Filter Trees for Managing Spatial Data Over a Range of Size Granularities

Kenneth C. Sevcik   Nikos Koudas
Computer Systems Research Institute
University of Toronto

Abstract

We introduce a new file organization for the storage and manipulation of spatial (or multi-dimensional) data that is able to execute spatial join operations with great efficiency. The *Filter Tree* information structure is a hierarchical organization that tends to separate spatial entities by size, placing larger entities at the higher levels of the Filter Tree, and smaller entities at lower levels. Within each level, index entries for the entities are ordered by a space-filling curve (Hilbert curve). This allows the algorithms to use bulk I/O requests, exploiting the locality in the index information, and minimizing the number of I/O transfers from disk. We provide algorithms for constructing Filter Trees, for performing range queries on a Filter Tree, and for performing spatial joins between a pair of Filter Trees. In full spatial joins, a minimum number of index block accesses are needed using Filter Trees, as each leaf block is read at most once. We derive some basic properties of Filter Trees, and analyze the performance of the algorithms based on some assumptions about the characteristics of the entities stored. Finally, we include results from experiments using a prototype implementation of Filter Trees to treat both synthetic and real sets of spatial entities. Our experimental results show that full spatial joins can always be done more efficiently with Filter Trees than with current competitive methods, and that in some cases the improvement in performance is very large.

1 Introduction

An ever increasing number and range of applications involve the storage and manipulation of spatial or multi-dimensional data. These applications include geographic information systems, animation, virtual reality, robotics, remote sensing and data mining. In all these applications, some attribute values are drawn from large fully-ordered domains (e.g., coordinate positions in Euclidean space).This makes it possible to view the entities to be stored in the database as occupying positions or hyper-volumes in a multi-dimensional space. Queries investigate the intersections and relative positions in space between pairs of entities and between entities and specified hyper-rectangular volumes.

Many multi-key file organizations have been designed, analyzed, and used. These organizations support the operation of locating all items contained in a specified hyper-range of the space, or locating the item(s) "closest" to a specified point in the space according to some distance metric. Some of these organizations have been extended or generalized to handle items that correspond not to points, but to hyper-volumes in the multi-dimensional space.

In this paper, we introduce a new file structure called *Filter Trees*. We describe algorithms for the construction of Filter Trees and for the processing of range queries and spatial joins on Filter Trees. We demonstrate, using a mixture of analysis and experimentation with a prototype implementation, that Filter Trees have substantial performance advantages over previously proposed file structures in processing some spatial queries of the types needed in applications such as those mentioned earlier.

Filter Trees derive their relative advantages through the principles of *hierarchical representation*, *size separation*, and *locality of accesses*. Filter Trees involve a recursive binary partitioning of the data space in each dimension. Entities associated with a particular level are all grouped together. Each entity is placed at the lowest-level of the tree at which it is fully enclosed by a single cell of the partition at that level. This method of determining the level at which an entity is stored tends to cause larger entities to be stored high in the tree (because they can be contained only in large cells), while smaller entities tend to sink to lower levels of the tree because they fit into smaller cells. Sometimes small entities will be caught at higher levels in the tree because they happen to lie across the boundary between two large cells. However, under reasonable statistical assumptions about where entities are placed, the fraction of such entities is small.

The algorithms for processing Filter Trees are designed to limit the portion of the index and data space that must be explored in order to respond to a query, and also to maximize the degree of locality within the portion of space that is explored. The locality is exploited by using space filling Hilbert curves (of different degrees at different levels of the hierarchy) to order the items stored at a particular level.

The remainder of the paper is organized as follows. Section 2 reviews relevant prior work. Section 3 presents the precise definition of Filter Trees, and describes the algorithms for constructing and using them. In Section 4, the properties and behavior of Filter Trees are analyzed under some simple assumptions about the characteristics of the entities stored in the structure. Section 5 contains experimental results with a prototype implementation of Filter Trees, using both synthetic and real data sets. Various generalizations of the basic Filter Tree are explored in Section 6. Finally, Section 7 recaps the essential points of the paper, and indicates directions for extending the work.

2 Related Work

A primary goal for Object Relational Data Base Management Systems (ORDBMS) [Sto93] and Geographical Information Systems (GIS) is to provide efficient access to multidimensional data. The access methods that provide access paths to such data are called *Spatial Access Methods* (SAMs) or *Multidimensional Access Methods*.

Typical queries for multidimensional access methods are *range queries*, *nearest neighbor queries* and *spatial joins*. Spatial Access Methods have been an active area of research over the years. They can be categorized as either *entity-grouping* or *space-partitioning*. With the entity grouping approach, the way in which entities are clustered and stored in blocks is determined by what items are stored and the order in which they are inserted. Entity grouping approaches include the Grid file [NHS84], K-D-B trees [Rob81] and Point-Quadtrees [Sam90]. Space partitioning approaches impose a regular decomposition of the space and include EXCELL [Tam81], MX-Quadtrees and PR-Quadtrees [Sam90].

Multidimensional objects that are not points can be organized in a multidimensional point access method by transforming the object into a point in a higher dimensionality space. Query execution can proceed with some additional overhead. Hinrichs and Nievergelt report their experience in using the Grid file for such a purpose [HN83].

A very popular alternative way to organize and access multidimensional objects is the R-tree [Gut84]. In an R-tree, multidimensional objects are represented by their Minimum Bounding Rectangle (MBR). An MBR corresponds to a rectangular solid in spaces of two or more dimensions. The MBRs of different objects are allowed to overlap. Consequently the R-tree does not impose any decomposition in the space. Because of the overlap in the index entries, the organization and grouping of data into blocks has great impact on the performance of the method. Several improvements have been proposed on R-trees, like the R^+-tree [SRF87], the R^*-tree [BKSS90] and the Hilbert R-tree [KF94]. All these improvements have the same objective, namely to minimize the problems introduced by overlapping index entries. They either introduce MBR decomposition (R^+-tree) or try to better organize the MBRs into storage blocks, so that the overlap is minimized (R^*-tree and Hilbert R-tree). Among these, Hilbert R-trees were shown experimentally to outperform the alternative R-tree designs for range searches.

Space filling curves [Jag90] have been used for clustering multidimensional objects. Orenstein [Ore86] proposed the use of space filling z-curves (named for the visual pattern they induce) to index multidimensional objects by linearizing the space they occupy. More recently, several researchers have used Hilbert curves which have superior clustering properties at a cost of more complexity [KF93].

Abel and Smith [AS83] first proposed a method to organize rectangles based on their sizes. They used a concept similar to the *locational code* of linear-quadtrees [Gar82] and organized the rectangles in a B-tree index. A similar approach was proposed by Kedem [Ked82] [Sam90]. Filter Trees combine size separation with the use of space filling curves to achieve very efficient execution of spatial joins.

3 Filter Trees

3.1 Assumptions

In this section, we define a new file structure, called *Filter Trees* , that is suitable for the storage and processing of spatial data. We introduce Filter Trees in a rather limited context initially to facilitate the presentation. The initial assumptions are:

- the spatial objects are two-dimensional rectangles,
- their dimensions lie in the range $(2^{-k}, 1)$ for some k, and
- there are no updates to the set of objects.

The two dimensional case is easiest to understand and is most relevant to geographic information systems for representing maps. In Section 6, we will discuss generalizing Filter Trees to handle entities of three or more dimensions, allowing updates, and normalizing arbitrary attribute domains to the range [0,1).

3.2 Definition

In two-dimensional space, we assume that each entity to be stored in the database consists of (1) a *shape*, which is defined by a simple, closed polygon, and (2) additional information. The entity records, which include the shape and the additional information, are stored in blocks to form the bulk of the database. It is desirable (although not mandatory) to cluster the records so that those contained in one block tend to be located close to one another or even overlap in two-space.

From the two-dimensional shape of each object, we may calculate the *minimum bounding rectangle, (MBR)*, which is the smallest rectangle that is aligned with the axes of the two-dimensional space and encloses the entity's shape. The storage of and access to an entity in the Filter Tree is based completely on its MBR. For convenience of exposition, we will refer to the two dimensions as x and y, although their interpretation in specific cases will depend on the application.

The minimum bounding rectangle is specified by the coordinates of its lower left corner (x_l, y_l) and upper right corner (x_h, y_h), where x_l, x_h (respectively, y_l and y_h) are smallest and largest values of the x (respectively y) coordinate, anywhere along the perimeter of the entity's shape. The coordinates of the centre of the MBR are (x_c, y_c), where $x_c = \frac{x_l+x_h}{2}$, and $y_c = \frac{y_l+y_h}{2}$.

Physical storage of both MBRs and entity records requires a serialized ordering of the entities. To obtain this serialized order while retaining locality of overlapping and neighboring entities in two-dimensional space, we map the center of each entity's MBR to a space filling Hilbert curve. The Hilbert curve position of the center of the MBR (x_c, y_c) can be calculated from the binary representations of the coordinates, x_c and y_c. For z curves, this requires only an interleaving of the bits of the binary

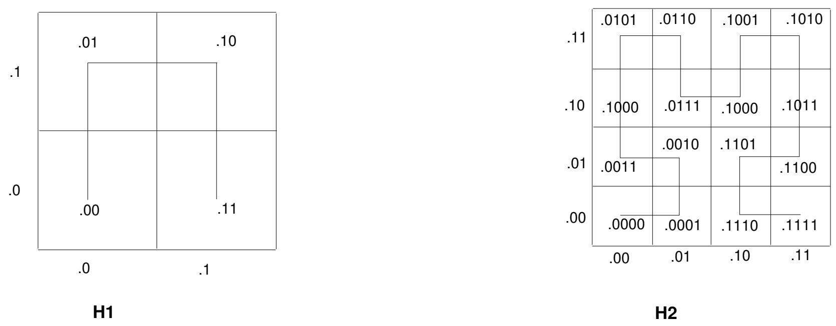


Figure 1: Hilbert Curves of degree 1 (H_1) and 2 (H_2)

representations of x_c and y_c; for Hilbert curves in two dimensions, the transformation is more complex, involving manipulation of bit pairs based on a state transition table. The algorithms and state transition table are available in the appendix.

