimizes the MISE (Mean Integrated Squared Error) [26], which is a measure of the goodness-of-the-fit of the bin histogram to an unknown data rate. Then we choose the optimal N^* from 2 to N to minimize the modified variance-to-mean ratio V/M and the optimal u_i is $\frac{\max(X)-\min(X)}{N^*}$, where $M = \frac{1}{N'}\sum_{i=0}^{N} k_i$, $V = \frac{1}{N'}\sum_{i\text{-th bin is not empty}} (k_i - M)^2$, and N' is the number of non-empty bins.

3.6 Spatial Data Indexing

Spatial data can be considered as a k-dimensional spatial object represented by an MBR denoted by R. R can be uniquely determined by its lowest corner point R_l and highest corner point R_h, written as: $R_l = (a_1, \ldots, a_k)$, and $R_h = (b_1, \ldots, b_k)$.

Static Spatial Data Indexing

Alg. 4 shows the pseudocode for creating a PL-tree index from static spatial data. The process is recursive, similar to point data, but has the following changes:

Partitioning. We need to consider the size of the object while determining U to ensure each object covers at most 2^k hypercubes. Let S_I denote a static set of spatial objects. We calculate U as follows: calculate U' according to Section 3.5 and $U'' = \{u_1, \ldots, u_k\}$, where $u_i = \max_{(R_l,R_h)\in S_I}(b_i - a_i)$, then let $U = \max\{U', U''\}$.

Algorithm 4. CreateIndex_Cuboid(PLTNode N, DataSet A)

```
1: N.U ← U-Calculator(A)
2: for all data in A do
3:    if data is huge then
4:       N.HList.add(data)
5:    else
6:       L ← N.Label(data)
7:       pos ← N.Search_in_children(L)
8:       if L is not found then
9:          temp ← new PLTNode
10:         insert (L, temp) to N.E
11:         temp.add_data(data)
12:      else
13:         temp ← N.E[pos].ptr
14:         temp.add_data(data)
15:      end if
16:   end if
17: end for
18: for all child of N do
19:   if child.size > split_threshold then
20:      CreateIndex_Cuboid(child, child.dataset)
21:   end if
22: end for
23: return N
```

It ensures that every R must be confined in a region consisting of 2^k adjacent hypercubes that share a common point in the center of the region.

Mapping & Labeling. For any $R = (R_l, R_h) \in S_I$, we map R to a label L using $R' = (S_U((a_1, \ldots, a_k)), S_U((b_1, \ldots, b_k))) = ((a'_1, \ldots, a'_k), (b'_1, \ldots, b'_k)) = (R'_l, R'_h)$ and $L = f_2(R'_l, R'_h)$. The calculation still follows the bijective property of f_k such that the mapping from bounding boxes to labels is bijective.

Re-coordination. Instead of re-coordinating a single point, we re-coordinate both R_l and R_h for a bounding box R.

Huge Objects. Even if we choose a big U for indexing spatial data, it is still possible that there are relatively huge objects (e.g. one object is of half size of the hypercube). Thus, we use an extra link list (HugeObjectList) to store such huge objects.

Dynamic Spatial Data Indexing

When the insertion algorithm dynamically inserts a new spatial data entry R into the existing index structure, R may be too large for the size of U. We could re-calculate U and re-coordinate all existing data, but this is time consuming. Instead, we logically cut R, along the lines of the existing hypercubes, into several smaller hyper cuboids. We then logically replace R with the derived smaller hyper cuboids for indexing. A logical cut implies that the cut does not physically generate smaller objects to replace the original object. Each derived smaller hyper cuboid is simply a pointer to the original

object. The coordinates of each derived hyper cuboid are calculated on the fly, which may be disposed after indexing. Thus, what we finally get is a set of pointers at different positions in the index pointing to the same original data. The original data will not be indexed. Alg. 5 and 6 show the pseudocodes of insertion and deletion for handling dynamic spatial data. We omit the search algorithm as it is similar to point data. Compared to the algorithms for indexing point data, inserting a spatial data may cost much more due to the extra insertions from the logical cut and the maintainance of HugeObjectList.

