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

ABSTRACT

The quest for processing spatial data in high-dimensional space has resulted in a number of innovative indexing mechanisms. Most of the early methods index data according to their geometric relationship. While they have success to a certain degree, these methods, unfortunately, either lack efficiency, particularly at a higher dimensionality, or are too complex to implement. These drawbacks have made the geometry-based indexing methods less desirable in large-scale applications involving high-dimensional data. To conquer these problems, we introduce a new algebra-based method for indexing high-dimensional data. Our method first partitions the original data space into hypercubes, and then labels each object in a hypercube using a bijective pairing function. All objects in the same hypercube have the same label and vice versa. The bijective pairing function provides the hypothesis to map a high-dimensional vector into a scalar value. This partition and label process continues recursively, until each hypercube contains a pre-determined number of data points. From a structural point of view, our method forms a tree of labels with a number of children for each parent node which we call a PL-tree. Hypercubes in the PL-tree are indexed by labels instead of high-dimensional vectors. In this paper, we present algorithms to construct a PL-tree index and algorithms to carry out the range queries. We have also done experiments that compare the performance of PL-tree with some popular indexing methods.

Keywords

access methods and indexing, query processing, high-dimensional database

1. INTRODUCTION

Large-scale science and engineering applications demand efficient querying mechanisms to retrieve information from high-dimensional data. In these applications the databases always consist of millions of data objects in several tens to a few hundreds of dimensions. To index such high-dimensional datasets, reducing the dimensionality is a natural and important way, based on the observation that a small number of dimensions may carry the most information of high-dimensional datasets. However, in many cases, even after reduction, we may still have tens of dimensions.

Many indexing structures have been developed for indexing multidimensional data. One can find the survey of many popular indexing methods in [7]. The R-tree [11] indexing is first introduced for indexing low-dimensional data and its variants such as R*-trees [3] and X-trees [6] have since become the dominating indexing methods of multidimensional data. Most of the early indexing mechanisms index data according to their geometric relationships. Unfortunately, these geometry-based methods either lack efficiency at high dimensionality or are too complicated to implement, due to the “curse of dimensionality”, i.e., the overlap of the bounding boxes in the directory rapidly increases as the dimensionality goes up.

However, breaking the spell is not impossible with a new line of thinking. We have designed an algebraic indexing method that avoids introducing overlaps between directory rectangles. Our method, called PL-tree (Partition and Label tree) scales up well in terms of both the dimensionality and the data size. For practical needs, PL-tree indexing also achieves the following properties:

1. Low complexity. It incurs slow growth on the total number of page accesses and the total CPU time for queries as dimensionality and/or the number of records increase.

2. Distribution insensitivity. It works well regardless of the distribution of the data being indexed.

3. Easy implementation. It is easy to understand and straightforward to implement.

The main idea of the PL-tree is as follows: (1) Partition the original space into hypercubes, where each hypercube is called a sub-home. (2) Map objects in a sub-home to a fixed point that is unique to the sub-home. (3) Label each object in a sub-home using a pairing function on the fixed point so that all objects in the same sub-home have the same label and objects in different sub-homes have different labels. (4) If the number of objects contained in a sub-home is greater than a pre-determined bound (e.g. as determined by the database page size), continue this process recursively on each sub-home. 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 sub-home and the children points to the node of that sub-home. The PL-tree indexing method cuts down search redundancy in range queries, and is especially suitable for indexing a large amount of high-dimensional data. The uniqueness property of labeling plays an important role in making range queries efficient since hashing techniques are not suitable to generate labels for this purpose; and using pairing function to generate labels incurs little space overhead.
In this paper, we present algorithms to construct the PL-tree and carry out point/range queries. For dynamic datasets we also devise data insertion and deletion algorithms. We demonstrate that PL-trees have the properties of low complexity, distribution insensitivity, and easy implementation. Also, we carry out a comprehensive experimental analysis and show that PL-trees, using real and synthetic data of arbitrary distributions, are superior to the widely used indexing methods such as R*-trees [3] and X-trees [6] on page accesses, index size, total elapsed time, and other performance measures.

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

2. RELATED WORK

Many indexing methods have been developed in the past several decades. These methods utilize different strategies to carry out multidimensional data.

