An Implementation and Performance Analysis of Spatial Data Access Methods

Diane Greene[1]

Computer Sciences Department
University of California, Berkeley
Berkeley, Ca.

ABSTRACT

Four spatial data access methods, R-trees, K-D-B-trees, R+ trees, and 2D-Isam, have been implemented within a preliminary version of the POSTGRES DBMS. These access methods have been tested over a range of shapes and sizes of two dimensional objects and for a range of logical page sizes. The cpu time, number of disk reads and writes and the resulting tree sizes have been tabulated for insert and retrieve operations. We conclude from this measurement study that R-trees provide the best tradeoff between performance and implementation complexity and that choice of implementation is crucial to the performance of all the methods investigated.

1. Introduction

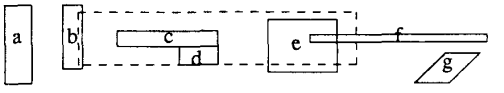
Spatial data access methods provide a means of retrieving objects which are n dimensional points or solids. In particular, they optimize spatial queries which retrieve all points or solids which are enclosed in or overlap a specific search region.

Recently there have been a substantial number of access methods proposed to manage such point or solid data. These include [BANE86, GUTT84, HINR84, NIEV84, OREN86, ROBI81, and SELL87]. However, performance studies of these access methods have often been analytical rather than based on real world implementations (e.g. [SELL87, NIEV84]). Also the non-analytic performance studies construct a hollow shell of a data base system thus leaving out such functions as primary index or heap accesses, and buffer and lock management [ROBI81 , GUTT84]. This paper reports on implementation techniques and experimental results obtained from implementing and testing several of the popular access methods over a range of solid spatial data sets. This study was performed by implementing the access methods directly in the prereleased public domain version 1.0 of POSTGRES [STONE87]. Thus, the experimental results reflect total data management costs.

The second section of the paper describes the four access methods that were tested and their specific implementations, focusing on the insert, split and retrieve algorithms, and efficiency considerations. Section 3 lists the parameters for the tests, how the rectangular data sets were varied, what operations were performed and how the measurements were instrumented. Section 4 presents and discusses the results and Section 5 is the summary with suggestions for further work.

2. The Access Methods

The access methods which we consider optimize queries that retrieve all the data objects overlapping a user specified rectangular manhattan window as noted in Figure 1. Here objects b,c,d, e, and f overlap the search region. Boxes a through f are legal and box g, a non-manhattan rectangle, is not.



Definition of overlap and allowable geometry.

Figure 1

In this study we restricted boxes to two dimensional manhattan rectangles, specified by 4 floating point numbers, xmin, xmax, ymin and ymax. The restriction to two dimensions is simply for convenience; all results should generalize to the n dimensional case.

Each access method builds a secondary index on the spatial data, and the original records are stored in the DBMS heap. Each of the methods can handle dynamic updates to the index, although 2D-Isam is restricted in that overflow occurs after a fixed percentage of updates.

The insert and retrieve operations were implemented for each access method. The delete operation was not implemented.

We chose a representative collection of access methods to study, namely R-trees [GUTT84], K-D-B-trees [ROBI81], R+ trees [SELL87], and a hybrid extension of ISAM called 2D-Isam. The access methods which we studied are described in the next four subsections.

2.1. R-Trees

The structure was implemented as originally proposed in [GUTM84] and is shown graphically in Figure 2. It is a multi-level tree structure designed to handle n-dimensional objects. The tree is balanced and space utilization will always be at least 50%. All non-leaf nodes are assigned a minimum and maximum number of entries and each entry contains an identifier which locates its child node and also the spatial region which the child covers.