Query Processing

The query processing for spatial data is similar to that for point data; the only difference is that the objects in HugeObjectsList of each nodes are also considered. Due to space constraints, we do not provide the detailed algorithms here. Moreover, due to the logical cut, there may be some duplicate object pointers in the result for range queries and kNN queries. Therefore, we need to remove the duplicates before returning the result.

4 Performance Evaluation

In this section, we present the experimental results on point, range and kNN queries for multidimensional data. All indices are constructed using dynamic indexing methods. We compare PL-trees with R*-trees and X-trees. The R*-tree is the most common method for indexing multidimensional data, while the X-tree is an improved variant of the R*-tree for high-dimensional data. We also compare PL-trees with Quadtrees and some other indexing methods.

4.1 Data Sets and Configurations

We use both synthetic and real world data. For synthetic data, we generate a uniformly distributed data set containing 1,000,000 points in 15-dimensional space. Moreover, we use the following real world data for different evaluations:

1. USPP. A Point data set containing 15,206 populated places in the U.S.
2. TIGER (http://www.census.gov/geo/www/tiger/). A spatial dataset containing 556, 696 non-uniformly distributed polygons.
3. LLMPP. A High-dimensional data set from Lymphoma/Leukemia Molecular Profiling Project (http://llmpp.nih.gov/lymphoma/), which contains 1,843,200 items of 17 integer and 15 float attributes. We only took the 15 float attributes and randomly select 500,000 data points.
4. MAPS. The MAPS Catalog data containing photometric and astrometric data from the Palomar Observatory Sky Survey (http://aps.umn.edu/catalog/). It has 90 million items of 39 integer attributes, and we select one field P105 (40,398 points).

We implement the algorithms in C++ on an Intel Core i5 2.53G machine running Windows 7 with a 4GB memory. We apply LRU for buffer pool which is in the size of 1000 pages. The page size is 4KB for data sets of dimensionality no more than 8 and 8KB for higher dimensionalities. We focus on the following evaluation metrics: number of pages accessed, index size, and total query time.

Algorithm 5 . Insert_Cuboid(PLTNode N, DATA $data$)

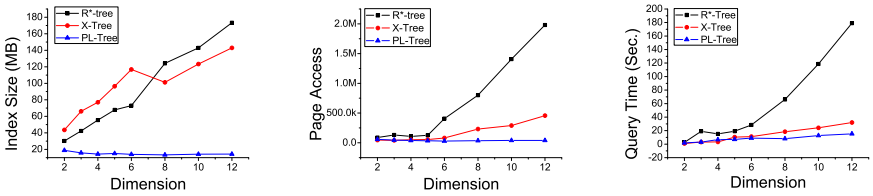
```
1: if data has larger size than N.U then
2:     LogicalCuts ← partition(data, N.U)
3:     for all lc in LogicalCuts do
4:         Insert_Cuboid(N, lc)
5:     end for
6: else
7:     if N is a leaf node then
8:         N.add_data(data, L)
9:         if IsHuge(data, N.U) then
10:            N.HList.add(data)
11:        else
12:            insert data into N.E
13:            if N is full then
14:                IndexCreate_Cuboid(N,N.E)
15:            end if
16:        end if
17:    else
18:        L ← N. Label(data)
19:        pos ← N. Search_in_children(L)
20:        if L is not found then
21:            temp ← new PLTNode
22:            insert (L, temp) into N
23:            Insert_Cuboid(temp, data)
24:        else
25:            Insert_Cuboid(N.E[pos].ptr, data)
26:        end if
27:    end if
28: end if
```