2.1 R-tree Based Methods

The R-tree family is the most popular indexing structure in multidimensional space, which includes R-trees [11], R+-trees [21], R*-trees [3], Hilbert R-trees [14], PR-trees [2], TV-trees [16], and X-trees [6]. An R-tree is a dynamic, height-balanced indexing structure similar to a B-tree which models database partition using Minimal Bounding Rectangles (MBRs). Each node in the R-tree contains as many sub-MBRs as possible within the page size. A node is split into two nodes if the number of sub-MBRs contained in the node becomes too large. The splitting algorithms vary among different R-tree variants. R-tree splits an MBR by minimizing the areas of the resulting MBRs, while R*-tree also considers the overlaps besides the area. Hilbert R-tree [14] groups similar MBRs using a “good” ordering based on the Hilbert curve to reduce the area and the margin of the resulting MBRs. PR-trees [2] use priority rectangles on bulk-load data where rectangles are represented as 4-dimensional points, and are inserted to a kd-tree [4]. However, these R-tree variants are used mainly for indexing low-dimensional data but not suitable for high-dimensional data. There are several R-tree-based structures designed to handle high-dimensional data. TV-trees [16] reduces dimensionality by ordering dimensions on their importance and organizing the directory so that only important information among data objects is stored. X-trees [6] introduces the concept of supernodes to minimize overlaps in high dimensional space. The basic idea is to keep the directory in a hierarchy as much as possible, which helps to prevent splits and avoid higher overlaps. However, how to determine the maximum overlap is difficult, and X-trees are more suited for medium-dimensional data space. Hybrid-trees [8] uses a multidimensional data structure to index high-dimensional data by taking the advantages of the data partition in the R-tree family and space partition in K-D-B trees [4].

2.2 Feature Based Methods

Recently, the feature-based similarity search has become an important search paradigm in database applications. SS-trees [23] indexing was proposed for this purpose, which uses Minimum Bounding Spheres (MBSs) rather than MBRs as bounding regions. SR-trees [15] were proposed to overcome the drawbacks introduced by any MBR or MBS-based indexing mechanism, and at the same time retain the advantages of what MBSs and MBRs can offer. The region of an SR-tree is specified by the intersection of an MBR and an MBS, but SR-tree requires more storage space for storing information of both MBRs and MBSs in internal nodes. A-trees [20] apply Relative Approximation to the hierarchical structure of SR-trees by introducing the concept of VBRs (Virtual Bounding Rectangles) which contains and approximates the MBRs and data objects.

2.3 Metric Based Methods

In addition to the tree-based indexing structures, there are also metric-based indexing structures, which include VP-trees [24] and M-trees [9]. The metric-based indexing structures differ from the previous indexing structures in that they are based only on the relative distance between the data points. The VP-tree 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 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. An internal node stores representatives along with their covering radius. At each level, the data points are chosen and associated to their closest representative.

2.4 Space Partition Methods

The Pyramid technique [5] was proposed to support efficient range queries which is based on a special partitioning strategy that is optimized for high-dimensional data. It divides the data space into 2-dimensional pyramids, where the center of the data space is the top. A data point is approximated by the pyramid it belongs to and by the distance to the top. However, Fonseca and Jorge [10] pointed out that if the database is not uniformly distributed, the efficiency of range queries cannot be guaranteed. The Grid File [18] partition space into buckets for indexing multi-dimensional data. The buckets are organized by the grid directory which contains a k-dimensional array and k one-dimensional arrays. VA-File [22] (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. Similar to the Pyramid technique, the performances of Grid File and VA-file are also sensitive to the distribution of the data.

2.5 Dimension Reducing Methods

In iMinMax(θ) [12, 25], a d-dimensional data point is mapped to a 1-d 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-tree [10] also maps high-dimensional data points into a single dimension. It calculates the Euclidean norm of a n-dimensional data point and inserts this value into a B+ tree. There is also a B+*-tree based indexing method called iDistance [13], which is an indexing and KNN query processing technique on point data in high dimensional metric space. The iDistance method improves the efficiency of KNN queries by using reducing dimensionality. The iDistance method uses a clustering algorithm to choose reference points, and calculates the distance between each point and its closest reference point. This distance and a scaling value is called iDistance. Multi-dimensional data points are mapped to one-dimensional scalar values. However, iDistance only works well with KNN queries since it utilizes distance as the (dis)similarity between data points.