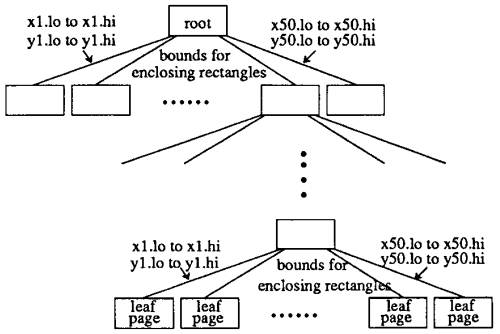
Each intermediary non-leaf node has a rectangular region associated with it which encloses all rectangles in any of its descendant nodes. Intermediary nodes on a given level can overlap; hence their rectangles do not represent disjoint regions.

The amount of overlap between sibling nodes is directly proportional to the number of paths which much be traversed for a given spatial query. In the extreme case, if every enclosing rectangle covered the entire search space then every leaf would have to be inspected on every search query. Unless rearrangement of the index structure is performed, or the data happens to be inserted in an optimal order, it is difficult to generate a tree that will have small overlaps between the siblings at the upper levels.

When a leaf page fills it is split and if this causes the parent page to overflow then the split is propagated upwards. The splitting algorithm is crucial to how much overlap exists in the resulting rectangles.

1 Currently at Sybase in Emeryville, California

† Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089 and ESL, Incorporated. Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089 and ESL, Incorporated.



R-Tree with Max Children = 50

Figure 2

Data Structure for R Trees

An R-tree is made up of nodes which are treated as logical pages, we refer to them as node pages. There are four possible types of nodes; a root node, a root that is also a leaf, a leaf node, and a node that is neither a root or a leaf. The structure of a node page includes the number of entries, i.e. the number of child nodes that it points to, a type flag, the parent page node identifier, the index into the parent page's array of entries, and an array of child page entries. Each child entry contains its enclosing rectangle and node identifier. The parent identifier is necessary for propagating splits upwards. The parent index is useful for implementing non-recursive retrieves and also for debugging.

The number of bytes in a logical node page is determined by the data type of the indexed attributes and by the maximum number of entries allowed per node. Page size is an optimization parameter for spatial data access methods. A large page size means that the tree will be shallow, a small page size means that sorting and searching individual nodes, will go quickly but the tree will be deep.

Insertion

It is important to pack the nodes densely and this can be done by finding the best leaf to insert a new data object into. The goal is to minimize the amount of overlap between siblings. The algorithm we used is as follows:

- Let the root be the current node.
- FindBestLeaf (current_node)
  - If the current node is a leaf:
    - Insert (the_new_object_data).
  - Pick the child of the current_node whose enclosing rectangle area will be increased the least by the addition of the new object data.
  - Expand the enclosing box of that child to include the new object data.
  - Get the new child and make it the next current node.
  - FindBestLeaf (current_node)
- Insert(new_object_data)
  - If insert causes overflow:
    - Split leaf node.
    - Add it to parent.
    - If this causes overflow.
      Split parent.

Splitting

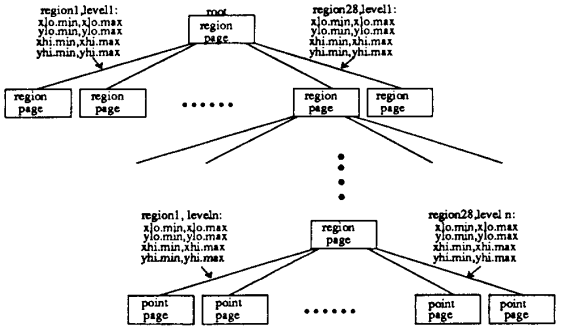
The split algorithm finds the two most distant rectangles in the current node, we call these seeds. This takes $O(n^2)$ time. The split is then made along the axis perpendicular to the axis which has the greatest separation of the two seeds. The separation numbers are normalized by the length of the node's enclosing box along the appropriate axis. Once the split axis and coordinate are chosen, the entries are sorted by low value along that axis. Entries usually remain sorted and so by using quicksort, this generally takes only linear time. The low side node gets the first (MAXENTRIES/2 - 1) entries, the next entry goes in the side whose enclosing box will be increased the least by its addition, and the remaining entries go in the high side.