Algorithm 6 . Delete_Cuboid(PLTNode N, DATA $data$)

```
1: if data has larger size than N.U then
2:     LogicalCuts ← partition(data, N.U)
3:     for all lc in LogicalCuts do
4:         Delete_Cuboid(N, lc)
5:     end for
6: else
7:     if N is a leaf node then
8:         if data IsHuge(N.U) then
9:             remove data from N.HList
10:        else
11:            remove data from N.E
12:        end if
13:        if N.E and N.HList are empty then
14:            remove N from N.parent
15:        end if
16:    else
17:        L ← N. Label(data, N.U)
18:        pos ← N.Search_in_children(L)
19:        if L is not found then
20:            return NOT_FOUND
21:        else
22:            Delete_Cuboid(N.E[pos].ptr, data)
23:        end if
24:    end if
25: end if
```

4.2 Index Size

One of the great features of a PL-Tree is that the size of index is dimensionality independent since PL-Trees store scalar labels instead of the multi-dimensional information. R-Tree-based methods store MBRs whose size increases linearly with dimensionality. By contrast, a label is a constant size. We create indices on synthetic points with different dimensionalities ($D = 2, 3, 4, 5, 6, 8, 10, 12$) and compare the sizes of index files. Fig. 4(a) shows that the index size of R-Tree-based indices increases about linearly with dimensionality. As expected, the size of a PL-Tree index is dimensionality independent. A PL-tree for the 2-dimensional dataset is 18872 KB, which is 0.62 times as large as an R*-Tree; while for the 12-dimensional dataset it is 14274 KB, which is only 0.08 times as large as an R*-Tree.



(a) Index size (b) Number of pages accessed (c) Total query time

Fig. 4. Performance Comparisons on Synthetic Datasets

4.3 Results for Point Queries

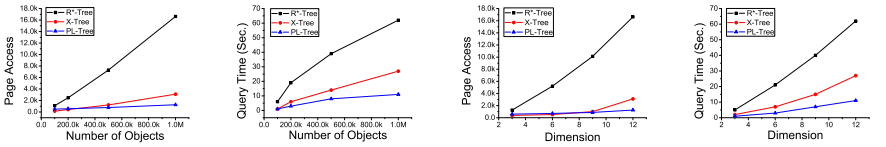
We evaluate point queries using synthetic data sets of various dimensionalities. We fix the data size at 500,000 records and create indices with different dimensionalities. We then randomly select 10,000 points to carry out point queries. For point queries on a hierarchical tree index, the query cost directly corresponds to the height of the tree. However, this is only true if there is no overlap between hypercubes of directory nodes. Due to the overlap of R-tree-based indices, Fig.4(b) shows that R*-Trees incur 50 times as many page accesses as PL-Trees for $D = 12$. X-Trees are designed as the hybrid of linear array-like and hierarchical R-Tree-like directory, and provide a much better performance by avoiding the splits to reduce overlaps. PL-Trees guarantee that there is only one path from root to leaf for a single point query, which implies that the number of page access only corresponds to the depth of the leaf nodes.

4.4 Results for Range Queries

Synthetic Datasets

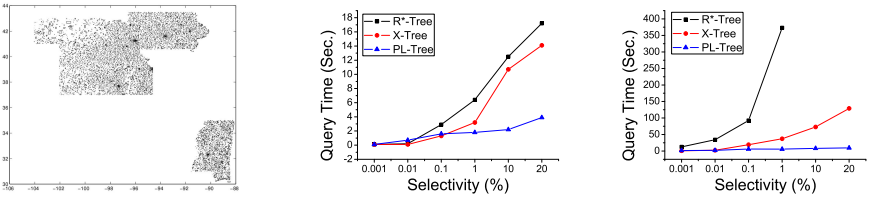
In the first experiment, we compare the performance of range queries on synthetic data. We fix the selectivity to 0.1%, and carry out 10 random range queries. By varying the number of data points and the dimensionality respectively, we evaluate the impact of data size and data dimensionality on the range query performance. We measure both the number of page accesses and the total elapsed time.

