Society for Science and Education
United Kingdom

TMLAI TRANSACTIONS ON MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

Volume 6 No. 6
ISSN 2054-7390

An Efficient Indexing Method for Load Networks

Guang-Ho Cha
Department of Computer Engineering, Seoul National University of Science and Technology, Republic of Korea
ghcha@seoultech.ac.kr

ABSTRACT

We propose a new indexing method, called the Hilbert-tree, to support the fast retrieval on road networks. Our goals are twofold: increasing the storage utilization and decreasing the directory coverage of the index tree. The first goal is achieved by absorbing splitting if possible, and when splitting is necessary, converting two nodes to three. This is done by proposing a good ordering on the directory nodes. The second goal is achieved by representing the directory regions compactly. We note that there is a trade-off between above two design goals, but the Hilbert-treee is so flexible that it can control the trade-off. We present the design of our index tree and associated algorithms. In addition, we report the results of a series of tests, comparing the proposed index tree with the buddy-tree, which is one of the most successful access methods for a multidimensional space. The results show the superiority of our method.

Keywords: Indexing method; Load networks; Multidimensional index.

1 Introduction

Geographic databases are becoming increasingly popular. In these applications, typical queries are the range queries and the nearest neighbor queries. It is assumed that objects lie in an *n*-dimensional feature space and each dimension corresponds to a specific feature. A domain of a feature is a set of values from which a value for the feature can be drawn. The feature space or domain space is defined as a Cartesian product of the domains of all organizing features. We call any subset of the domain space a region. We can map each object into a point in an *n*-dimensional feature space by using *n* feature-extraction functions. We distinguish between point access methods (PAMs) and spatial access methods (SAMs) which are designed to handle multidimensional point and spatial data, respectively.

The basic principle of multidimensional PAM is to partition the *n*-dimensional feature space into several regions, each containing not more than a fixed number of entries. Each region corresponds to one disk page and, upon becoming full, is split into two. Our index tree, the Hilbert-tree, improves the performance by allowing the regions to have more general shape and by representing them compactly, contrary to previously suggested PAMs.

2 The Hilbert-tree

The major aims of the Hilbert-tree are to increase the storage utilization and to decrease the directory coverage. The high storage utilization both in the directory and data nodes results in a small number of

DOI: 10.14738/tmlai.66.5861
Publication Date: 27th December, 2018
URL: http://dx.doi.org/10.14738/tmlai.66.5861

nodes and in turn a small number of disk accesses. The small directory coverage reduces the area covered by the directory regions.

Definition 1. The storage utilization *U* of a tree *T* is:

$$U = \frac{1}{n}\sum_{i=1}^{n}\frac{F_i}{P_i}$$

where F_i is the number of entries in node *i*, P_i is the maximum number of entries that a node *i* can have, and *n* is the number of nodes in the tree.

Definition 2. The node coverage C_n of node *i* and the directory coverage C_d of tree *T* are:

$C_n(i)$ = the area spanned by all the entries enclosed in the node *i*,

$$C_d(T) = \bigcup_{i=1}^{k} C_n(i)$$

where *k* is the number of nodes in the tree *T*.

To achieve above design goals, we use a space filling curve, and specifically, the Hilbert curve to apply a linear ordering on the data objects and the directory regions. A space-filling curve is a mapping that maps the unit interval onto the n-dimensional unit hypercube continuously. The path of space-filling curves provides a linear ordering on the grid points. The Peano curve (also known as the Z-curve) [1], the Hilbert curve [2], and the Gray-code curve [3] are examples of space filling curves. In [4] and [5], it was shown that the Hilbert curve achieves the better clustering than the others. The basic Hilbert curve on a 2×2 grid,

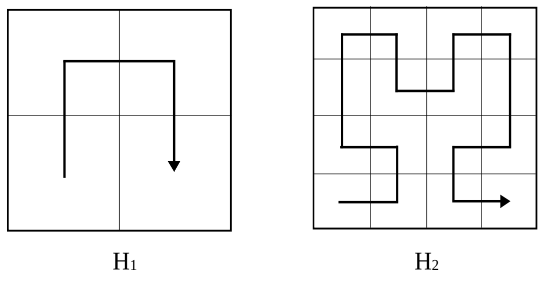


Figure.1 Hilbert curves of order 1 and 2

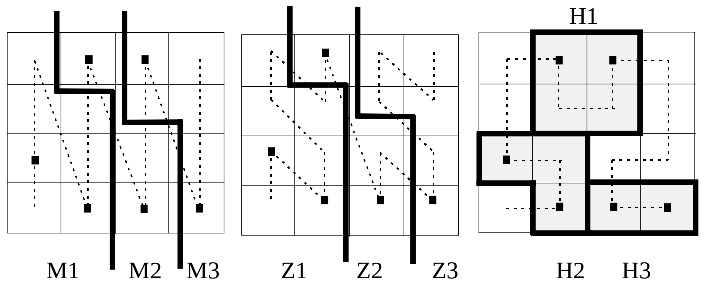
denoted by H1, and the Hilbert curve of order 2, denoted by H2, are shown in Figure 1. The Hilbert curve can be generalized for higher dimensionalities.