Retrieval

Normally this would be a simple recursive algorithm that traversed the tree until it had retrieved from all leaves that had overlapping data. However, retrieving all qualifying data objects at once can overflow the DBMS buffers. Our algorithm is non-recursive, a marker is used to indicate where the last retrieved data object was found and the qualifying records are passed to the DBMS one at a time.

2.2. K-D-B Trees

K-D-B trees are another multi-level balanced tree structure designed to handle n-dimensional point data. They are designed for point data and hence, a two dimensional rectangle must be mapped into a four dimensional point; xmin, ymin, xmax, ymax. The nodes at a given level in the tree are partitioned into disjoint subregions that when summed, span the entire search space. That is, each dimension is a domain which spans -inf to +inf along the axis in which the points of that domain lie. Figure 3 illustrates the structure of a K-D-B tree.



K-D-B Tree with Max Children = 28

Figure 3

Splits divide a node's search space along one of the n-dimensions or domains. The region is divided into two by picking an element in that domain and placing all points less than the domain element in the left node, and all points equal to or greater, in the right node. Regions cannot overlap and hence, if the chosen split intersects any of its child nodes then the split must propagate downwards. This can cause the creation of empty nodes and result in poor space utilization. An empty node will be created if the split intersects the child's region and if the points of the child happen to all be clustered on one side of that region. The elements are either all less than or all greater

than or equal to the split coordinate. Clustered regions in a given domain can result from splits that were generated in and gave a good distribution along a different domain.

It is difficult to choose the "best" domain to split on and as suggested in [ROBI81], we implemented a cyclic splitting strategy. In other words, the domain to split on was rotated among xmin, ymin, xmax and ymax. The actual value of the split was chosen to balance the number of points between the two new pages. This strategy caused many downward propagating splits which in turn caused numerous empty and near empty nodes to be created. The result was extremely poor performance. Matters could have been improved in two ways. We could have implemented a split strategy that minimized downward splits. This could be done by finding a split domain that had a point which was either < or >= to all of its children's regions for that domain. Anywhere that the coordinate of a split in a given domain intersects the child's range for that domain, a downward split is generated. Another improvement would have been to develop a way to reorganize the structure once the space utilization fell below a set minimum value. This would be done by coalescing nodes. Merging nodes at any level other than at the leaves is extremely difficult in a K-D-B tree. It generally requires rearranging more than just two nodes since just two regions may not be joinable. Some of the ranges of their domains might be incompatible. [ROBI81] discusses this problem and offers no solution except to restrict joins to the leaf level. Both of the above discussed improvements would have required a significant coding effort which we felt to be unjustified.

K-D-B trees are not particularly suited for solid object data. The mapping to points makes queries more difficult and more computationally expensive. An overlap query for an access method that uses enclosing boxes to define its regions could find the x overlap of the window defined by XLOW,XHIGH, YLOW,YHIGH as follows:

if((xhigh >= XLOW && xlow <= XHIGH)

The equivalent query for the K-D-B tree would be:

if((x_{low}high >= XLOW && x_{low}low <= XHIGH) &&

(x_{high}high >= XLOW && x_{high}low <= XHIGH) &&

where x_{low}high refers to the high range of the low value, and x_{high}high refers to the high range of the high value. K-D-B trees do not make use of the fact that a solid object's region of overlap continuously covers the space between its low and high values. The relationship between the low and high values is not integrated into the data structure. Each value is in a different dimension and treated as a separate domain and so, operators such as overlap and operations such as find_the_optimal_split, are more complex to write.