We first fix the dimensionality to 12 and vary the data size from 100,000 to 1,000,000 records. Fig. 5(a) shows that PL-trees always perform much better than R*-trees and slightly better than X-trees on larger datasets. Then, we fix the data size to 1,000,000 and vary the dimensionality ($D = 3, 6, 9,$ and 12). Fig. 5(b) shows that X-trees perform slightly better when D is 6 or lower, but PL-tree performs better when D is higher. The reason is that, for relatively low dimensionalities, X-trees have significantly less overlap than R*-trees, and less intersection check than PL-trees. However, for high dimensionalities ($D = 9$ and 12) we observed that the performance of X-Trees decreases rapidly while PL-trees maintain a rather smooth curve despite the dimensionality growth. The results show that PL-trees scale well on both size and dimensionality.



(a) Performance over Data Size (b) Performance over Data Dimensionality

Fig. 5. Range Query Performance on Synthetic Datasets

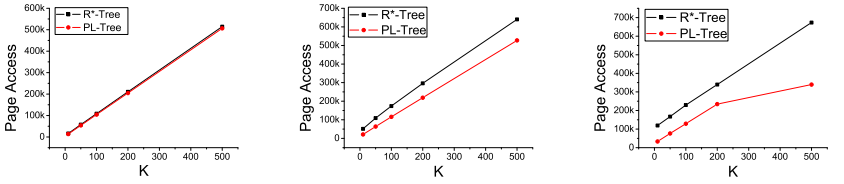


(a) TIGER2D Overview (b) Query Times on TIGER (c) Query Times on LLMPP

Fig. 6. Range Query Performance on Real-World Datasets

Real-World Datasets

We also evaluate indices on TIGER and LLMPP data sets to further examine the performance of PL-trees for various dimensionalities. We vary the selectivities from 0.001% to 20%. Fig.6 shows that as expected, PL-trees still outperform R*-Trees and X-Trees, especially for high selectivity. This is because a larger query range tends to incur more overlaps for R*-trees and X-trees, which lead to more I/O costs. The speedup factor of PL-trees over R*-trees and X-Trees reaches up to 7.5 and 3.6, respectively, when the selectivity is 20%.



(a) Results on MAPS2D (b) Results on MAPS4D (c) Results on MAPS6D

Fig. 7. kNN Query Performance on Real-World Datasets

4.5 Results for kNN Queries

In this section, we compare the performance of PL-trees with R*-trees on kNN queries. We use the first 2, 4, and 8 attributes of MAPS data to get three real datasets: MAPS2D,

Table 1. Comparisons with Quadtrees

	Quadtree	PL-tree	Ratio
Creating Time (sec.)	0.368	4.700	11.2
Creating Disk I/O	40027	738871	54.15%
Avg. Read I/O of PQ	5.829	4.194	76.33%
Avg. Read I/O of kNN			99.07%
Avg. Time of RQ (sec.)	0.4234	0.2211	0.522
Avg. IntrCheck of RQ	77324	17265.8	22.32%
Avg. Read I/Oof RQ	71076.8	41375.8	58.21%

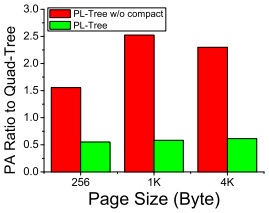
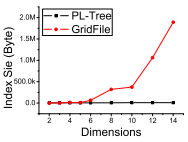
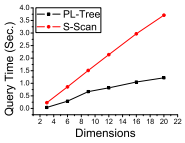