Our use of Hilbert curves involves relating (x, y) coordinate pairs in the unit square (with k bits of precision) to Hilbert values in the unit line (with $2k$ bits of precision) [1]. Thus each of the 4^k cells in level k of a Filter Tree can be identified either by a pair of k-bit x and y coordinates, or equivalently by the corresponding $2k$ bit binary fraction representing a Hilbert value.

Figure 1 illustrates this relationship for Hilbert curves of degree 1 and 2. Note that:

- The ($2k$ bit) Hilbert value of a level k cell is the prefix of the ($2k$+2 bit) Hilbert values of the four level k+1 subcells, where the four subcells are distinguished by appending 00, 01, 10 and 11 as the least significant bits of the $2k$+2 bit binary fractions.

- For a cell at level k with a $2k$ bit Hilbert curve value, the cell at level k-1 containing the level k cell corresponds to the Hilbert value of the subcell, but truncated after the $2k$-2nd bit.

- For every $2k$ bit binary fraction, b, the corresponding cell is adjacent to the two cells that correspond to $b + 4^{-k}$ and $b - 4^{-k}$ (unless one of these numbers is outside (0,1)).

The last point means that the Hilbert curve is an optimal space-filling curve in the sense that no serialization of the cells can do any better than having every pair of adjacent $2k$ bit binary fractions correspond to cells that are adjacent in two-space.

3.3 Hierarchy of Filters

The Filter Tree is based on a hierarchy of regular grids that divide the unit square into subsquares. At level j, the grid consists of lines at $\frac{k}{2^j}$, $k = 0, \ldots, 2^j$ in both the x and y dimensions. For example, the

[1]Most previous work has enumerated degree k Hilbert curves using the integers 0 to $4^k - 1$. The binary fractions used in our work are precisely the integers used by others divided by 4^k for degree k Hilbert curves.

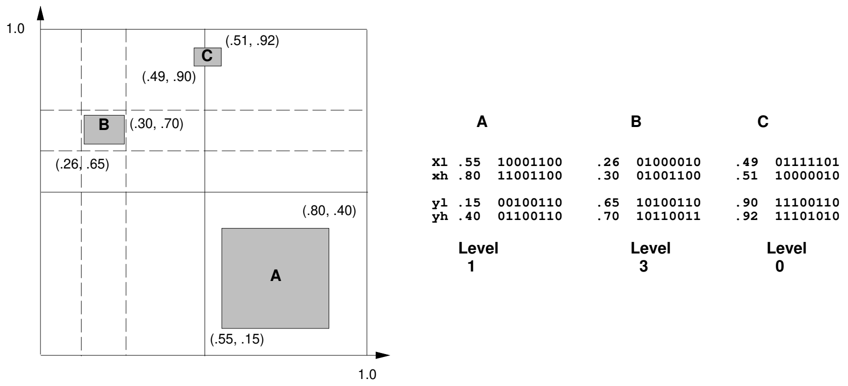


Figure 2: Filter Tree Example

level 3 grid partitions the unit square into 64 squares of size $1/8 \times 1/8$. The hierarchy has L levels, where the smallest MBR's have sides no smaller than 2^{-L}.

Each entity to be stored in the Filter Tree is associated with a level in the tree by examining its MBR. At an intuitive level, we drop the MBR through the grids at the levels of the hierarchy. The MBR of an entity comes to rest at the first level at which its MBR is not fully contained within a single cell. If an MBR has one side of length greater than 2^{-j}, then it will be associated with a level no lower than j. Thus relatively large rectangles are guaranteed to be associated with higher levels in the tree, and relatively small rectangles will tend to be associated with lower levels. According to their locations, however, some small rectangles will be associated with high levels (because they happen to straddle grid lines at high levels).

More mathematically, the level of the hierarchy with which an entity is associated is determined as follows: Express the x and y coordinates of the MBR as binary fractions, and count the number of initial bits in which x_l agrees with x_h and also y_l agrees with y_h. If that number is j, then the entity is associated with level j of the hierarchy.

Figure 2 illustrates this process for three rectangles of differing sizes. Entity A is large and resides at level 1 of the Filter Tree. Entity B is much smaller, and fits within a 1/8 by 1/8 cell, so it is associated with level 3 of the tree. Entity C is smaller still, but its location on the line $x = 1/2$ causes it to be associated with level 0 of the tree.

The bulk of the data in a Filter Tree is located in the *entity records*. Each entity record contains all the information associated with the corresponding entity. There is some advantage to ordering the entity records according to the Hilbert values of the centers of their MBRs so that proximity in two-space is preserved in the serialized entity record file as much as possible.

Entity records are located through *entity descriptors*. For each entity, there is a corresponding entity descriptor stored in the *entity descriptor file*. An entity descriptor contains:

- specification of the minimum bounding rectangle (MBR) of the entity, (x_l, y_l), (x_h, y_h),

- the Hilbert curve coordinate associated with the center of the MBR, $H(x_c, y_c)$,
- a pointer to the disk block in which the corresponding entity record is stored.

The entity descriptor file is organized so that: (1) the descriptors for all the entities associated with a particular level are stored together; and (2) within each level, the descriptors are ordered by the Hilbert value of the centre of their MBRs. A consequence of (2) is that the entities contained in a particular cell of a particular level will all be stored contiguously. The descriptors are packed into blocks, with each block containing about 50 to 100 entity descriptors (assuming 32 bytes per descriptor and a block size in the range of a few kilobytes).

For the part of the entity descriptor file associated with each level of the Filter Tree, there is a *cell index*. The cell index is a B-tree that records the Hilbert value of the last entity descriptor in each block. This requires one (12 byte) entry in the cell index for each block of the entity descriptor file for the level (plus a small additional cost for the upper levels of the B-tree).

3.4 Processing Algorithms

3.4.1 Construction Algorithm

The algorithm for constructing a Filter Tree from a set of entities is given in Figure 3. This algorithm presumes a static set of entity records. The alternative case, where modifications or additions and deletions to the set of entity records are allowed, will be discussed in Section 6.5. In the static case, the effort to construct the tree (sorting the entity records themselves into Hilbert order, packing them into blocks, and storing the blocks contiguously on secondary storage) will be amortized over all the queries answered using the tree.

3.4.2 Spatial Joins Using Filter Trees

In this section, we describe how spatial joins are executed using a Filter Tree structure. Spatial joins deal with correlations of entities between two or more spatial data sets according to some correlation predicate. This predicate can specify conditions on the overlap between two entities, the maximum (minimum) distance between them, etc., and only entity pairs that satisfy the predicate will be included in the spatial join. Spatial joins find many applications in GIS and they are particularly useful in spatial data mining applications [NH94].

Join processing proceeds in two steps. The first step, called the filter step, identifies a list of candidate pairs that might qualify to be in the output. This set is derived based on the partial evaluation of the predicate with the information included in the entity descriptors. The next step, called the refinement step, tests the full predicate against the full entity records for each object pair produced during the filter step. Thus, the purpose of the filter step is to narrow the search space of the refinement step, in order

Given a set of entity descriptors:

- Create from each entity record an entity descriptor:

  1. From the shape of the entity, derive its minimum bounding rectangle.
  2. From the corners of the MBR, determine the level of the Filter Tree at which the entity is to be stored.
  3. From the coordinates of the center of the MBR, derive the Hilbert value associated with the entity.
  4. Include in the entity descriptor a pointer to the block of secondary storage that contains the entity record.

- Create the entity descriptor files for each level of the Filter Tree:

  1. Group the entity descriptors for each level of the tree.
  2. For each level, sort the entity descriptors according to Hilbert value order.
  3. Pack the entity descriptors into contiguous blocks of secondary storage, inserting an entry for each block into the B-tree cell index for the level. The entry specifies the Hilbert value of the last entity descriptor in the block.

Figure 3: Filter Tree Construction Algorithm

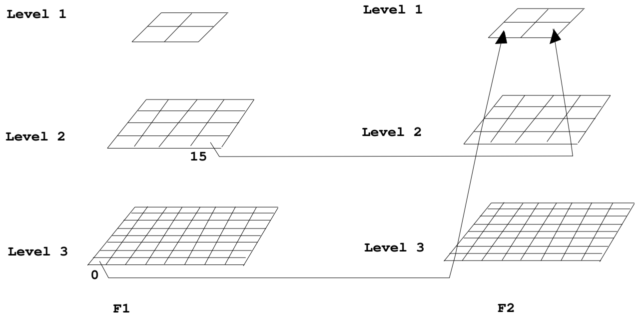


Figure 4: Spatial Join example

to reduce the number of entity records that must be read from disk. Two different indexing methods will identify the same set of candidate pairs and will transfer the same number of entity records from disk, applying the same algorithms for predicate evaluation between them. Therefore, the critical factor in choosing a method for performing spatial joins is the performance of the filter step.

A spatial join between two Filter Trees involves an index sweeping process. However, the structure of the Filter Tree makes the sweeping process very efficient. For any pair of data sets, their full spatial join can be computed with the minimal amount of IO, namely by reading each block of the entity descriptor file at most once.

Consider the hierarchies of filters, F_1 and F_2, shown in figure 4. There are three levels in each hierarchy. If we wish to search for matches between entity descriptors in cell 0 of F_1 and all the cells of F_2 we may restrict our search to cell 0 of F_2 and its enclosing cells at higher levels (in the direction of the arrow in figure 4). No other cells need be considered, since, by the definition of the Filter Tree hierarchy, cells are disjoint. In a similar fashion, matching the descriptors in the 15th cell of level 2 in F_1 involves looking at the corresponding cell in F_2 and its enclosing cell at level 1 only.