2.3. R+ Trees

R+ trees combine the features of R-trees and those of K-D-B trees. Like R-trees, the structure uses enclosing rectangles to partition the search space at each successive tree level. Unlike R-trees, enclosing rectangles can not overlap sibling enclosing rectangles. As in K-D-B trees, the search space is divided into disjoint sub-regions. K-D-B trees divide the regions into ranges of point domains, R+ trees divide the space into solid objects which we call enclosing rectangles. The union of all the enclosing rectangles at any given level of the tree will completely cover all of the data objects which are identified at the leaf level. That is, the disjoint regions sum to the manhattan cover of all the data objects and data is placed in each leaf node that has an overlapping enclosing rectangle. A data object that overlaps the entire search region is pointed to by each and every leaf node and thus would have number_of_leaves minus one, duplicate entries.

Dividing the search region into disjoint subspaces means that retrievals will be efficient; there will always be only one path to the data for a given area of overlap. In R-trees, several paths might cover the same overlap region. There is a tradeoff; because the leaves of an R+ tree contain duplicate entries, more space is required and thus there can be more levels in the search path than that of the equivalent R-tree. This phenomena is illustrated in Figure 4 where the R+ tree has 3 levels and the R-tree has 2. This typically happens when there is large overlap between the data objects themselves.

Figures 4 and 5 show the effect of large and small overlaps on the two structures. The examples are for 20 box data sets and the MAXENTRIES per node is set at 6.

The data set of large overlapping boxes shown in figure 4 cause R+ trees to have duplicates, hence the extra level of tree. The R- tree shows quite a bit of overlap between the root enclosing boxes but this is offset by the more even distribution of data amongst the nodes. The R+ tree has several nodes that have only one or two entries.

In Figure 5, the data set boxes are small and there is minimal overlap. R+ trees tend to partition the search area into nearly equal enclosing boxes while the R-tree shows a large disparity between the areas of the resulting enclosing boxes. Since the input data was randomly generated and thus has a nearly even distribution, the R+ tree is bound to perform better for this sort of data. In addition, the R+ tree splits tend to occur along the same axis and this will pave the way for clean, non-downwards propagating splits as the tree grows.

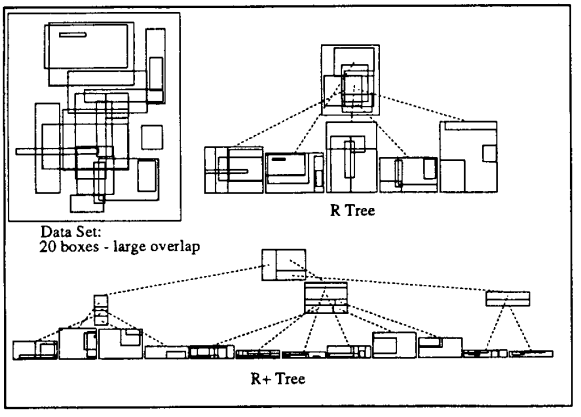


Figure 4

† In order to evaluate the effectiveness of various insertion and split algorithms, a user interface was created which graphically tracks the access method tree nodes accessed and modified during an insert or retrieve query. The root node has the enclosing boxes for the children drawn inside of it; the bottom level leaf nodes are the actual data objects. The box in the upper left corner contains the entire data set which was inserted.

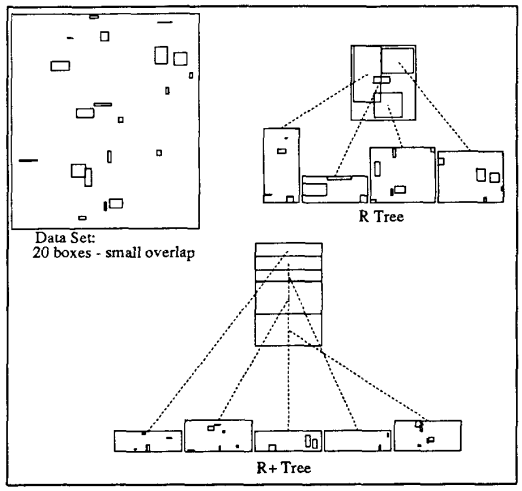


Figure 5

As in R-trees, when a leaf node fills, it is split. If adding the new leaf to the parent node causes an overflow, then the split propagates up. The parent node may not have a way to split cleanly or the only clean split may cause a skewed distribution of the child entries in the resulting two new nodes. A clean split is one which lies only on the boundaries of a node's child enclosing boxes. For example, with 6 entries per node, the clean split might occur such that 6 entries ended up on one side and only one on the other. Figure 6 illustrates an R+ tree split that caused such an imbalance.