3. PL-TREE INDEXING

PL-trees take the advantages of both space partition and dimension reducing methods, and deal with two types of data in a high-dimensional hyperspace: point data and spatial data (Spatial data
are represented by bounding boxes). Without loss of generality, we assume that data are located in a bounded hyperspace referred to as the data space. Let \( k \) be the dimensionality of the data space. For simplicity, we assume that point data or vertices of bounding boxes in the data space are in the non-negative quadrant of a coordinate system, we refer to this coordinate system as a **home system**. Let \( \mathbb{R}_0 \) and \( \mathbb{N} \) denote the set of non-negative real numbers and integers. Let \( D = (d_1, \ldots, d_k) \) be a point in the home system, where \( d_i \in \mathbb{R}_0, i = 1, \ldots, k \). Let \( U \) be a positive number to be used as a re-scaling unit. Define a scaling function \( S_U : \mathbb{R}_0^k \rightarrow \mathbb{N}^k \) to map \( D \in \mathbb{R}_0^k \) to an integral point in \( \mathbb{N}^k \) as follows:

\[
S_U(D) = ([d_1/U], \ldots, [d_k/U]).
\]

A pairing function is a bijection (i.e., a one-to-one, total, and onto mapping) from integral \( k \)-dimensional points to integers. The function \( f_x : \mathbb{N}^k \rightarrow \mathbb{N} \) defined below is a standard pairing function commonly used in recursion theory [19]. For all \((i_1, \ldots, i_k) \in \mathbb{N}^k\):

\[
f_x(i_1, \ldots, i_k) = \begin{cases} 
\frac{1}{2} (i_1 + i_2)(i_1 + i_2 + 1) + i_3, & \text{if } k = 2, \\
2 i_1, & \text{if } k > 2.
\end{cases}
\]

### 3.1 Point Data Indexing

#### 3.1.1 Creating an Indexing

To build a PL-tree indexing structure of point data, we start a recursive process of partitioning, mapping, labeling, and re-coordination.

**Partitioning**: We divide the home system into hypercubes of size \( U \), where \( U \) is called the **re-scaling unit** which is chosen based on how many children nodes we would want a node to have. Each hypercube consists of \( U \) units on each edge, then we re-scale the home system into a new coordinate system where each hypercube consists of 1 unit on each edge in the coordinate system. We refer to this new coordinate system as a **U-system**.

**Mapping**: Let \( C_1 = (l_1, \ldots, l_k) \) be the lowest corner point of a hypercube \( C \) in the \( U \)-system. That is, \((U \cdot l_1, \ldots, U \cdot l_k)\) is the lowest corner point of \( C \) in the home system, and for any point \((p_1, \ldots, p_k)\) of \( C \) in the home system, we have \( p_i \geq U \cdot l_i \), where \( i = 1, \ldots, k \). Let \( D = (d_1, \ldots, d_k) \in C \). If there is at least one \( i \) such that \( d_i = (l_i + 1)U \), then we say that \( D \) is an upper-boundary point.

**Lemma 3.1.** Let \( D = (d_1, \ldots, d_k) \in C \) and \( D' = S_U(D) \). If \( D \) is not an upper-boundary point, then \( D' = C_l \).

**Proof.** Since \( D \in C \), we have \( d_i \geq U \cdot l_i \) for all \( 1 \leq i \leq k \). Since \( D \) is not a u-point, we have \( d_i < (l_i + 1)U \) for all \( 1 \leq i \leq k \). Thus, \( d'_i = [d_i/U] = l_i \).

It follows from Lemma 3.1 that for any point \( D \in C \), if \( D \) is not an upper-boundary point, then \( D' = C_l \). If \( D \) is a mapping, then \( D' \) is mapped to the same fixed point \( C_l = S_U(D) = D' \) under function \( S_U \) (see Figure 1). We note that, it is possible to use different re-scaling units on different axes. For example, if data are distributed very unevenly on projections of different dimensions, we may want to use different re-scaling units on different axes. In other words, we may use a smaller unit on the coordinate axis with a larger amount of data points projected on that axis and a larger unit otherwise.

**Labeling**: For each hypercube \( C \), let \( C_l \) denote the set of points in \( C \) with \( C_l \) being the lowest corner point excluding the upper-bounding points of \( C \) in \( U \)-system, and \( D \) is the corresponding set of points in home-system. It follows from Lemma 3.1 that \( C_l = \{ D \mid S_U(D) = C_l \} \). We define a label as \( L = f_x(C_l) \) (note that \( L \geq 1 \)); then all points in \( C_l \) share this label. The hypercubes represented by these labels become the children of the root, where the root represents the home system. For convenience, we assign a label of 0 to the home system.