The spatial join algorithm is designed to allow every cell at each level of the tree to be processed in this way while reading each block of the descriptor index file only once. This is accomplished by sweeping through the entity descriptor files at each level of each participating Filter Tree in increasing Hilbert value order.

We identify *processing intervals* within the range $(0, 1)$ in terms of end markers taken from each block of the entity descriptor file. Let ${e_{lj}}^F$ be the highest Hilbert value of any entity descriptor in the jth block of level l of the Filter Tree F. There are as many end markers as there are blocks in the entity descriptor files of both trees together. We sort the full set of ${e_{lj}}^F$ values and delete any duplicates. Then the Hilbert value ranges delineated by successive pairs of end marker values in the sorted list, have the property that they are fully contained within one block at each level of each participating tree. Consequently, it is possible to process each such interval in turn while keeping in memory just one block from each level of each tree. When processing of all possible join pairs has progressed to Hilbert value

${e_{lj}}^{F_i}$, then we are done with the jth block of level l of tree F_i, and we replace it by the $j+1$st block of level l, enabling us to proceed with the next processing interval. All ${e_{lj}}^{F_i}$ values are not necessarily unique. Processing intervals ended by non-unique values will simply cause more than one block to be replaced before starting the next processing interval.

Within a processing interval, the following actions are carried out. Levels 0 to L of each tree are addressed in turn, and the spatial join step illustrated in Figure 4 is carried out on the set of entities in the current block of that level. Let ${S_l}^{F_i}(e_n, e_{n+1})$ denote the set of entities in level l of tree F_i that have Hilbert values in the range (e_n, e_{n+1}). By the way the processing intervals were defined, all the entity descriptors in all these sets will be in memory while the processing interval is treated. Then for levels $l = 0, \ldots, L$ in turn, we:

- match entities in ${S_l}^{F_1}(e_n, e_{n+1})$ against those in ${S_{l-i}}^{F_2}(e_n, e_{n+1})$ for $i = 0, \ldots, l$.
- match entities in ${S_l}^{F_2}(e_n, e_{n+1})$ against those in ${S_{l-i}}^{F_1}(e_n, e_{n+1})$ for $i = 1, \ldots, l$

(Note that the ranges of i differ in the two steps in order to avoid matching ${S_l}^{F_1}(e_n, e_{n+1})$ and ${S_l}^{F_2}(e_n, e_{n+1})$ twice.)

The spatial join carried out in this way is as efficient as possible, reading each entity descriptor block only once, and yet requiring that only one block of each level of each tree be in memory at a time. By doubling the (small) memory requirement, a double-buffering technique can be used to overlap the reading of the one block at a particular level with the processing of the previous one.

The basic algorithm for spatial joins as described above can be optimized in some ways at the cost of some increase in complexity. Also, some exceptional cases must be handled (such as unusual situations where many entities have the same Hilbert value to $2L$ bits of precision). More discussion and the complete algorithm are available in the appendix.

3.4.3 Range Queries

In this section, we present the range query algorithm for Filter Trees. Given a query window specified by its lower left and upper right point coordinates, we wish to retrieve all entities in the tree that overlap this window. Assume that the coordinates of the lower left point are (x_l, y_l) and the coordinates of the upper right point are (x_h, y_h). In order to answer the query, we have to search each level in the Filter Tree. However, searching within each level can be very efficient, because we can identify the relevant blocks to fetch.

At each level, each cell that covers any part of the query area must be examined. Within each level, the set of cells to be examined will form a set of Hilbert value intervals. The union of the intervals at level k will be a subset of the intervals at level $k-1$, reflecting the fact that some cells included at level $k-1$ have only one or two (rather than four) subcells included at level k.

Once an interval to be scanned is identified, the cell index can be used to identify the first and last blocks of the entity descriptor file that contain entities with Hilbert values in the interval. Then all blocks from the first through the last can be read with a single IO request.

At lower levels of the tree (say 10 and below), the number of cells is so large that we must avoid having to enumerate all the cells in a range query. (Most cells at these levels will be empty anyway, since the number of cells will surpass the number of entities stored in the tree.) Because the construction algorithm for Filter Trees packs the contents of successive cells (empty and otherwise) into blocks, we need only determine what sequence of blocks contain entities with Hilbert values in a specified range.

In order to determine a set of Hilbert value intervals that together cover all the cells touched by a range query, we use the following approach. We choose a specific level of the Filter Tree, called c, to be the *containment level* for processing the query. This means that intervals to be processed will be identified and expressed with Hilbert values of precision $2c$ bits.

From the query coordinates, $(x_l, y_l), (x_h, y_h)$, we can identify the minimal rectangular set of cells at level c that completely covers the query area. Every interval that passes through the query area starts and ends with one of the cells on the outer border of the rectangular area. Consequently, we can identify all the relevant intervals by traversing the perimeter of the rectangular area, and keeping track of all level c cells that are the start and/or the end of an interval. For each level c cell on the border of the query area, the Hilbert values of the cell and its neighboring cell outside the query area are calculated and compared. (Cells covering the corners of the query will have to be compared with adjacent cells in each dimension.) When the Hilbert value of the border cell is exactly 4^{-c} bigger than that of its neighbor, then that cell is the start of a new interval; when the Hilbert value of the border cell is exactly 4^{-c} smaller than that of its neighbor, the border cell is the end of an interval. By recording all the cells that start intervals and all those that end intervals while traversing the entire query border, and then simply sorting the two sets, all intervals are identified by pairs of entries in the two sorted sets. The algorithm is given in the appendix. Choosing a larger value for c causes the intervals selected to include less marginal area outside the query at the cost of having a larger number of border cells to traverse.

Section 4.3 will present an analysis of the effect of choosing a particular containment level. The appropriate choice depends more on the number of entities stored in the Filter Tree than on the precise dimensions of the query. For around 10,000 entities, level 6 is a good choice, whereas for 10,000,000 entities level 11 is good. The analysis to support these choices is given in Section 4.3

Each interval determined by the steps outlined above can be used to identify a sequence of blocks in the entity descriptor file for each level of the tree. Each sequence of blocks can be read with a single bulk IO request [Wei89][SLM93]. If k is the lowest level of the Filter Tree at which the query area is fully enclosed in a single cell, then only a single interval (or sequence of blocks) will be required at levels 0 through k of the tree. Below level k, there will generally be two or more intervals involved, each corresponding to a sequence of blocks. These sequences of blocks may be adjoining or even overlapping in a single block at the ends. By considering all the sequences of blocks involved for a particular level of the filter tree and merging all sequences that overlap or are adjacent, it is possible to do a minimal number of bulk IO requests to obtain all the relevant entities to the query at that level of the Filter Tree. Note that it may pay to merge two sequences even if they are separated by a block or two rather than adjacent or overlapping, since the single longer bulk IO request including the intervening blocks may be less costly than two IO requests for the sequences separately.

4 Analysis of Filter Trees

In this section, we analyze some properties of Filter Trees. For this purpose, we will make specific assumptions about the distributions of sizes and placements of (the minimum bounding rectangles of) entities stored in the Filter Tree.

4.1 Distribution of Square Objects Over Levels

First, we consider the probability distribution across levels of the Filter Tree of $d \times d$ objects, assuming that the object centers are uniformly distributed over the unit square. At Filter Tree level j, $d \times d$ objects will fall through only if their centers are at least distance $\frac{d}{2}$ from the lines $\frac{i}{2j}$ for $i = 0, 1 \ldots 2j$ in both the x and y dimensions. Thus, in order to fall through level j, the center of a $d \times d$ object must be in one of 4^j squares, each of which has area $(\frac{1}{2^j} - d)^2$. Consequently, the fraction of $d \times d$ objects that fall below level j is $4^j(\frac{1}{2^j} - d)^2 = (1 - 2^j d)^2$.

Since the fraction that fall through level j-1 is $(1 - 2^{j-1}d)^2$, then the fraction that reside precisely level j is

$$f_d(j) = (1 - 2^j d + 4^{j-1}d^2) - (1 - 2^{j+1}d + 4^j d^2) = 2^j d(1 - \frac{3}{4}2^j d) \tag{1}$$

Knowing that the cumulative total at levels 0 through j is $2^{j+1}d - 4^j d^2$, we can conclude that the distribution of level occupancy for $d \times d$ objects is:

$$f_d(j) = \begin{cases} d(2-d) & j = 0 \\ 2^j d(1 - \frac{3}{4}2^j d) & j = 1, \ldots, k(d) - 1 \\ (1 - \frac{1}{2}2^{k(d)}d)^2 & j = k(d) \end{cases} \tag{2}$$

where $k(d) = \lceil -\log_2 d \rceil$ is the lowest level to which any $d \times d$ object can fall (since d must be less than 2^{-k}). Then the average level occupied in the Filter Tree by $d \times d$ objects is:

$$I(d) = \sum_{j=0}^{k(d)} j f_d(j) \tag{3}$$

Table 1 illustrates the distribution $f_j(d)$ for various values of d. Note that, for the internal levels of the tree, since $2^j d = 2^{j+1}\frac{d}{2}$, $f_d(j) = f_{d/2}(j+1)$.