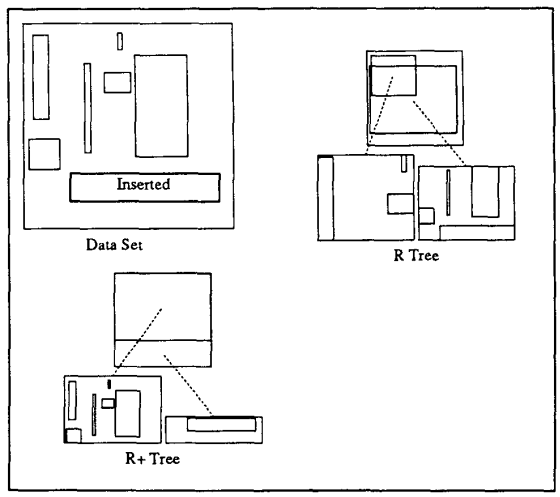


Figure 6

If it is not possible to find a "good" clean split, (i.e. balanced), then it is necessary that the split be propagated back downwards through any of the child entries that were intersected by the choice of split axis and coordinate. Figure 7 illustrates how this might happen.

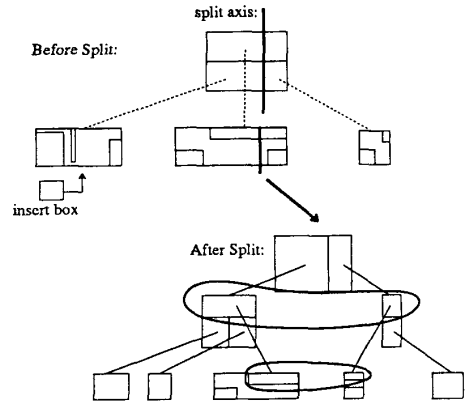
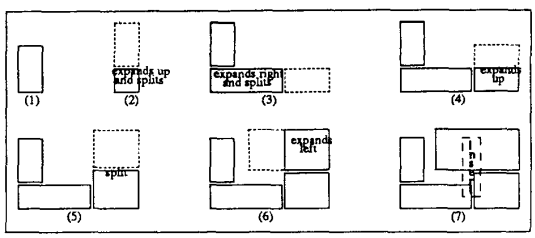


Illustration of Split Which Gets Propagated Up and Down
Nodes that were split on the downward propagation, are circled.

Figure 7

When splits can propagate down, there is no way to guarantee a minimum number of entries per node. This is because the split axis and coordinate is chosen at a higher level. It may so happen that the lower level node's enclosing boxes all fall on one side of the split.

Enclosing boxes will contain empty spaces because regions are initially divided so as to encompass the entire search plane. The first split at the root node will result in two nodes that cover the entire xy plane. For example, if the split is vertical at x = 5.0 then the left node's enclosing box will have the coordinates: x_{low} = -inf, x_{high} = 5.0, y_{low} = -inf, y_{high} = +inf; the right node's enclosing box will be: x_{low} = 5.0, x_{high} = +inf, y_{low} = -inf, y_{low} = +inf. If the region were partitioned to manhattan cover exactly the current set of data objects, empty spaces between the enclosing boxes would result. It could become difficult to fill these spaces, an expansion in one direction might not be sufficient. Figure 8 illustrates this problem. It is computationally hard to find the best way to expand a set of enclosing boxes to cover a space such as the one in the middle of frame (7) of Figure 8.



Step (7) illustrates a situation where an insert box cannot be covered by simple expansion of Steps (1) through (6) illustrate a possible sequence leading up to (7).