For each \( L \) we maintain a pointer from \( L \) to all the data points with label \( L \). Since \( f_x \) is a bijection, it is straightforward to show that a different \( C_l \) has a different label; hence the pointers from \( L \) will only link to data points exactly in \( C_l \).

**Re-coordination**: If the size of \( C_l \) is larger than the page size, we need to partition this hypercube into sub-hypercubes with a smaller \( U \). Before the partition, we start a re-coordinating process on the hypercube. That is, we consider the hypercube as a new home system, called **sub-home**. For any \( D \) in the sub-home, we re-coordinate it to:

\[
D_r = (d_1 - d'_1 \cdot U, \ldots, d_k - d'_k \cdot U).
\]

Repeat the same procedure of partitioning, mapping, labeling, and re-coordinating for sub-home. The new labels from this process become the children of the node that represents the sub-home with label \( L \). That is, each sub-home is treated independently with a new coordinate system. This process is repeated recursively on each sub-homes until points of a sub-home can be placed in a page. Figure 1 demonstrates a home system and one of its sub-homes as a child node.

#### 3.1.2 Query Processing for Point Data

The query point on PL-tree is straightforward. Suppose we query the point data \( D = (d_1, \ldots, d_k) \) in PL-tree, we first query \( (0, D) \) in the root where 0 is the home label of home system, if the root is a data point we just check all points in the root sequentially. Otherwise, we calculate the label of \( D \) in \( U \)-system by using \( L_D = f_x(S_U(D)) \). Then, we search \( L_D \) in the label list with home label \( L = 0 \). If \( L_D \) does not exist, we can conclude that \( D \) is not in the database. If \( L_D \) is found, we re-coordinate \( D = (d_1 - [d_1/U]U, \ldots, d_k - [d_k/U]U) \), and query \( (L_D, D) \) in the sub-node with label \( L_D \). The query is repeated recursively until a data node is reached or the label of \( D \) is not found.

A range query asks for a particular range of data points, represented as a hyper cuboid. Let \( Q \) denote the query range, hypercubes in the home system that intersect with \( Q \) can be classified into **in-**
nerblocks and outerblocks, as shown in Figure 2. By innerblock it means a hypercube that is fully contained in \( Q \), and outerblock the one that is partially intersected with \( Q \).

![Figure 2: Innerblocks and Outerblocks](image)

We devise a procedure RQP to carry out range queries as follows. Given a range query \( Q \), RQP identifies the innerblocks and outerblocks by checking coordinates. For each innerblock with label \( L \), RQP returns all data points under the sub-node with label \( L \). This is because points that fall in the innerblock also fall in the query range. For each outerblock, the query processing casts the range query on the intersected portion as the subrange query problem on the corresponding hypercube in the sub-home, and processes the query recursively; that is, treat the subrange query as a query on this sub-home.

### 3.2 Structure of the PL-tree

There are two types of nodes in the PL-tree: directory node and data node. The data nodes only contain the pointers to the actual data objects, and the directory nodes contain labels together with pointers to sub-nodes where the labels are sorted in the increasing order. Figure 4(a) shows the structure of a simple PL-tree. However, we find that when the dimensionality increases, the volume of the space increases so fast that the data becomes sparse. Therefore, we find that when the dimensionality increases, the volume of different nodes are not unique (label value “10” in the example), so we only place the sub-nodes of different labels with the same level of the tree. Figure 4(b) is the corresponding physical storage structure. Note that the children with same home label are placed in the same split node. If \( N \) is a data node, the children list is divided according to the values in the \( L \) list. If there is only one home label contained in a data node, the data node will be converted into a directory node.

![Figure 3: Structure of PL-tree Nodes](image)

![Figure 4: PL-tree Structure](image)

flowTreatment procedure is called when the home-label list or the children list of \( N \) is full. It splits \( N \) as evenly as possible, if \( N \) is a directory node, the children list is divided according to the values in the \( L \) and Offset lists. Note that the children with same home label are placed in the same split node. If \( N \) is a data node, the children list is divided according to the values in the \( L \) list. If there is only one home label contained in a data node, the data node will be converted into a directory node.