Level	*d=1*	*d=1/2*	*d=1/4*	*d=1/8*	*d = 1/16*	*d=1/32*	*d=1/64*	*d=1/128*	*d=1/256*
0	1	0.75	0.4375	0.2344	0.1211	0.0615	0.0310	0.0156	0.0078
1		0.25	0.3125	0.2031	0.1133	0.0596	0.0305	0.0154	0.0078
2			0.2500	0.3125	0.2031	0.1133	0.0596	0.0305	0.0154
3				0.2500	0.3125	0.2031	0.1133	0.0596	0.0305
4					0.2500	0.3125	0.2031	0.1133	0.0596
5						0.2500	0.3125	0.2031	0.1133
6							0.2500	0.3125	0.2031
7								0.2500	0.3125
8									0.2500
$I(d)$	0	.25	.8125	1.5781	2.4570	3.3955	4.3645	5.4389	6.3411

Table 1: *Distributions of level occupancies for* $d = 2^{-j}$, $j = 0, \dots 8$

The distribution of level occupancy and the average level occupied by squares of size d both indicate that only a small fraction of small squares are caught at levels significantly higher than the lowest level to which squares of their size can fall. This demonstrates the size separation achieved by the Filter Tree structure.

If the probability density function of the sizes of objects to be stored in the Filter Tree is $p(d)$, then the aggregate distribution of level occupancy is given by:

$$f(j) = \int_{t=0}^{\infty} p(t) f_t(j) dt \tag{4}$$

There are many possible assumptions that can be made about the distribution $p(d)$. One of interest is that the average coverage of the unit square is the same for objects of all allowed sizes (that is, $\frac{p(d')}{d'^2} = \frac{p(d'')}{d''^2}$ for any d', d'' pair). If the bounds on object sizes are d_{min} and d_{max}, then

$$p(d) = \frac{1}{d^2}\left(\frac{1}{d_{min}} - \frac{1}{d_{max}}\right) \tag{5}$$

For this case,

$$f(j) = \frac{\int_{t=d_{min}}^{d_{max}} \frac{f_j(t)}{t^2} dt}{\left(\frac{1}{d_{min}} - \frac{1}{d_{max}}\right)} \tag{6}$$

Some of our experiments (presented in Section 5) are based on this distribution.

4.2 Distribution of Rectangular Entities over Levels

The analysis of section 4.1 can be generalized to apply to rectangular entities rather than square ones. For rectangles of size $d_1 \times d_2$, the fraction that fall below level j is given by

$$4^j(\frac{1}{2^j} - d_1)(\frac{1}{2^j} - d_2) = 1 - 2^j(d_1 + d_2) + 2^{2j}d_1d_2 \tag{7}$$

Since the fraction that fall through level $j-1$ is $1 - 2^{j-1}(d_1+d_2) + 2^{2(j-1)}d_1d_2$, the fraction that reside at level j is

$$f_{d_1d_2} = 2^{j-1}(d_1+d_2) - \frac{3}{4}2^{2j}d_1d_2 \tag{8}$$

So the overall distribution is given by:

$$f_{d_1d_2}(j) = \begin{cases} d_1 + d_2 - d_1d_2 & j = 0 \\ 2^{j-1}(d_1+d_2) - \frac{3}{4}2^{2j}d_1d_2 & j = 1, \ldots, k(d) - 1 \\ 1 - 2^{k-1}(d_1+d_2) + 2^{2(k-1)}d_1d_2 & j = k(d) \end{cases} \tag{9}$$

where $k(d) = \lceil -\log_2(max(d_1, d_2)) \rceil$.

4.3 Range Query Precision and Cost

In describing the algorithm for processing range queries in Section 3.4.3, we pointed out the importance of limiting the total length of the Hilbert value ranges that are searched to process the range query. Here we analyze the tradeoff between the computation invested to restrict the ranges and the excess portion of space searched outside the query area.

At each level of the Filter Tree, we must examine each cell that is either enclosed or intersected by the border of the query range. For lower levels of the tree, however, there are too many cells to consider each one individually. Instead, we choose (carefully) a particular level of the Filter Tree the containment level, c, and calculate the minimal set of Hilbert value ranges required to cover all cells at that level that are contained in or overlap the query range.

Consider now a particular range query with dimensions d_x by d_y, and a chosen containment level, c. Figure 5 illustrates the situation that must hold whenever $\min(d_x, d_y) > 2^{-c+1}$. The range query processing algorithm described in Section 3.4.3 identifies and scans all the Hilbert value ranges that cover the $n_x \times n_y$ cells at level c. The cost of identifying the ranges is the calculation of the Hilbert value for each boundary cell and their external neighboring cells. This requires a total of $4(n_x + n_y - 1)$ calculations of a Hilbert value from (x, y) coordinate pairs.

The portion of Hilbert value ranges searched unnecessarily (because it is outside the query area but inside the bordering cells) is $n_xn_y2^{-2c} - d_xd_y$. Allowing for the worst possible dimensions, d_x and d_y,

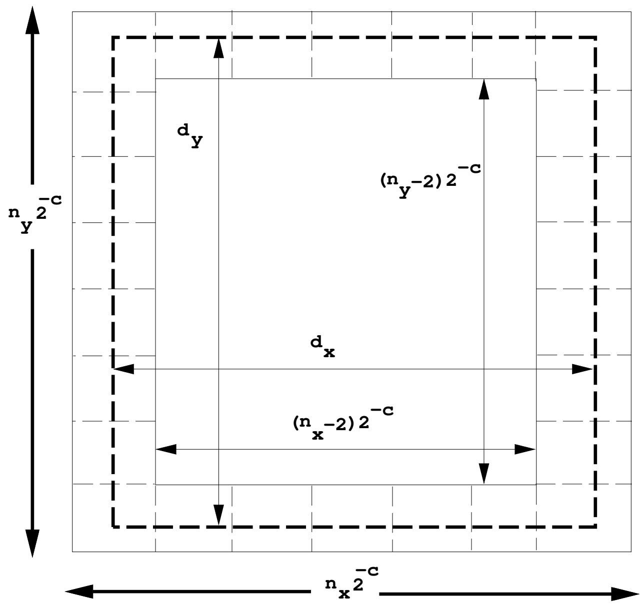


Figure 5: The border of a range query covered by a rectangle of a cell at level c of the Filter Tree

and the worst possible alignment of the query with the cells at level c, an upper bound on the proportion of the space scanned unnecessarily, W, is given by:

$$W = n_x n_y 2^{-2c} - d_x d_y \leq (d_x 2^c + 2)(d_y 2^c + 2)2^{-2c} - d_x d_y = 2^{-(c-1)}(d_x + d_y + 2^{-(c-1)}) \qquad (10)$$

since $d_x \geq (n_x - 2)2^{-c}$ and $d_y \geq (n_y - 2)2^{-c}$. Expressing the number of required Hilbert value calculations, n_H, in terms of d_x and d_y, we have:

$$n_H = 4(n_x + n_y - 1) \leq 4((d_x + d_y)2^c + 3) \qquad (11)$$

For a range query of dimensions $d_x \times d_y$ on a Filter Tree that stores N spatial entities, we would like the containment level, c, to provide an appropriate tradeoff between the computation required (n_H) and the excess area scanned (W). The fact that increasing c by one roughly halves W while roughly doubling n_H suggests that any weighted sum of W and n_H will have a concave upward shape indicating the existence of an optimal c value that minimizes the function. Further, the optimal c value will be one for which the two components of the cost function have approximately equal magnitude. In particular consider minimizing the cost function:

$$C_{total} = (n_H \times c \times C_H) + (W \times \frac{N}{b} \times C_B) \qquad (12)$$

where:

- C_H = processor time required per level to convert (x, y) to a Hilbert value,

N	*best* c
10,000	6 or 7
100,000	8
1,000,000	9 or 10
10,000,000	11

Table 2: *Best values of* c *for different data base sizes*

- C_B = cost of reading and scanning a block of entity descriptors, and
- b = blocking factor of entity descriptors.

The first term represents the cost of doing all the Hilbert value calculations of cells along the border of the query, and the second term estimates the cost of processing Hilbert value ranges outside the query if the intervals to be scanned are chosen at level c. Thus, the best choice of c is the one for which:

$$n_H \times c \times C_H \cong W \times \frac{N}{b} \times C_B \tag{13}$$

or

$$4((d_x + d_y)2^c + 3) \times c \times C_H \cong 2^{-(c-1)}(d_x + d_y + 2^{-(c-1)})\frac{N}{b} \times C_B \tag{14}$$

Retaining only dominant terms on each side and dividing by $(d_x + d_y)$ yields:

$$c \times 2^{c+2} \times C_H \cong \frac{N}{b} \times 2^{-(c-1)} \times C_B \tag{15}$$

or

$$\boxed{c \times 2^{2c+1} \cong \frac{N \times C_B}{b \times C_H}} \tag{16}$$

In our implementation b is about 60 and $c \times C_H$ is very close to $c \times 0.05$ milliseconds. If we assume C_B is about 30 milliseconds then the equation becomes $c \times 2^{2c+1} \cong 10 \times N$. Based on these assumed parameter values, Table 2 shows the best choice of c for various values of N.

By retaining only the dominant terms of equation 14 in equation 15, the dependence of choice of c on the query dimensions is lost. While in general we can afford a slightly larger value of c for queries with smaller dimensions (because their perimeters are smaller), this is a secondary effect. It is sufficient to choose c once for each Filter Tree according to the number of entities it contains.

The right hand side of equation 14 provides a conservative estimate of the excess cost of retrieving blocks that contain no entries relevant to the query. Depending on how the entity descriptors are packed into blocks, there may be fewer blocks composed completely of entities outside the query than is suggested by the estimate in the right hand side of equation 14. This means that smaller values of c may suffice to limit the number of blocks accessed as much as possible. Thus, the values in Table 2 can be interpreted as the largest values of c that would pay off if the distribution of actual entities led to the best possible packing of entity descriptors into blocks.

5 Experimental Results

In order to assess the performance benefits and limitations of Filter Trees, we conducted a series of experiments involving spatial joins and range queries, using the algorithms described in Section 3.4. We experimented with both real and synthetic data sets [2].