Fig. 8. Compact PL-trees

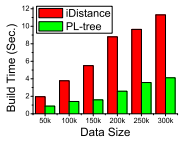


(a) v.s. GridFile

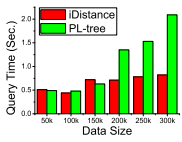


(b) v.s. S-Scan

Fig. 9. Comparison to GridFile and S-Scan



(a) Build Time



(b) Query Time

Fig. 10. Comparison to iDistance

MAPS4D, and MAPS6D. We randomly generate 1000 points for kNN queries and count the number of pages accessed. Fig .7, shows that the numbers of page accesses are proportional to the values of k, and PL-trees outperform R*-trees in that their numbers of accessed pages are only 83.6%-98.4%, 41.2%-81.3%, and 28.0%-50.5% of those of the R*-trees for 2D, 4D, and 6D datasets, respectively. We also observe that the performance of PL-trees over R*-trees increases with dimensionality.

4.6 Comparison with Quadtree

Quadtrees bear some similarity with PL-trees. However, PL-trees enjoy flexibilities in space partitioning and have a novel usage of Cantor pairing functions for labels. In this section, we evaluate these two indices on USPP data. We create indices on USPP using different page sizes (from 256B to 4KB). For point queries, we select 1% points of the data sets and search them in indices. To carry out kNN queries, we randomly generate 1000 points for processing kNN queries with different k. We use different selective factors from 0.1% to 30% for evaluating range queries. Table 1 shows that PL-trees need less disk I/O but much more CPU times due to the calculation of U, and perform better than Quadtrees on point queries but similarly on kNN queries. For range queries, PL-trees outperform Quadtrees in all metrics. Especially, PL-trees require significantly less intersection checking since inner-blocks are not checked (see Section 3.3). Moreover, to illustrate the benefit introduced by the compact structure (see Section 3.4), we also compare the performance on range queries for PL-trees with and without the compact structure. Fig.8 shows the ratio of page accessed number of PL-trees to that of the Quadtree, it shows that PL-trees outperform Quadtrees with the compact structure, but not without the compact structure. This verifies that the compact structure is very effective in improving performance.

4.7 Comparison with Other Methods

We also compare PL-trees with the GridFile, S-Scan, and iDistance. We first compare PL-trees with the GridFile, we create the index for 10,000 randomly generated data points with the dimensionality ranging from 2 to 14 and compare the index sizes. Fig. 9(a) shows that the space overhead of the GridFile increases dramatically especially for high dimensional data. We also evaluate the range query performance of PL-trees and pure sequential scans (S-Scan) on 100,000 synthetic data points in different dimensions with selectivity of 0.1%. Fig. 9(b) shows that PL-trees are far superior to S-Scan. Finally, we also compare PL-trees with iDistance. which is of high efficiency for point queries and kNN queries. We create the indices on 6-dimensional point data in different sizes from 50,000 to 300,000 points. We randomly select 1000 points to evaluate the performance of point queries on PL-trees and iDistance. Fig. 10 shows that the build time of an iDistance index is much longer but iDistance outperforms PL-tree for point queries. However, iDistance has a serious drawback that it only works for point data.

5 Conclusions

In this paper, we propose a new indexing method for high dimensional data. PL-trees label objects in high-dimensional space by using the Cantor pairing function which maps a high-dimensional vector into a scalar label bijectively. Due to the "curse of dimensionality", many existing indexing methods do not perform well for high dimensional data. Our indexing employs a novel usage of the Cantor paring functions to cope with high dimensionalities. Crucially, one can restore the corresponding multidimensional data from a scalar value since the Cantor pairing function is invertible. Our results show that our new indexing method scales up nicely with dimensionality and data size, and outperforms the state-of-the-art indexing methods in many ways.

Acknowledgments. Jie Wang was supported in part by the NSF, under the grants CCF-0830314, CNS-1018422, and CNS-1247875. Tingjian Ge was supported in part by the NSF, under the grants IIS-1149417 and IIS-1239176.