### 3.3 Spatial Data Index

To index spatial data, what we need more is to find an appropriate re-scaling unit to partition the data space. Let \( O \) be a \( k \)-dimensional spatial object and \( R \) the Minimal Bounding Box of \( O \). We use \( R \) to represent \( O \). We note that \( R \) is uniquely determined by any of its \( 2^{k-1} \) diagonals in any direction. Let \( (x_1, \ldots, x_k) \) and \( (y_1, \ldots, y_k) \) be the two end points of any diagonal of \( R \). Let

\[
R_I = \{ \min\{x_1, y_1\}, \ldots, \min\{x_k, y_k\} \},
R_H = \{ \max\{x_1, y_1\}, \ldots, \max\{x_k, y_k\} \}.
\]

Then \( R_I \) is the lower corner point of \( R \) and \( R_H \) is the highest corner point of \( R \). We will use \((R_I, R_H)\) to represent \( R \). We write \( R_I \) as \((a_1, \ldots, a_k)\) and \( R_H \) as \((b_1, \ldots, b_k)\).

#### 3.3.1 Creating an Indexing
Algorithm 1 ChooseSubNode(PLTreeNode N, Data D, HomeLabel L)
1:   label ← pairing_func(D, N, U)
2:   sub_list ← the sub-list of N.children with home label L
3:   if sub_list = NULL then
4:       temp ← new PLTreeNode
5:       N.add_child(temp, L)
6:   else
7:       pos ← find_position(label, sub_list)
8:       temp ← sub_list[pos]
9:   end if
10:  return temp

Algorithm 2 OverflowTreatment(PLTreeNode N)
1:   if N is a data node then
2:       sort N.children according to N.L
3:   if N.L contains only one home label L then
4:       // Convert N into a directory node
5:       N ← CreateIndex_Cuboid(N, N.dataset, L)
6:   else
7:       divide N into two nodes N' and N'
8:       P ← parent node of N
9:       update P.child[i].ptr where P.child[i].label is in N'.L
10:  end if
11:  else
12:   divide N into two nodes N' and N'
13:  P ← parent node of N
14:  update P.child[i].ptr where P.child[i].label is in N'.L
15:  end if

Algorithm 3 shows the pseudocode for creating PL-tree index from existing data. The process is still recursive, similar to point data, including partitioning, mapping, labeling, and re-coordination.

**Partitioning:** Let $S_l$ denote a static set of spatial objects. We will discuss how to handle dynamic data later. Let
\[
U \geq \max_{(r_l, r_h) \in S_l} \{b_1 - a_1, \ldots, b_k - a_k\}. \tag{1}
\]
Partition the home system into hypercubes of size $U$, the following lemma is straightforward.

**Lemma 3.2.** Let $R$ be a bounding box. Then $R$ must be confined in a region consisting of $2^k$ adjacent hypercubes that share a common corner point in the center of the region such that on each 2-dimensional (2D) projection, these hypercubes look like the shape of $\mathbb{R}$ (see Figure 5(a)).

**Mapping:** For any $R = (R_l, R_h) \in S_l$, let
\[
R' = (S_l((a_1, \ldots, a_k)), S_l((b_1, \ldots, b_k))) = ((a'_1, \ldots, a'_k), (b'_1, \ldots, b'_k)).
\]
We have the following corollary of Lemma 3.2:

**Corollary 3.3.** For any $1 \leq i \leq k$, we have $b'_i = a'_i$ or $b'_i = a'_i + 1$.

We note that if $R$ falls in a hypercube and $R_h$ is not an upper-boundary point, then $b'_i = a'_i$ for all $1 \leq i \leq k$.

**Labeling:** Apply $f_x$ on points $(a'_1, \ldots, a'_i)$ and $(b'_1, \ldots, b'_i)$ to produce labels $l_e$ and $l_b$ for two points, then apply a 2D $f_x$ on $(l_e, l_b)$ to produce the label $L$ for $R$. The PL-tree maintains a point from $L$ to $R$, it follows the bijective property of $f_x$ that the mapping from bounding boxes to labels is bijective. A label $L$ may be shared by multiple bounding boxes. The objects in this bounding box would be in the sub-node of label $L$, let:
\[
A_L = \{ R \mid R \in S_l \text{ and } R \text{ has label } L \}.
\]

**Re-coordination:** If $|A_L| > B$ where $B$ is the page capacity, we will split the hypercube by re-coordinating. We re-scale the hypercube as a new coordinate system, referred to as a sub-home system, and re-coordinate objects in $A_L$ into the new sub-home system. That is, for any $R \in A_L$, we re-coordinate to $((a_1 - a'_1 \cdot U, \ldots, a_k - a'_k \cdot U), (b_1 - b'_1 \cdot U, \ldots, b_k - b'_k \cdot U))$ in the sub-home system. This prevents labels for bounding boxes in subproblems from becoming too large.