The Filter Trees in our experiments have at most 21 levels (numbered 0 to 20), because the real data sets that were available to us produce at most 21 levels in their Filter Tree representation. We used data sets extracted from the TIGER data file of US Bureau of the Census [Bur91]. The first one consisted of 53,145 line segments representing road segments from Long Beach County, California. We will refer to this set as the *LB* data set. The second file consisted of 39,068 lines segments representing road segments in Montgomery County, Maryland. We will refer to this file as the *MG* data set. We used these data sets because they have been used previously by other researchers. While they are "real", it is unlikely they are "typical" of spatial data sets because they treat sequences of highway segments and hence exhibit a low degree of overlap among intervals. For that reason, we also generated some synthetic data sets using various discrete probability distributions.

In a Filter Tree, the distribution of the sizes of the entities is of great importance, since it determines the occupancy of each level. We experimented with two distributions to generate synthetic data sets:

- "equal area coverage " (equation 5). This is the distribution used in our analysis in Section 4.1. We generated one data set following this distribution, having 50,000 descriptors in levels 5 to 12 [3]. We refer to this as the *EA* data set.
- triangular shaped distribution. Given a "peak" level and the min and max levels, the sizes of MBRs in the synthetic data set has a triangular shaped distribution.

More formally the "triangular distribution" is defined as:

$$p(d) = \begin{cases} x_1 + \frac{d-x_1}{x_2-x_1}\left(\frac{2}{x_3-x_1}\right) & x_1 \le d \le x_2 \\ x_3 - \frac{x_3-d}{x_3-x_2}\left(\frac{2}{x_3-x_1}\right) & x_2 \le d \le x_3 \end{cases} \tag{17}$$

[2] All of our experiments were conducted on a sparc 20 with a 60MHz Supersparc+, SPEC Int 92 4492, SPEC FP 92 4888.

[3] We chose to restrict the size of the Filter Tree for this data set, due to the practical problem of generating the distribution of equation 5 with 50,000 descriptors over 21 levels.

where x_1, x_3, x_2 correspond to the minimum, maximum and peak level. The motivation for using the triangular distribution came from observation of the distributions of the sizes in the LB and MG data sets. Using the "triangular distribution", we generated two synthetic data sets with 50,000 descriptors: Set TR_1 was generated using $x_1 = 4, x_2 = 17, x_3 = 20$ and TR_2 using $x_1 = 4, x_2 = 15, x_3 = 20$.

For all the experiments we conducted with Filter Trees, we present the corresponding performance of Hilbert R-trees for comparison. We chose to present performance numbers for Hilbert R-trees because they outperform all other variants in the R-tree family for range queries and we expect they are also better for spatial joins due to their clustering properties [KF94].

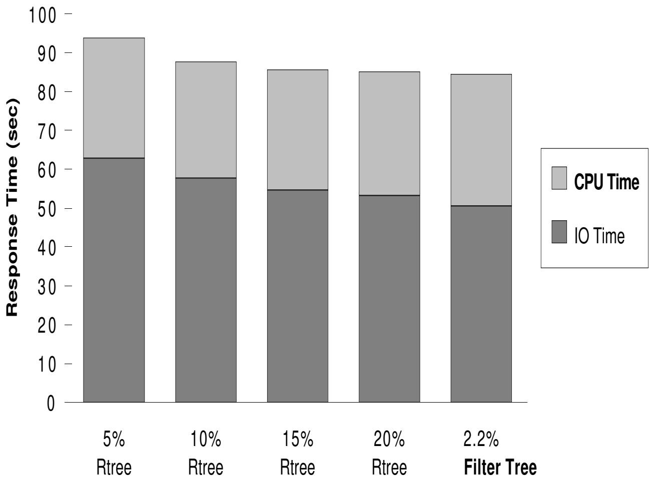
5.1 Spatial Joins

We present and discuss the experimental results obtained from the application of our spatial join algorithm to the real and synthetic data sets. For all experiments, we present the measured response time and the proportions of IO and CPU time. The estimates for IO time are obtained by precisely counting the number of IO operations occuring during each experiment and charging 30 ms for each IO operation. CPU time is then the measured response time minus the estimated IO time. Although these estimates are not exact, they suggest the balance between IO and CPU time in the join algorithms. Leaf index blocks have exactly the same structure for both access methods. However the index fanout of the Filter Tree is much higher than that for R-trees. For our prototype implementation the fanout for Filter Trees was 63 and for R-trees 42 ($\sim 34\%$ higher for Filter Trees). In an enhanced implementation, Filter Trees can have fanout up to three times bigger than that of R-trees.

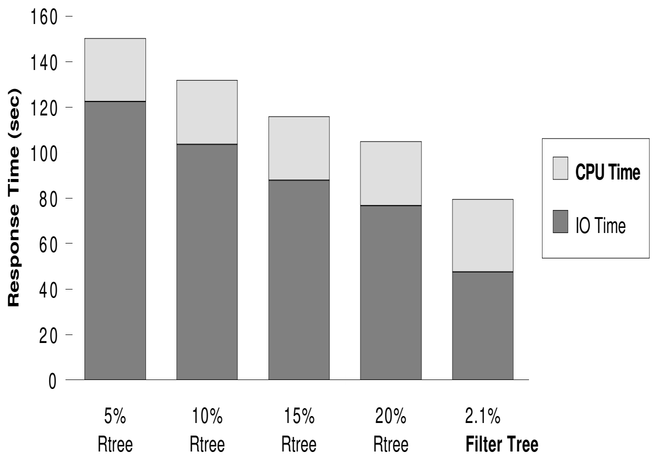
For comparison with the Filter Tree join algorithm, we implemented the best R-tree join algorithm proposed by Brinkhoff et al. [BKS93]. The R-tree join algorithm involves an index sweeping process. When the indexes have the same height, the algorithm proceeds top-down sweeping index blocks at the same height. At a specific height, the pairs of overlapping descriptors are computed and, at the same time, the rectangles of their intersections are computed also. This information is used to guide the search in the lower levels, since descriptors not overlapping the rectangle of intersection of their parents need not be considered for the join. The algorithm uses a buffer pinning technique that follows a greedy approach trying to keep relevant blocks in the buffer in order to minimize block re-reads. When the indexes do not have the same height, the algorithm proceeds as described above up to a certain point and then degenerates into a series of range queries.

For all the experiments, we assumed that the R-tree indexes and the Filter Tree cell indexes fit entirely in main memory. This is a realistic assumption even for large data files and it is especially true for Filter Trees since the index size is smaller than for R-trees for most data sets. For our spatial join experiments we experimented with the following types of joins: (a) self joins joining a data set with itself (which is useful in identifying pairs of overlaps within a data set) and (b) joining two distinct data sets. For the latter, we used two different alternatives:

- joining one of LB and MG with one of TR_1 and TR_2, or
- joining a data set (D) and a synthetic data set (D') generated from D as follows: If x_{max} and y_{max} are largest sizes of any entity in D in the x and y dimensions, respectively, then for each entity in D, we generate a new entity in D' having as a lower left point the center of the entity from D and sizes in x and y uniformly distributed between zero and x_{max} and y_{max} respectively. That way, a synthetic set with statistical properties similar to D is generated.



(a) LB data set, self join



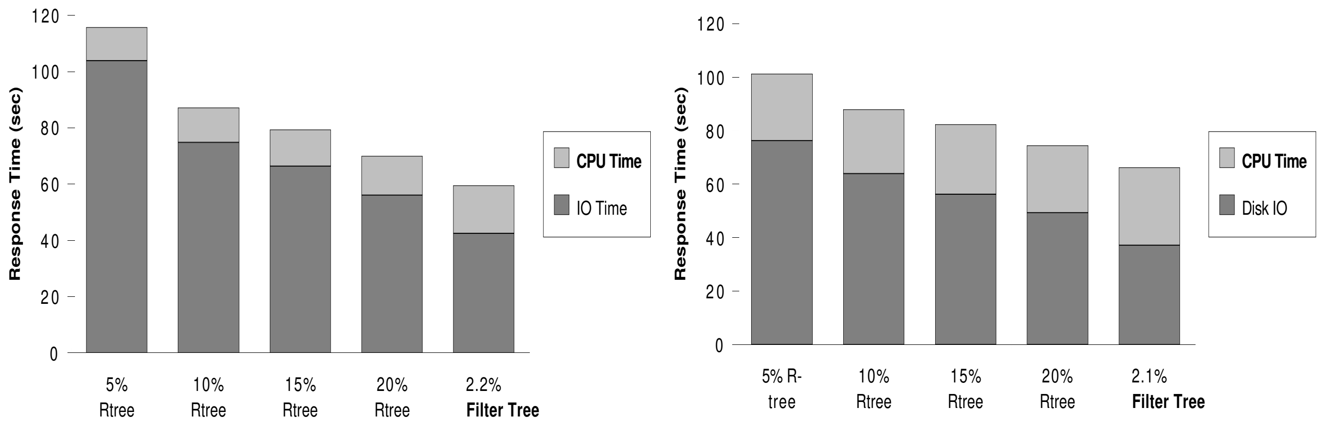
(b) EA data set, self join

Figure 6: Performance of self joins for real and synthetic data sets

In figure 6a, we present the performance of self joins for the LB data set, for R-trees and Filter Trees. For the R-tree join, we varied the buffer size available during the join operation and we present it as a percentage of the total number of blocks of both files. Increased buffer size improves the IO behavior of the R-tree join algorithm. This basically means that the buffer hit ratio increases, since more blocks can stay memory resident. The buffer pinning part of the R-tree join algorithm tries to minimize the number of re-reads for data blocks and the increased buffer size obviously helps.

The LB data set in its Filter Tree representation has 19 levels. This means that the Filter Tree join can proceed with only 38 blocks of buffer space, which is only 2.2% of the total set of blocks. Filter

Trees provide 10% savings in response time when 5% buffering is available for R-trees. The Filter Tree performance is matched by the R-tree when 20% buffering of the underlying space is provided to the R-tree. Figure 6b presents the results of the same experiment using the EA data set. Filter Trees can perform the join with almost 50% savings in response time with 2.1% buffer space, relative to an R-tree with 5% buffering. Even with 20% buffering available for R-trees, Filter Trees still achieve 23% savings requiring only 2.1% buffer space.