The zkdb tree [6], G-tree [7], and the MB$^+$-tree [8] also use linear orders such as z order and column-wise order. However, they have a shortcoming in common with respect to the spatial locality (see Figure 2). The MB$^+$-tree divides the data space into three regions M1, M2 and M3 in this example. This may distribute objects such that the distant objects are clustered in the same region instead of nearby objects. The zkdb tree and G-tree have the same problem as shown in Figure 2(b). On the other hand, the directory regions, H1, H2 and H3 of the Hilbert-tree are compact and the near objects are clustered in the same region.

2.1 Basic Ideas and Properties

The main idea of the Hilbert-tree is to create an indexing scheme that it can support the following:

- When an overflow occurs in a node, try to absorb it and when splitting is necessary, convert two nodes to three nodes. As a result of this the index tree has higher storage utilization.
- Maintain the directory region in a minimal way to reduce the directory coverage.
- Control the correlation between the storage utilization and the directory coverage to compromise the trade-off between them.



(a) MB$^+$-tree (b) zkdb tree (c) Hilbert-tree

Figure 2. The points in a 2-dimensional space are arranged in (a) column-wise scan order (b) Z order (c) Hilbert order. The dashed lines show the ordering path and the heavy lines show the partitioning boundaries. The squares represent data points.

- Maintain the node occupancy not to be fallen below a certain minimum to have predictable and controllable worst-case characteristics.

To absorb splitting we need the ordered list of the objects. We transform Cartesian coordinates in an n-dimensional feature space into locations on the Hilbert curve. Thus, objects are represented by points on the Hilbert curve and we can store them in a sorted order. Through this ordering every node has a well-defined set of siblings and the Hilbert-tree absorbs splitting by redistributing the objects of the oveflowing node into adjacent sibling and adjusting the directory regions. When splitting is necessary, convert the two nodes, the overflowing node and one of the two adjacent siblings, to three. Thus this splitting is called 2-to-3 splitting. When we select one of the two adjacent siblings, we select one that makes the directory coverage minimal. Resulting from this, the Hilbert-tree yields average storage utilization more than 80% and guarantees the worst-cast storage utilization is more than 66.7% (2/3) of full capacity. This concept is similar to that used with the B*-tree [9]. However, the B*-tree operates on a 1-dimensional space and the Hilbert-tree can be viewed as a generalization of it.

To reduce the directory coverage we introduce the concept of minimum bounding interval (MBI) that covers all regions of the lower nodes. This plays a similar role as a minimum bounding rectangle(MBR) used in the SAM such as R-tree[10]. But it does not allow overlap and is not rectangular. For every internal node of the Hilbert-tree, its MBI is stored. Specifically, an internal node in the Hilbert-tree contains at most C_n entries of the form

$$(I, ptr)$$

where C_n is the capacity of an internal node, I is the MBI that encloses all the children of that entry and that is represented by two Hilbert values at either end of the interval, and ptr is a pointer to the child node. We maintain these entries in a Hilbert order. Another advantage of using a linear order is that binary search can be used in searching the entries within a node. When a node is large, the difference between a binary search method, with a log(n) cost and linear search, with an average cost of n/2 is significant, where n is the number of entries in a node.

Although it is known that the trees with best storage utilization may produce nearly best query performance, it is not always the case. Depending on the data distribution, the minimal directory coverage may play more important role than the maximal storage utilization in query performance. From the extensive performance tests we have got the experience that there is a trade-off between them. When we increase the storage utilization by absorbing splitting, there is also a tendency to increase the directory coverage. On the other hand, the storage utilization is reduced if we only intend to reduce the directory coverage. With this insight we can relax the 2/3 minimum node occupancy resulting from 2-to-3 splitting to some extent to reduce the directory coverage. In Hilbert-tree, we can get a good search performance in a wide range of data distribution by controlling these two parameters, the storage utilization and the directory coverage, adaptively depending on data distribution.

2.2 Insertion

To insert an object on the Hilbert-tree, we first calculate the Hilbert value h of the object and traverse the tree using it as a key. In each level we choose the branch with minimum Hilbert distance from h among the entries in the node. Once we reach the leaf level, we insert the object in its correct order.

```
Algorithm Insert(node N, object o)
// Insert object o into tree rooted at N. h is the Hilbert value of o.
{
    I1. Use ChooseLeaf(o, h) to choose a leaf node L  in which to place o.
    I2. Insert o into L in the appropriate place according  to the Hilbert order.
        If L overflows, invoke HandleOverflow(L, o).
    I3. Form a set S that contains L, its adjacent sibling, and the new leaf (if split occurred). Use
        AdjustTree(S) to update the MBIs that have been changed.
    I4. If node split propagation caused the root to split, create a new root.
}
```

2.3 Deletion

A deletion is straightforward, unless it causes an underflow. In such a case, an underflowing node resulting from a deletion can borrow keys from or merge with its adjacent siblings.