Figure 8

When the data becomes extremely dense it is possible that R+ trees will not work. If all of the entries in a full leaf node overlap,

then any split will require that each of the entries be split as well. This means that both resulting pages will be full and there will be no place for the entry that is to be added. In fact, for any given choice of split axis and coordinate, there are criteria that can be used to determine if the proposed split will be possible; i.e. the split will not result in overflow any of the new nodes, the number of entries overlapping the split axis plus the number of entries falling to one side of it, must be less than or equal to the maximum allowable number of entries per node, (MAXENTRIES). For example, take MAXENTRIES = 10, a full node and an 11th entry ready to be inserted. If there are 3 entries lying to the left of the split axis and 8 entries overlapping then that split is not feasible. The split would result in 11 entries being inserted on the left side which is more than MAXENTRIES allows. To program around this limitation requires allowing chaining of overflow nodes at the leaf level.

Data Structure

The R+ tree node pages are identical to those of R-trees except that each child entry at the leaf level also includes a flag to handle the problem of duplicate entries. A data object is inserted into every leaf it overlaps, the flag is used to indicate if that object is the first occurrence in the tree. It is assumed that the index will always be traversed from left to right so that first occurrence can be defined as the leftmost instance. If an overlap is found on an object that is not the first, it is ignored since retrieving it would cause duplicate retrievals.

Inserting.

A box tuple is placed in all of the leaves that have overlapping enclosing boxes. This means that starting at the root, all nodes having overlapping enclosing rectangles must be traversed down to the leaf level. Any single insert to a leaf node can conceivably cause a split which propagates all the way up to the root and leaves the tree index completely rearranged. This presents a problem in that it is difficult to tell which inserts have been made and which have yet to be executed. In order to avoid duplicate inserts and insure that every leaf with an overlapping enclosing box gets the insert, the insertion is implemented nonrecursively. Flags are used to indicate which nodes have already been searched and/or inserted into. These flags cannot be written into the data structure since they are only meaningful during the set of insertion commands that set them.

Splitting.

The idea is twofold, minimize the number of splits that propagate downwards and make the resulting two nodes as balanced as possible. The first consideration is given priority. The algorithm is as follows:

- In the node to be split, find the two most distant rectangle entries.
- Sort all rectangle entries on the low value of the axis by which they are most distant.
- For each entry i:
  - Count the number of overlapping entries, (num_overlapping).
  - Count the number of entries whose high value is lower than the given entry's low value, (num_less_than).
- Make sure a possible split exists, that is, for at least one entry:

(num_overlapping + num_less_than <= MAXENTRIES) && (num_less_than > 0)

- If no split exists, try the other axis.
- If still no split exists, R+ trees, (without chaining), fail.
- For the axis chosen, find the best split:
- For all the entries, find the minimum value of:

MAXENTRIES/2 - num_less_than + MAXENTRIES*num_overlapping

The multiplicative factor of MAXENTRIES is used to penalize splits which intersect child nodes since they cause downward split propagation. We experimented with a range of multiplicative factor values, starting with 1. No significant performance differences were apparent. Values greater than MAXENTRIES/10 worked well.

After a split, all of the entries in the two resulting nodes will be sorted by the low value perpendicular to the split axis. If a subsequent split occurs on the same axis then sorting using quicksort will take only linear time.

Split Propagation Down

A split that propagates downward can be done recursively. It must be done such that changes to nodes occur in a backwards order, that is, beginning at the bottom level. This means that at each recursive call, new left and right nodes are created and then passed down until the leaf level is reached. As the recursive calls returned, the left and right nodes can then be updated with the new child nodes which resulted from the downward propagate call.

- Propagate_Down(Pass in left and right parent, entry to be split, and split axis)
- Make new "left" and "right" nodes.
- Use current "to be split" node to look at each child entry and:
- If child lies completely to the left or right of the split put it in the left or right node respectively.
- Otherwise
  Call Propagate_Down, passing the left and right nodes as the parents.
- Update the two parent nodes by adding the new left and right nodes and removing the old "to be split" node that they originated from.