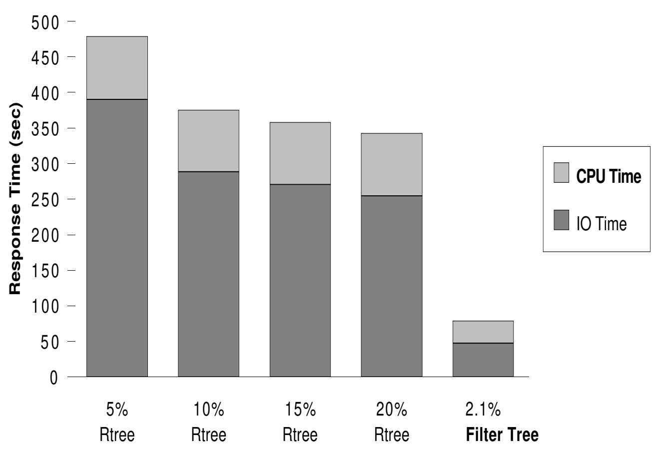


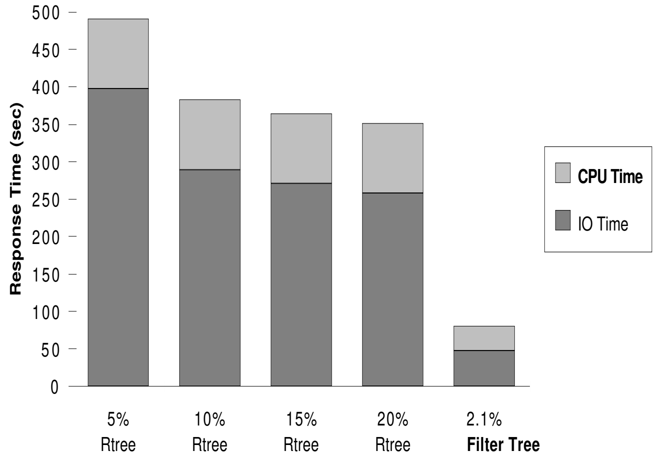
(a) Join performance results for MG and TR_1 (b) Join performance results for MG and MG'

Figure 7: Join performance for R-trees and Filter trees using the MG data set

Figures 7a,b present experimental results for the join performance of Filter Trees and R-trees, using the MG data. Filter Trees perform the best in both cases achieving 32% and 23% savings in response time respectively relative to the R-tree with 5% buffering case.

Figures 8a,b present join results for the TR_1 and TR_2 data sets with sets having similar statistical properties. The general trends for the performance of the R-tree join algorithm remain the same, with increased buffer size improving the total response time. However, for these data sets, the buffer pinning mechanism of the R-tree algorithm is not so effective, since a higher buffering percentage is needed in order for the algorithm to attain IO efficiency. In particular, even with 20% buffering, R-trees have to read each block three times on average to perform the join. The Filter Tree join algorithm can proceed with only 34 blocks, which is 2.1% of the total file size. Comparing figures 8a and 8b, it is interesting





(a) Join performance results for TR_1 and $TR_1{}'$ (b) Join performance results for $TR_2{}'$

Figure 8: Join performance for R-trees and Filter-trees on Synthetic Data sets

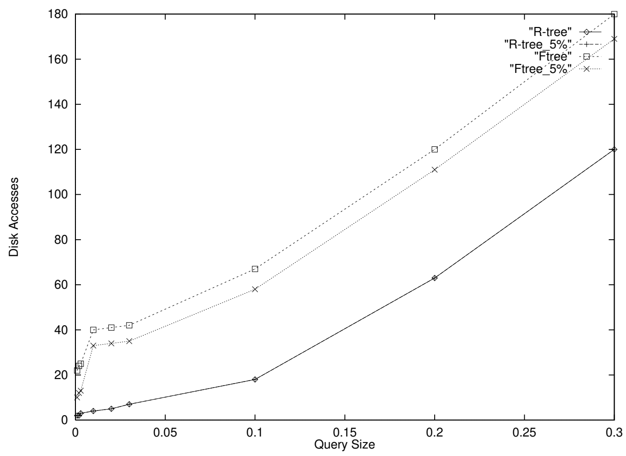
to note that, as the peak of the distribution is shifted toward lower levels, the R-tree join algorithm becomes less efficient. An increase in the number of larger entities in the file causes more ambiguity in the R-tree index. As a consequence, the IO and CPU time requirements of the R-tree join algorithm is higher[4].

We expect that, in the scope of real life spatial data base applications, the performance benefits of the Filter Tree approach will range somewhere between those reported for the LB and MG data sets (in figure 6) and those for the TR_1 and TR_2 data sets (in figure 8).

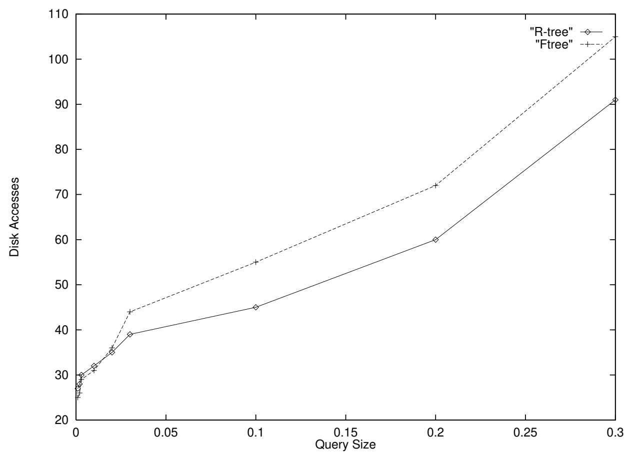
5.2 Range Queries

In this section, we present experimental results for the performance of range queries on Filter Trees. Filter Trees, due to their size separation principle, require at least one disk access at every level of the tree in order to answer a range query. We present the total number of blocks transfered versus query sizes for one real data set (LB) and one synthetic data set (TR_1).

[4]The above observations hold for an additional distribution we experimented with, in which the descriptor fraction at each level follows the Zipf distribution.



(a) LB Data set



(b) TR_1 Data set

Figure 9: Range Query Performance for Real and Synthetic Data sets

We processed 100 random queries inside LB and TR_1 and we present the average number of disk accesses per query. Figure 9a presents the results of two experiments. For the first experiment, buffering was turned off. In figure 9a, we can see that R-trees perform better for range searches on LB. For small queries (on the order of 0.001 of the space) R-trees perform an average of 3-4 disk accesses to answer the queries. The same queries, in Filter Trees, require one disk access for each level and incur a higher cost. As the query size increases, both R-trees and Filter Trees require more disk accesses on the average. For the second experiment shown in figure 9a, we provided 5% buffering of the total file size for both R-trees and Filter Trees. For R-trees no improvement is observed. This is expected since the queries are random. However for Filter Trees, the average cost of each disk access is lower for any query size, because the cost of visiting each level in the hierarchy is amortized over all queries. For Filter Trees, lower levels are likely to fit in a single block (as it is the case for levels 2 and 3 of the LB set). These levels, as well as other levels with few blocks per level, will remain in memory, as each query will use them. Consequently the cost of accessing these level is amortized over all queries.

Figure 9b presents range query performance results for TR_1. For this set, R-tree searches are not very efficient. Small queries require on average almost 20 disk accesses. For this data set, Filter Trees are able to perform a little better for range searches, since they need about one disk access per level (TR_1 has 17 levels). When the degree of overlap between MBRs becomes larger and the index height increases, R-tree searches become inefficient, because the search follows many paths down to the leaves and often finds nothing relevant. Filter Trees can adapt better to distributions with high overlap between MBRs. Their performance for large range queries remains worse than R-trees however.

6 Generalizations and Extensions

In this section, we describe several generalizations to Filter Trees as described so far.

6.1 General Attribute Domains and Distributions

When the domains of the attributes that define the multidimensional space are not [0,1) and/or the distributions of the attribute values are known to be non-uniform, transformation of the attribute values is required before the Filter Tree is constructed.

Given spatial objects in two-space, we map them into the unit square, so that the coordinates of all vertices of the entities are in the range [0, 1). This normalization is accomplished as follows.

Let $F_x(x)$ be a continuous and invertible approximation to the cumulative distribution function (CDF) of the minimum and maximum coordinate value for each entity in the first dimension, and $F_y(y)$ be the corresponding CDF of vertex coordinates in the second dimension.

Then we map the coordinates of each vertex of each entity into the unit square by applying the inverse functions:

$$x = {F_x}^{-1}(x^{'}), \quad y = {F_y}^{-1}(x^{'}) \tag{18}$$

where $(x^{'}, y^{'})$ are the coordinates of the point in the original space.

If the locations of the entities in the two dimensions are independent, then the transformation will cause entities to be mapped uniformly over the unit square. If they are not independent and the correlation is known, then the dependence can be taken into account in the inverse mapping of one of the two coordinate values in order to once again approach an approximately uniform distribution of entities after the transformation.

6.2 Three and Higher Dimensions

While there are many applications based on two-dimensional spatial entities, many others require use of three or even more dimensions. The two basic mechanisms of the Filter Tree, namely binary recursive partitioning and Hilbert curve ordering, both generalize to higher dimensions [Jag90]. The number of cells at level j with k dimensions is k^j. Consequently, the number of levels that it is practical to use decreases as k increases.