2.4 Range Search

The search algorithm starts with the root and examines each branch that intersects the query region recursively following these branches. At the leaf level it reports all entries that intersect the query region as qualified objects.

```
Algorithm RangeSearch(node N, QueryRegion r)
// Perform a query with range r on the tree rooted at N. Ni.child is a child node of node Ni.
```

```
{
 R1. [Search nonleaf node]
      For every entry Ni in N
          Invoke **Overlap(MBR(Ni), r)** to determine whether it is contained in, overlaps, or lies outside r.
          If it is contained in r, output all objects belonging to Ni.
            Else if it overlaps, invoke **RangeSearch(Ni.child, r)**.
 R2. [Search leaf node]
      Output all the objects that intersect r.
}
```

2.5 Nearest Neighbor Search

Nearest-neighbor queries can be handled with a branch-and-bound algorithm [11] on the Hilbert-tree. Two lower and upper distance value bounds δ_{low} and δ_{high} are introduced to order and prune the paths of the search space in the Hilbert-tree:

- δ_{low} gives the minimum distance between a given point and an MBI. However, due to empty space inside the MBIs, the actual nearest neighbor might be much farther than δ_{low}.
- δ_{high} gives the distance between a given point and an MBI, which guarantees the finding of an object in MBI at a Euclidean distance less than or equal to this distance.

Given a query point, the algorithm examines the top-level branches, computes δδlow and ⍰high for the distance, and traverses the most promising branch with the depth first order. At each stage of traversal, the order of search is determined by the nondecreasing order of ⍰low. The objects with the distance to a given query point greater than ⍰high of the farthest MBI and the MBIs with ⍰low greater than the distance between the query point and the farthest object are discarded in each traversal stage.

Algorithm NNSearch(Node N, Point P, Neighbor Near)

/* Return the k nearest neighbors. N is a current node, P is a search point, Near is a list holding the k current nearest neighbors in nondecreasing distance order. BrList is a list holding branches of nonleaf nodes. Ei.child is a child of node Ei */

```
{
 N1. [Search leaf node]
    For every entry Ni in N
        Determine the distance, disti, between P and Ni.
        If disti is less than Near[k].dist
            Assign Ni to the Near[k].
            Rearrange Near in correct order.
```

```
N2. [Search Nonleaf Node]
  For every entry Ni in N
    Compute $\delta_{low}$, $\delta_{high}$ and store them with Ni into a BrList.
    Sort the BrList based on $\delta_{low}$.
    Remove the unnecessary branches in BrList as compared with Near
  For arranged entries Ei in BrList
    Invoke NNSearch(Ei.child, P, Near).
    Remove the unnecessary branches in BrList as compared with Near.
```

3 Experimental Results

To assess the performance of the Hilbert-tree, we implemented it and ran experiments on a four dimensional space. We compared our tree against the buddy-tree. In [12], it is reported that the buddy-tree is the best one among PAMs with respect to the average range query performance. For all operations, we have measured the number of disk accesses per operation.

3.1 Experimental Setup

To experiment we generated 5 groups of 4 dimensional data files that contained different distributions of data: uniform, diagonal, bit, x-parallel, and clustered. Each file contains 100,000 objects. To test the range queries we generated six groups of range queries. The regions of the six groups are squares varying in size which are 0.01%, 0.1%, 1%, 10%, 20%, and 40% of the data space and their centers are uniformly distributed in the data space. To test the nearest neighbor queries, the numbers of neighbors we used are 20, 40, 60, 80 100, and 120. For each experiment, 1,000 randomly generated queries were asked and the results were averaged.

We experimented two types of the Hilbert-tree with minimum storage utilization of 66.7% (2/3) and 25% (1/4), which are abbreviated by H* and H+, respectively, in the results. The buddy-tree is abbreviated by BUDDY.

3.2 Results and Analysis

Tables 1, 2, and 3 show the range query, nearest neighbor query, and insertion costs for each distribution as averages over all six types of queries, respectively. For the sake of an easier comparability, we have normalized the average number of disk accesses for the range and nearest neighbor queries and the average index file size in BUDDY to 100% in each table. In Table 4 we computed the unweighted average over all five distributions.