In the sub-home we will repeat the same procedure of partitioning, mapping, labeling, and re-coordinating with a new scale unit $U'$ of the subproblem. However, we note we should ensure that $U' < U$, otherwise it will defy the purpose of partitioning. Thus, we must exclude bounding boxes with an edge of length larger than $U'$. For bounding boxes of this kind, we simply maintain an extra link list to store them. In practice, we always choose much larger $U$ and $U' < U$, so that we can avoid such large bounding boxes. However, if there are relatively huge objects in the multidimensional data set, we still need a independent extra link list to store them. We use a U_Scan procedure to determine $U'$ and store the huge objects.

3.3.2 Neighbors of a hypercube

Let $C_x = ((x'_1, \ldots, x'_k), (x'_1 + 1, \ldots, x'_k + 1))$ be a hypercube after re-coordination. Then any bounding box that intersects with $C_x$ whose lowest corner point is in $C_x$ must have a diagonal that is re-coordinated to $((z'_1, \ldots, z'_k), (z'_1, \ldots, z'_k))$, where $z_i \in \{ x'_i, x'_i + 1 \}$. That is, there are $2^k$ labels for the neighbors of $C_x$ in a $k$-dimensional space.

Figure 5(b) demonstrates an example in a 2-dimensional space. In this figure, $C_x$ is the square labeled as 0, with three neighbor squares labeled as 1, 2, and 3. Any rectangle falling in this figure that intersects with Square 0 must have one of the following diagonals: $0 \rightarrow 0, 0 \rightarrow 1, 0 \rightarrow 2, 0 \rightarrow 3$; $x \rightarrow y$ denotes a diagonal that crosses Square $x$ and Square $y$.

Denoted by $Q$ the hyper cuboid of a range query with the lowest corner point $(a_1, \ldots, a_k)$ and the highest corner point $(b_1, \ldots, b_k)$ in the current coordinate system, we have $b_i \geq a_i$ for $1 \leq i \leq k$. 

![Figure 5: 2D example PL-tree hypercubes](image-url)
Let $n_Q$ denote the total number of unit hypercubes in $Q$. Then

$$n_Q = \sum_{i=1}^{k} (b'_i - a'_i + 1),$$

where $a'_i = |a_i/U|$ and $b'_i = |b_i/U|$. Therefore, the total number of labels of hypercubes intersecting with $Q$ is $2^k n_Q$. Similar to point data indexing, these $n_Q$ unit hypercubes are also divided into two types: innerblocks and outerblocks (See Figure 2). Correspondingly, the $2^k n_Q$ labels are calculated and divided into two types: innerlabels and outerlabels.

3.3.3 Range query processing

Processing a range query for spatial objects is almost the same as that for point data, except that the query processing procedure RQP also needs to look at all neighbors of each hypercube that intersects with the query range. Algorithm 4 demonstrates the process of the range query issued on hyper-cuboid data. In particular, we check each hypercube intersecting with the query range by calculating neighbor labels. If the label is an innerlabel, RQP simply returns all the records under it. Otherwise, RQP casts the query problem into a subproblem on the intersected portion of the outerblock in the sub-home system. The extra link list of huge objects is checked sequentially, in practice the list is always empty or very small if $U$ is large enough.

We note that in the case when the number of labels is smaller than the total number of hypercubes intersected with the query range and their neighbors, it is more efficient to search sequentially the hypercubes that contain spatial data without going through neighbors.

3.3.4 Dynamic spatial data

When the insertion algorithm dynamically inserts a new spatial data into the existing index structure, the re-scaling unit $U$ may be smaller than the size of the data. One could re-calculate $U$ and re-coordinate all existing data. But this is time consuming. Instead, we do a logically cut: along the lines of the existing hypercubes, the oversized data $R$ is divided into several smaller hyper cuboids. We then logically replace $R$ with the derived smaller hyper cuboids for indexing. By logical cut it means that the cut does not physically generate smaller objects to replace the original object. Each derived smaller hyper cuboid has a same pointer to the original object. Thus, what we get is a set of pointers at different positions in the indexing system pointing to the same original data. The original data will not be indexed. Algorithm 5 and 6 show the pseudocode of insertion and deletion for handling dynamic spatial data.

Due to the logical cut, there may be some duplicated object pointers in the result for a range query. Therefore, we need to remove the duplication before returning the result for a query.