In the range query implementation, it is necessary to identify all Hilbert value intervals that pass through the hyperrectangular range specified by the query. As seen in Section 3.4.3, in the two-dimensional case, range query processing involved calculating the Hilbert values for the pairs of cells along the border of the query rectangle at a selected level of the tree. In higher dimensions, the number of pairs of boundary cells increases more quickly from level to level. In general a higher containment (smaller c) level will be chosen to determine the Hilbert value intervals with which to perform queries, in order to control the number of cell pairs on the query border that must be considered.

6.3 Supporting Points and Lines

Many applications that involve two-dimensional entities also must include zero-dimensional (points) and one-dimensional (lines) entities. Points can be accommodated by adding one additional level (level infinity) to the Filter Tree Hierarchy. The range query and spatial join algorithms can then be extended to handle the additional level in a manner compatible with its processing of the other levels. In this case level infinity of the filter tree will function as a point quadtree structure [Sam90].

Lines or curves, which might represent roads, rivers or borders in a mapping application, are more challenging to accommodate in Filter Trees. If many lines or curves extend across much of the two dimensional space, then treating them as a whole with the minimum bounding rectangle approach will cause them to be stored in high levels of the Filter Tree hierarchy. This would not lead to efficient processing of queries involving the lines and curves. To alleviate this problem, each linear entity can be decomposed into segments, with each segment then being stored in the Filter Tree. For line data, Filter Trees could behave as a PMR-Quadtree and use additional optimizations introduced for that structure [NS86]. With Filter Trees, it is advantageous to choose the segmentation of a linear entity so that segments are contained within a Filter Tree cell at some chosen level.

6.4 Multilevel Representations and Embedding in a Deep Hierarchy

The levels of Filter Trees can be thought of as corresponding to levels of resolution in the representation of the portion of the world modeled by the spatial data set. One range of relevant resolutions might correspond to continents and countries. Another range of resolutions at a lower level might represent cities and buildings. At a still lower (finer) level, a range of resolutions might reflect air ducts, wiring and plumbing connections in individual portions of buildings.

Conceptually, all these ranges of resolutions can be thought of as a continuum, and any particular Filter Tree represents spatial entities only over a small part of the full range. In this context, there is reason to represent a single entity at more than one level of representation in a set of related Filter Trees. For example, a highway or river might appear as a line-like entity at coarser levels of resolution, and also appear as entities with both length and width at finer resolutions corresponding to, for example, city maps. When entities are represented at multiple levels in this way, the benefit is that the major characteristics of the entity can be noted when processing only the higher levels of resolution of the Filter Tree, yet more complete details are available by processing lower levels. The price to be paid for this benefit is the need to coordinate and keep consistent the information stored at various levels about a single spatial entity.

Between the distances dealt with by astronomers and those dealt with by particle physicists there are fewer than one hundred binary orders of magnitude. Since individual Filter Trees can represent 10 to 15 (or even up to 30) levels, only a few levels of coordinated Filter Trees suffice to cover the full range of size resolutions of interest in the world.

The telecommunications utility of Brazil has a project underway currently to represent all types of telecommunications connectivity in the country [Mag95]. At the larger scale, there are major trunk connections linking different cities and regions. At a medium scale, shared and leased lines connect different points within a city. At a smaller scale, associated with each building or block there is a routing matrix that specifies connections to individual appliances (telephones, televisions, computers etc.). Filter trees could provide a means for integrating all this data into a consistent framework spanning all scales.

6.5 Allowing Updates to Spatial Data Sets Stored in Filter Trees

Many (perhaps most) spatial database applications deal with static or nearly static sets of spatial entities. In this case, the Filter Tree structure can be constructed as described in Section 3.4. This approach takes a great deal of initial processing, but blocks of entity records and entity descriptors are fully packed and are stored contiguously in Hilbert value order, which permits highly efficient processing.

However, there are other spatial database applications in which updates reflect the addition, deletion, movement, and transformation of spatial entities, and these updates occur interleaved with queries through the lifetime of the Filter Tree. To handle such applications with Filter Trees, some changes in methods are needed. To moderate the need for frequent reorganizations, some space can be systematically left in each block to allow for insertions and modifications so the set of spatial entities. As has been done with variations of B-trees, by modifying pairs or triples of adjacent blocks together, it is possible to maintain storage utilization at high-levels (80% to 90%) and still process updates efficiently. Correspondingly, the processing of range queries and spatial joins will generally require 10% to 20% more block transfers than in the no updates case.

7 Conclusions

We have presented Filter Trees, an efficient structure for performing spatial join operations between sets of spatial objects. The Filter Tree structure is based on three principles:

- Hierarchical Representation – Each entity is associated with a level that corresponds to a particular granularity of space partitioning.

- Size Separation – Entities of different sizes tend to be associated with different levels of the tree.

- Spatial Locality – Within each level, entities are ordered by their positions along a space-filling Hilbert curve in order to cause entities in a portion of the multidimensional space to map to contiguous portions of the linear storage space as much as possible.

Together these principles lead to a file structure that is capable of supporting spatial joins more efficiently than alternatives that have been proposed and evaluated previously. The experimental results described in Section 5 provided evidence of the benefits of using Filter Trees when spatial joins are frequent. For synthetic databases in which the spatial entities are randomly positioned, Filter Trees proved to require much less I/O for spatial joins relative to R-trees, but they required significantly more I/O for responding to range queries. Consequently, the choice between Filter Trees and R-trees depends on the frequency of spatial joins in the workload. Space partitioning, as provided by Filter Trees and PMR-Quadtrees [NS86], is a good approach when spatial join operations are involved between sets of line segments or sets of approximated spatial objects. This fact is supported both by our results for Filter Trees and the experimental results provided by Hoel and Samet [HS95].

For the LB and MG data sets, the advantage of Filter Trees relative to R-trees for doing spatial joins was much reduced. It is possible that this is due to the special structure of those databases where the entities stored are MBR's of successive highway segments, which by definition tend to not overlap extensively. We believe that many real spatial databases that store two dimensional objects (as opposed to the more one dimensional highway segments in the LB and MG data sets) will lead to performance in between the extremes seen in our experiments. Evidence for this is provided by our experiments with spatial joins of the MG data set with synthetic sets, where savings of 23% and 32% were observed for the two join strategies we investigated.

Future work will involve expanding the functionality of Filter Trees by adding a capability for other query types such as "nearest-neighbor". For R-trees, a branch and bound algorithm along with distance metrics was proposed recently by Roussopoulos et al. [RKV95]. We believe that nearest neighbor search using Filter Trees can be performed with a bounded number of IO operations, whereas no such bound exists for R-trees. In addition, our plans for future work include the consideration of various multi-dimensional access methods in the context of parallel database architectures.

8 Appendix

8.1 Computing Hilbert values

We present the algorithm xy_to_h to derive the Hilbert value ,h,of a point (x, y) and the function h_to_xy for the inverse Hilbert transform. The algorithms are table driven and the required tables are listed also. The variables x, y and h should each be interpretted as a binary fraction in $(0, 1)$ with j being the index of the jth bit to the right of the binary point. For a degree l Hilbert curve, x and y are l bits

state ind	*St*	T_1	T_2
0	1	0	0
1	0	0	1
2	2	1	1
3	0	1	0
4	0	0	0
5	3	1	1
6	1	0	1
7	1	1	0
8	2	1	0
9	2	0	1
10	0	1	1
11	3	0	0
12	3	1	0
13	1	1	1
14	3	0	1
15	2	0	0

Table 3: State Transition Table used by *xy_to_h* and *h_to_xy*

long and h is $2l$ bits long.

```
xy_to_h(x, y, l) returns (h) {
  h = 0, s = 0
  for j = 1 to l
  ind = 4s + 2x[j] + y[j]
   h += (2T1(ind) + T2(ind)) 2^-2j
   s = St(ind)
  end for
}
```

```
h_to_xy(h, l) return (x,y) {
  x = 0, y = 0, s = 0
 for j = 1 to l
    ind = 4s + 2h[2j-1] + h[2j]
    x += T1(ind)2^-j
    y += T2(ind)2^-j
   s = St(ind)
 end for
}
```

8.2 Algorithm ComputeRanges

The algorithm below assumes that (x_l, y_l, x_h, y_h) specifies a query, $H(x, y)$ is a function to compute the Hilbert value of (x,y), and *start*, *end* are lists of Hilbert values. In addition $H(x_l, y_l) = 4^l h(x_l, y_l)$ so that $H(x_l, y_l)$ are integers in $(0, 4^l - 1)$.

```
ComputeRanges(x_l, y_l, x_h, y_h, l) {
 for (j = y_l; j ≤ y_h; j++) {
    diff = H(x_l,j) - H(x_l-1,j)
    if (diff == 1)
        /* Start a new range */
        add H(x_l,j) to start
    if (diff == -1)
        /* end the current range */
        add H(x_l,j) to end
    diff = H(x_h,j) - H(x_h+1,j)
    if (diff == 1)
       /* Start a new range */
       add H(x_h,j) to start
    if (diff == -1)
    /* end the current range */
    add H(x_h,j) to end
   } /* end for */
 sort start, end. Then (start(i), end(i)) is
 a range of Hilbert numbers of blocks to be searched for
 0 ≤ i ≤ #intervals_found
}
```

A similar function can record the behavior of the curve along the y dimension (derived from ComputeRanges() by iterating through the x values substituting the calls to H() with the suitable values).

8.3 Spatial Joins

We provide an algorithmic description of spatial joins using Filter Trees and we present an example of its operation.

Spatial join between two filter trees is an index sweeping process. However, because of the structure of the filter tree the sweeping process is very efficient. For any pair of data sets, their join can be computed with the minimal IO effort, namely by reading each file only once. Data at each level of a filter tree are organized according to Hilbert order. The sweeping algorithm makes use of information derived from the Hilbert order of each level in order to achieve minimal IO effort. Assuming that we wish to join two filter trees F_1 and F_2 the algorithm makes use of the following information:

- $mn(F_i, L)$: the Hilbert number of the first entry in the current page at level L in the tree F_i, i = 1,2.
- $mx(F_i, L)$: the Hilbert number of the last entry in the current page at level L in tree F_i, i=1,2.
- $pr(F_i, L)$: the Hilbert number of the last entry in the current page at level L in tree F_i to be processed against all object approximations at level equal or higher levels in the opposite tree.