Retrievals

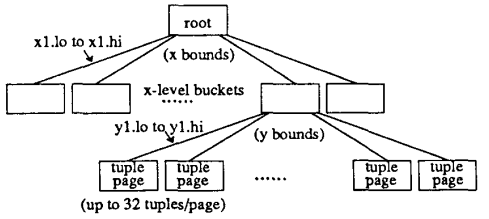
The implementation is the same as that for R-trees except that at the leaf level, the occurrence flag is checked and the tuple is only returned if it is the first/leftmost instance.

2.4. 2D-Isam

2D-Isam is a two-level tree structure similar to grid files [NIEV84]. The first level partitions the data along the x axis and the second level partitions along the y axis. Figure 9 shows the structure of an index built to handle 10,000 objects.

In order to build the index, the entire data set is first sorted on xmin and then partitioned to be distributed evenly among a collection Nx, of first level buckets. The root of the index points to each of these buckets and includes with each pointer the minimum and maximum x values of any object in the bucket.

Each of these Nx buckets can be thought of as an enclosing rectangle for the data beneath it. The rectangle extends the full length of the y dimension and has the x bounds discussed above. Notice that these enclosing rectangles overlap in the x dimension.



2D-ISAM Structure for 10,000 Tuples

Figure 9

Each x-level bucket is sorted on ymin and partitioned along the y axis into Ny buckets. These partitions make up the data pages at the bottom level. Each x-level bucket has the pointers to all associated Ny data pages along with the corresponding xmax and ymin. Again notice that y buckets can overlap.

2D-Isam has the property that 2 levels of traversal reach a data page packed with objects which are enclosed by a known set of x and y axis bounds. The structure includes provisions for overflow once the data pages fill up, however, the performance will degrade considerably once overflow occurs. Ideally, the structure should be reorganized once the buckets fill up.

The size and number of buckets are chosen at the time the structure is first created. The page and tuple size dictate what the number of tuples per page is:

page_size / tuple_size = tuples per data page.

This number of tuples per page is multiplied by a fill factor (*ff*) which was 0.75 in our implementation. Consequently, buckets were 75% full when the initial index was completed. The total number of data pages required is based on the total number of tuples and the calculated number of tuples per page:

total number of buckets required = total_tuples/ff(tuples_per_pag*

Since there are two levels, each non data page node contains a number of points equal to the *square root of the total number of buckets.*

In our implementation page size was 1024 bytes and data tuples contained 4 eight byte floating point numbers. Hence there can be as many as 32 tuples per data page. Each index page contains a collection of records of length 36. Hence the fanout is at most 28. In 2D-Isam $28^2 * 32 = 25098$ tuples will fill the structure. Larger data sets require a bigger page size or a deeper tree.

2.5. Integrating into the DBMS

The access methods were run in version 1.0 of the POSTGRES DBMS [STONE87]. They were implemented as "fast path" procedures, a special feature of POSTGRES that is described below.

In POSTGRES, a collection of POSTGRES commands can be defined as a procedure and stored in a relation. The procedure can then execute with any user supplied parameters and thus, the boundary between POSTGRES and an application is crossed just once rather than once per command. POSTGRES also allows user defined functions to appear in queries. For example, a user can write a function OVERPAID which takes an integer argument and returns a Boolean. After registration with the data manager, a user can write a query such as:

retrieve (result = OVERPAID (7))

which will simply evaluate OVERPAID for a constant parameter. In addition, the POSTGRES query language permits

function-name(parameter-list)

as a legal query and so, OVERPAID(7) could be submitted as a POSTGRES query. The run-time system accepts such commands from applications and simply passes the arguments to the code which evaluates the function. This avoids the overhead of type checking, parsing and query optimization, and results in a low overhead interface.