3.2.1 Search Cost

The Hilbert-tree outperforms the buddy-tree in all range query performance as shown in Table 1. For the nearest-neighbor queries, the Hilbert-tree outperforms the buddy-tree with the exception of the x-parallel distribution. In the x-parallel distribution, the directory coverage of the Hilbert-tree was even larger than that of the buddy-tree relative to other distributions. So the area of the Hilbert-tree to be inspected becomes larger than that of the buddy-tree. Considering Table 4, the Hilbert-tree offers itself to be the winner of our comparison.

3.2.2 Insertion Cost

Since the Hilbert-tree tries to absorb splitting and employs the 2-to-3 splitting, the number of nodes need to be inspected at overflow increases. However, Table 3 shows that there is no clear winner.

3.2.3 Storage Requirements

The Hilbert-tree requires fewer number of nodes (and thus less storage) than the buddy-tree. The savings are around 30%.

Table 1. Average over 6 types of range queries.

	uniform	clustered	bit	x-parallel	diagonal
BUDDY	100.0	100.0	100.0	100.0	100.0
H*	94.8	85.1	74.3	77.5	74.3
H+	96.3	84.3	75.8	78.4	74.5

Table 2. Average over 6 types of nearest neighbor querie.

	uniform	clustered	bit	x-parallel	diagonal
BUDDY	100.0	100.0	100.0	100.0	100.0
H*	92.1	98.4	95.6	124.0	86.5
H+	92.9	97.4	97.6	125.2	90.6

Table 3. Average over 6 types of inserts.

	uniform	clustered	bit	x-parallel	diagonal
BUDDY	4.07	4.70	5.20	5.34	5.24
H*	4.09	4.73	3.51	3.97	4.09
H+	4.67	4.75	4.09	4.59	4.72

Table 4. Average over 5 distributions.

	Range query	NN query	storage utilization	Index size	insert
BUDDY	100.0	100.0	61.8	100.0	4.91
H*	81.2	99.3	84.3	69.4	4.08
H+	81.9	100.7	80.7	72.7	4.56

4 Conclusion

In this paper, we proposed the Hilbert-tree as an access method for geographical databases. Contrary to previously suggested PAMs, the Hilbert-tree tries to absorb splitting and employs the 2-to-3 splitting by using the Hilbert ordering. Using these two properties, the Hilbert-tree achieved the average storage utilization more than 80%. Also, it reduces the directory coverage by maintaining the directory regions as minimal as possible. Moreover, by controlling the trade-off between the maximal storage utilization and the minimal directory coverage, it can cope with a wide range of data distributions

Based on these ideas, we implemented the Hilbert-tree and carried out performance experiments, comparing our method to the buddy-tree. Summarizing the outcome of our comparisons, we can state that the Hilbert-tree exhibits on the average 18% better range query performance than the buddy-tree, and results in a reduction in the size of a tree, and hence its storage cost. The good performance of the Hilbert-tree comes from the overall control over the storage utilization and the directory coverage.

ACKNOWLEDGEMENTS

This research was financially supported by Seoul National University of Science and Technology (2014-0748).

REFERENCES

[1] Peano, Z., *Sur une courbe qui remplit toute une aire plane*. Math. Ann., 1890. 36.

[2] Hilbert, D., *Uber die stetige Abbildung einer Linie auf ein Flachenstuck.* Math. Ann., 1891. 38.

[3] Faloutsos, C., *Gray codes for partial match and range querie*. IEEE Trans. on Software Engineering, 1998. 14(10): p. 1381-1393.

[4] H.V. Jagadish, H.V., *Linear Clustering of Objects with Multiple Attributes*. Proc. ACM SIGMOD Conf., 1990.

[5] Faloutsos, C. and Roseman, S., *Fractals for secondary key retrieval*. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on PODS, 1989.

[6] Orenstein, J.A. and Merrett, T.H., *A Class of Data Structures for Associative Searching*. Proc. 3rd ACM SIGACT-SIGMOD Symposium on PODS, 1984.

[7] Kumar, A., *G-Tree: A New Data Structure for Organizing Multidimensional Data*. IEEE Trans. on Knowledge and Data Engineering, 1994. 6(2): p. 341-347.

[8] Yang,Q., Vellaikal, A. and Dao, S., *MB+-Tree: A New Index Structure for Multimedia Databases*. Proc. Intl. Workshop on Multi-Media Database, 1995.

[9] Comer,D., *The Ubiquitous B-tree*. ACM Computing Surveys, 1979. 11(2): p. 121-137.

[10] Guttman, A., *R-Trees: A Dynamic Index Structure for Spatial Searching*. Proc. ACM SIGMOD Conf., 1984.

[11] Roussopoulos,N., et al., *Nearest Neighbor Queries*. Proc. ACM SIGMOD Conf., 1995.

[12] Seeger, B. and Kriegel, H.-P., *The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems*. Proc. 16th VLDB Conf., 1990.