Every time we start processing a new page, both mx and mn are updated. For ease of presentation we assume the existence of the following functions:

- GetPr($page$) : returns the current value of pr for $page$.
- GetLevel($page$) : returns the level of the tree that $page$ belongs.
- GetMx($page$) : returns the current value of mx for $page$.
- GetMinimumPr(F_i) : returns the page that has the minimum value of pr over all pages of tree F_i.

Based on the above the algorithm follows:

```
while (more pages in either file exist)
 page1 = GetMinimumPr(F_1);
 PairUp(page1, F_2,);
 page2 = GetMinimumPr(F_2);
 PairUp(page2, F_1);
end while
```

Each time the page that has the minimum value of pr in each tree is determined. This page is processed against all the pages of the other tree at levels less than or equal the page's level.

```
PairUp(page, F) {
  process all the entries in page with Hilbert values between
  GetPr(page) and min(mx(F, j)) ∀j, 1 ≤ j ≤ GetLevel(page)
  against all entries of pages in F, within the same range, at levels GetLevel(page)
  and lower, currently in memory.
  update the pr value of page to min(mx(F, j)) ∀j, 1 ≤ j ≤ GetLevel(page).
  if (GetPr(page) == GetMx(page)) then
   discard page and read the next one. Upon reading update mx, mn, pr for that page
  end if
}
```

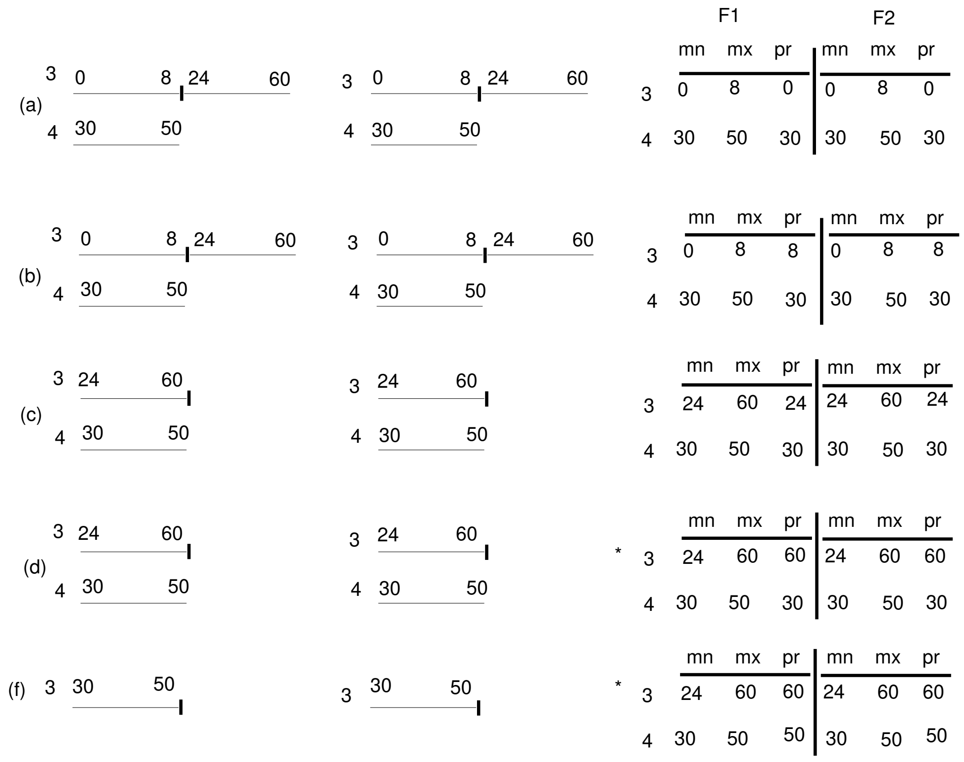


Figure 10: *Filter tree join algorithm example*

Processing a range of entities within a page, located at level l of tree F_1, against all pages of tree F_2 at levels l or smaller, located in main memory consists of (a) identifying relevant entities in the pages of F_2 and considering only those entities, so that we do not perform more work than is necessary (b) sorting the relevant entities and testing their intersection. For the intersection test, we can use fast computational geometry techniques, like plane sweep, to perform the intersections and report pairs of overlapping MBRs with the minimal effort. In order to identify relevant entities between the different levels of the filter tree we make use of the fact that data within a level of the filter tree are completely linearized on Hilbert order. Consequently, identification of relevant entries reduces to a range matching problem in one dimension. Entities at a specific level are organized according to the Hilbert values of that level. Consider a range of Hilbert values, $h_1 \ldots h_n$ generated at the first step of PairUp() above at a level l. This range has to be matched against all levels including l and lower of the matching tree.

We demonstrate the execution of the spatial join algorithm through an example. In figure 10, we present a filter tree with 2 levels and the ranges of Hilbert values that exist within each level. For simplicity and to keep the example short we consider the self join case. In addition, we present the Hilbert value fractions of each level using decimal values in order to make the example more readable. Figure 10a presents the initial ranges in the trees. The minimum value of pr for F_1 (0) is located at level 3. This value is the start of the first range. The end of it is computed from minimum value of mx

in F_2 at levels 3 and lower (in this case only level 3 exists and no lower level). This value is 8 and the first range is constructed, namely from 0 to 8. All the entities within the range from F_1 are processed with all entities within the same range from F_2. At the end, the algorithm updates the value of pr in F_1 to 8 and performs the same processing between F_2 and F_1 (the second call to PairUp()). Although for self joins this call is not necessary, in general for not self joins it should be performed. At the end of this call, the algorithm detects that pr equals mx at level 3 (10b) in both trees and the next page in been brought in, updating the values of mn, mx and pr for that page (10c). The minimum value of pr is located at level 3 again. The next range is formed from 24 to 60. This range is matched against level 3 of F_2 and vice-versa updating the values for pr. Although the value of pr equal mx for level 3 in figure 10d, this is the last page of F_1 at level 3 (as well as F_2) so is marked as such and stays in core. The minimum value of pr is located at level 4 and the next range is formed form 30 to 50. This range is matched against levels 4 and 3 of F_2 and vice versa. The value of pr is updated at level 4 in both trees to 50. At this point pr equals mx at level 4 also and since this is the last page of both trees at level 4 and all pages at all levels are visited, the algorithm terminates.

References

[AS83] D. J. Abel and J. L. Smith. A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem. *Computer Vision, Graphics, and Image Processing 24*, pages 1–13, March 1983.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins using R-trees. *Proceedings of ACM SIGMOD*, pages 237–246, May 1993.

[BKSS90] N. Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R* - tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, pages 220–231, June 1990.

[Bur91] Bureau of the Census. TIGER/Line Census Files. March 1991.

[Gar82] I. Gargantini. An Effective Way to Represent Quadtrees. *Comm. of ACM (CACM)*, 25(12):905–910, December 1982.

[Gut84] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

[HN83] K. Hinrichs and J. Nievergelt. The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects. *Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*, August 1983.

[HS95] Eric G. Hoel and H. Samet. Benchmarking Spatial Join Operations with Spatial Output. *Proceedings of VLDB*, pages 1–12, September 1995.

[Jag90] H. V Jagadish. Linear Clustering of Objects with Multiple Attributes. *Proceedings of ACM SIGMOD*, pages 332–342, May 1990.

[Ked82] G. Kedem. The quad-CIF tree: A Data Structure for Hierarchical On-line Algorithms. *Proceedings of the Nineteenth Design Automation Conference*, pages 352–357, June 1982.

[KF93] Ibrahim Kamel and Christos Faloutsos. On Packing R-Trees. *Second Int. Conf. on Information and Knowledge Management (CIKM)*, November 1993.

[KF94] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. *Proceedings of VLDB*, September 1994.

[Mag95] Geovane Magalhaes. Personal Communication. September 1995.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *Proceedings of VLDB*, pages 144–155, September 1994.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM TODS 1984*, pages 38–71, May 1984.

[NS86] Randal Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. *Proceedings of ACM SIGGRAPH 1986, Vol. 20, No 4*, pages 197–206, August 1986.

[Ore86] J. Orenstein. Spatial Query Processing in an Object-Oriented Database System. *Procedings of ACM SIGMOD*, pages 326–336, May 1986.

[RKV95] Nick Roussopoulos, Stephen Kelly, and Frederic Vincent. Nearest Neighbor Queries. *Proceedings of ACM SIGMOD*, pages 71–79, May 1995.

[Rob81] J.T. Robinson. The K-D-B-Tree: a Search Structure for Large Multidimensional Dynamic Indexes. *Proceedings ACM SIGMOD*, pages 10–18, 1981.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley publishing Co., June 1990.

[SLM93] Bernhard Seeger, Per-Ake Larson, and Ron McFayden. Reading a Set of Disk Pages. *Proceeding of VLDB*, pages 592–603, September 1993.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+ -tree : A Dynamic Index for Multi-dimensional Data. *Proceedings of VLDB 1987*, pages 507–518, September 1987.

[Sto93] Michael Stonebraker. The Miro DBMS. *Proceedings of ACM SIGMOD*, page 439, May 1993.

[Tam81] Markku Tamminen. The EXCELL Method for Efficient Geometric Access to Data. *Acta Polytechnica Scandinavica, No. 34*, March 1981.

[Wei89] Gerhard Weikum. Set-Oriented Disk Access to Large Complex Objects. *International Conference on Data Engineering*, pages 426–433, March 1989.

Contents