4. PERFORMANCE EVALUATION

In this section, we present the experimental results of a PL-tree’s
One of the great features of a PL-tree is that the size of index is dimension-independent, since a PL-tree directory node stores one-dimensional label values instead of the multi-dimensional MBRs. R-tree-based methods need to store MBR information in directory nodes thus will decrease the capacity of each page along as dimensionality increases. We create the index trees on synthetic data sets consisting of 500,000 random points with different dimensionalities ($D = 2,3,4,5,6,8,10,12$) and store the tree structure on disk. Figure 6 shows that the index size of R-tree-based methods increases dramatically with the dimensionality growth. As expected, the size of a PL-tree index is dimensionality-independent. In Figure 6, the size of a 2-dimensional PL-tree index is 18872KB, which is 0.62 times as large as an R*-tree, while the size of a 12-dimensional PL-tree index is 14274KB, which is only 0.08 times as large as an R*-tree.

4.1 Comparison on Index Size

To measure the performance, we answer point queries and range queries on datasets with different sizes and dimensionalities. We randomly selected 10,000 points from the dataset that has length larger than $N.U$ then

for all $e$ in LogicalCuts do

let $P$ be the parent of $N$ then

$P$.remove_child($N$)

end if

else

label ← pairing_func(data, $U$)

loc ← Node.search_in_children($label$)

if loc = NULL then

$N$.remove_child($e$)

end if

end for

if $N$.children.empty() then

return NOT_FOUND

end if

else

label ← pairing_func(data, $U$)

loc ← $N$.search_in_children($label$)

if loc = NULL then

return NOT_FOUND

end if

end if

else

label ← pairing_func(data, $U$)

loc ← $N$.search_in_children($label$)

if loc = NULL then

return NOT_FOUND

end if

end if

4.2 Point Query Synthetic Data

We then evaluate point queries over PL-trees using synthetic datasets of various dimensionalities. We fix the dataset size at $N = 500,000$ and create the indexes on datasets with different dimensionalities. We randomly selected 10,000 points from the dataset to do point query in the tree. Figure 7 shows the performance of point queries. For point queries on a hierarchical tree directory, the number of required page accesses directly corresponds to the height of the tree. However, this is only true if there is no overlap between directory rectangles. Due to the high overlap of R-tree-like directory for high dimensional data, R*-tree indexing incurs 50 times as many page accesses as PL-tree indexing for $D=12$. That means for a single point query multiple paths have to be followed.

Algorithm 5 Insert_Cuboid(PLTNode $N$, DATA data, HomeLabel $L$)

1: if data has length larger than $N.U$ then

2: LogicalCuts ← partition(data, $N.U$)

3: for all lc in LogicalCuts do

4: Insert_Cuboid($N$, lc, $L$)

5: end for

6: else

7: if $N$ is a data node then

8: $N$.add_data(data, $L$)

9: else

10: label ← pairing_func(data, $U$)

11: loc ← Node.search_in_children($label$)

12: if loc = NULL then

13: temp ← ChooseSubNode($label$, $L$)

14: else

15: temp ← $N$.children[loc], ptr

16: Insert_Cuboid(temp, data, label)

17: end if

18: end if

19: if $N$.num_child = $n$ or $N$.num_homelabel = $m$ then

20: OverflowTreatment($N$)

21: end if

22: end if

Algorithm 6 Delete_Cuboid(PLTNode $N$, DATA data, HomeLabel $L$)

1: if data has length larger than $U$ then

2: LogicalCuts ← partition(data, $U$)

3: for all $e$ in LogicalCuts do

4: Delete_Cuboid($N$, $e$, $L$)

5: end for

6: else

7: if $N$ is a data node then

8: for all $e$ in $N$.children do

9: if $e$ = data then

10: $N$.remove_child($e$)

11: end if

12: end for

13: if $N$.children.empty() then

14: Let $P$ be the parent node of $N$

15: $P$.remove_child($N$)

16: end if

17: else

18: label ← pairing_func(data, $U$)

19: loc ← $N$.search_in_children($label$)

20: if loc = NULL then

21: return NOT_FOUND

22: end if

23: Delete_Cuboid($N$.children[loc], data)

24: end if

25: end if

26: end if

Figure 6: Index Size on Synthetic Point Data
X-tree is designed as a hybrid of linear array-like and hierarchical R-tree-like directory which provides a much better performance by avoiding the split to reduce the overlap. However, PL-tree indexing guarantees 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.