References

1. Arge, L., de Berg, M., Haverkort, H.J., Yi, K.: The priority R-tree: A practically efficient and worst-case optimal R-tree. In: Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD), pp. 347–358 (2004)
2. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD), pp. 322–331 (1990)
3. Berchtold, S., Böhm, C., Kriegel, H.-P.: The pyramid-technique: Towards breaking the curse of dimensionality. In: Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD), pp. 142–153 (1998)
4. Berchtold, S., Keim, D.A., Kriegel, H.-P.: The X-tree: An index structure for high-dimensional data. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 28–39 (1996)

5. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33(3), 322–373 (2001)
6. Cantor, G.: Contributions to the Founding of the Theory of Transfinite Numbers. Dover, New York (1955); Original year was 1915
7. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 426–435 (1997)
8. Corral, A., Cañadas, J., Vassilakopoulos, M.: Processing distance-based queries in multidimensional data spaces using r-trees. In: Manolopoulos, Y., Evripidou, S., Kakas, A.C. (eds.) PCI 2001. LNCS, vol. 2563, pp. 1–18. Springer, Heidelberg (2003)
9. Fonseca, M.J., Jorge, J.A.: Indexing high-dimensional data for content-based retrieval in large databases. In: Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA), pp. 267–274 (2003)
10. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. 30(2), 170–231 (1998)
11. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD), pp. 47–57 (1984)
12. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. 24(2), 265–318 (1999)
13. Hoel, E.G., Samet, H., Tree, R.: Benchmarking spatial join operations with spatial output. In: Proceedings of the 21st International Conference on Very Large Data Bases, pp. 606–618 (1998)
14. Jagadish, H.V., Ooi, B.C., Tan, K.-L., Yu, C., Zhang, R.: idistance: An adaptive b+-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. 30, 364–397 (2005)
15. Kamel, I., Faloutsos, C.: Hilbert R-tree: An improved R-tree using fractals. In: VLDB, pp. 500–509 (1994)
16. Katayama, N., Satoh, S.: The SR-tree: An index structure for high-dimensional nearest neighbor queries. In: Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD), pp. 369–380 (1997)
17. Kim, Y.J., Patel, J.: Performance comparison of the r*-tree and the quadtree for knn and distance join queries. IEEE Transactions on Knowledge and Data Engineering 22(7), 1014–1027 July
18. Kothuri, R.K.V., Ravada, S., Abugov, D.: Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD 2002, pp. 546–557. ACM, New York (2002)
19. Leutenegger, S., Lopez, M., Edgington, J.: Str: a simple and efficient algorithm for r-tree packing. In: Proceedings of the13th International Conference on Data Engineering, pp. 497–506 (April 1997)
20. Lin, K.-I., Jagadish, H.V., Faloutsos, C.: The TV-tree: An index structure for high-dimensional data. VLDB Journal 3(4), 517–542 (1994)
21. Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst. 9(1), 38–71 (1984)
22. Ooi, B.C., Tan, K.-L., Yu, C., Bressan, S.: Indexing the edges - a simple and yet efficient approach to high-dimensional indexing. In: Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2000, pp. 166–174. ACM, New York (2000)
23. Sakurai, Y., Yoshikawa, M., Uemura, S., Kojima, H.: The A-tree: An index structure for high-dimensional spaces using relative approximation. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 516–526 (2000)

24. Samet, H., Webber, R.E.: Storing a collection of polygons using quadtrees. ACM Trans. Graph. 4(3), 182–222 (1985)
25. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R+-tree: A dynamic index for multi-dimensional objects. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 507–518 (1987)
26. Shimazaki, H., Shinomoto, S.: Kernel bandwidth optimization in spike rate estimation. Journal of Computational Neuroscience 29(1-2), 171–182 (2010)
27. Weber, R., Schek, H.-J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 194–205 (1998)
28. White, D.A., Jain, R.: Similarity indexing with the SS-tree. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 516–523 (1996)
29. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 311–321 (1993)