4.3 Range Query on Synthetic Data

For range queries, we generate random range queries according to the selectivity factor in the following way: suppose selectivity is 1%, then we first random pick 1% objects from the dataset and the bounding box of these objects is the query range we will used for the range query. The range queries used in this experiment are generated with a constant selectivity factor 0.1%. We first fix the dimensionality at $D=12$ and vary dataset size from 100,000 to 1,000,000 records. Figure 8(a) shows that the total number of page accesses of 10 random range queries. The PL-tree always performs much better than an R*-tree and slightly better than X-tree on larger data set, which means the PL-tree scales well to large database sizes.

To determine the impact of dimensionality on the performance of query processing, we fix the size of data set ($N=1,000,000$) and vary dimensionality from 3 to 12 using the same query times and selectivity factor. Figure 8(b) shows that X-tree indexing performs slightly better on dimensionalities 8 and below, because in low dimensionality, an X-tree has much less overlap than an R*-tree and a PL-tree also needs to search multiple paths for a single range query. However, in high dimensionalities ($D = 9$ and 12) we observe that the performance of an X-tree decreases rapidly while a PL-tree maintains a rather smooth curve regardless of the dimensionality increasing.

4.4 Experiments on Real Data

We also use real datasets in our experiments. We first use the dataset called TIGER Census Blocks from TIGER/Line® [17], which is a non-uniformly distributed 2D spatial data set containing the MBRs of 556,696 census blocks (polygons) of Iowa, Kansas, Missouri, and Nebraska. We vary the range query selectivity from 0.001% to 20% to evaluate both small queries and large queries on the data set. Figure 9(a) shows the comparisons of total elapsed time over range query selectivity. The PL-tree indexing performs better than the R*-tree and the X-tree indexing, especially when the size of a range query becomes larger, because this particular data set is highly clustered. The speedup factor of the PL-tree indexing over the R*-tree and the X-tree indexing reaches up to 7.5 and 3.6, respectively, when selectivity equals 20%.

We also evaluate the performance of the PL-tree indexing on high dimensional real data set in another experiment. This experiment is performed on a data set from Lymphoma/Leukemia molecular profiling project. The raw data contains 32 attributes of 17 integer and 15 float attributes. The total number of data items reaches to 1,843,200. We only take the 15 float attributes and randomly select 500,000 data items for this experiment. The total elapsed time for processing range queries is demonstrated in Figure 9(b), which shows that PL-trees again yields better performance over R*-trees and X-trees in high dimensional data.
4.5 Comparison with Other Methods

PL-tree partitions the space sub-hypercubes and pairs each sub-hypereube into a one-dimensional label value. We compare PL-tree indexing with iDistance [13] and GridFile [18] because iDistance has the similar dimension reduction to PL-tree and GridFile has the similar space partitioning.

The GridFile carries out range queries well but its updates are too expensive since it utilizes k-dimensional dynamic array to organize the spatial data. Therefore, for high dimensionality, updating such dynamic array requires unreasonable overhead in both space and time. We generate 10,000 data points with the dimensionality ranging from 2 to 14 and create the indexes. As shown in figure 10, the space overhead of the GridFile increases dramatically, especially for high dimensionalities.

The iDistance is an indexing method which utilizes the dimension reduction to improve the indexing efficiency. It partitions space into hyperspheres and employs one-dimensional distance value as point (dis)similarity. We fix the dimensionality at $D=8$ and vary the size of data set from 50,000 points to 300,000 points. In figure 11, we create the index and do point query 200 times, it shows that the building time of iDistance indexing is much longer, but iDistance performs better than PL-tree for point queries. However, iDistance has a serious drawback that it can only handle point queries.

5. CONCLUSIONS

In this paper, We propose a new indexing method for high-dimensional data. PL-tree uses a pairing function to label high-dimensional objects in high dimensional hypercubes. This pairing function can map a high-dimensional MBR vector into a label value bijectively. We introduce both point queries and range queries. Point queries are carried out directly, while range queries are first transformed into hyper cuboids and then carried out through lookup of the labels of hypercubes intersected with the hyper cuboids. Our experiment results show that the PL-tree scale up well with increased dimensionality and the size of the data set.

Due to the “curse of dimensionality”, geometry-based indexing may be not a good choice for indexing high-dimensional data. Our method provides a possibility to reduce the high-dimensionality to a scalar value by using a pairing function. This pairing function is invertible so that we can also restore the multi-dimensional vector from the label value.

6. REFERENCES