Finally, POSTGRES has user defined access methods. The abstraction for the access methods is a collection of 13 functions, such as am_build, am_insert, am_retrieve, and so forth. These functions can be directly called by a user written procedure. As a result, a the procedure can be coded directly against the access method level of POSTGRES. This permits short circuiting many of the data base services that would otherwise cause considerable overhead.

The use of "fast path" for these tests, means that the parser, planner and executor were short circuited. Otherwise, the full DBMS was used and so, in addition to the access method secondary index insertion and retrieval times, the tests include the time to insert into and retrieve from the heap where the relation's records are actually stored. The times also include calls to buffer management and the use of read and write locks.

Buffer size for POSTGRES was set to the Sun OS default page size of 8K. We started out with the number of buffers set to 8 but had to increase this to 16 for some of the more degenerate R+ tree cases. All of the tests tended to be I/O bound and more buffer space would have alleviated the bottleneck somewhat. A query runs much faster if the logical pages it accesses can remain in the buffer pool until query completion. In some cases this could imply the need for the entire index to remain in the buffer pool and this is not realistic. The access methods were implemented with only the logical node pages which lay along the path of a recursive split propagation pinned. Once an insert at a given leaf was completed, the node pages used thus far were released. Note that R+ trees can make multiple inserts into the structure for a single insert query thus requiring the entire index to be locked. Also, in R+ trees, since splits can propagate up and down the trees, a larger amount of buffer space is required than for R-trees.

3. The Parameters

3.1. The Data Sets

The access methods were tested on data sets of 10,000 rectangles. The rectangle data was generated by specifying the lengths of the sides and then randomly generating the origin, (xmin and ymin), with a uniform distribution over a 100 x 100 "world". The range over which the size and shape of the rectangles were varied, is described below.

First, the percentage of the 100 x 100 "world" which was covered by the 10,000 rectangles, was varied by changing their dimensions according to the following values:

10,000 small rectangles:.01 x.01 square => .01% of the world covered

10,000 medium rectangles:.1 x .1 square => 1% covered

10,000 large rectangles:1 x 1 square => 100% covered

10,000 large rectangles:10 x 1 square => 1000% covered

Secondly, the node page size was varied by changing the MAXENTRIES per node according to the following values:

25 nodes per page=> page size 614 bytes
50 nodes per page=> page size 1214 bytes
100 nodes per page => page size 2414 bytes
200 nodes per page => page size 4814 bytes
300 nodes per page => page size 7214 bytes

3.2. The Operations

The operations performed on the data sets were insert and retrieve. An insert query takes a single record, inserts it into the heap, and then inserts the heap tid, (an identifier which identifies the record's location), into the access method index. The record's box field is used as the indexing data. Records were inserted one at a time and no presorting was performed.

Retrieve queries took a user specified rectangular window and used an overlap operator to fetch all records which had box attributes satisfying the condition that they either intersected or lay wholly inside the boundaries of the retrieve window.

The size of the retrieve windows was varied in order to see how the access methods were effected by the selectivity of the search region. The windows were set as follows:

1 x 1 window => .01% of a 10,000 unit square area
2.5 x 2.5 window => .1%
5 x 5 window => .5%
10 x 10 window => 1%

3.3. Measurements

Instrumentation

The CPU, System, and real time, and the disk I/O's, were measured using the information available through Unix system calls. In addition, the code was instrumented with counters in order to measure the number of buffer and page I/O's, the size of the index structures in terms of bytes and number of nodes, the number of splits occurring for a given set of inserts, and the number of duplicate leaf entries in an R+ tree. Lastly, the expected value of the number of nodes that would have to be traversed to find a point overlap in an R-tree was computed after each series of inserts. This latter measurement is an indication of the amount of overlap between sibling nodes.

The Machine

All tests were performed single user, on Sun Microsystems 8MB Sun 3/75 running release 3.4 of Sun Unix 4.2, with its own Micropolis 60MB 1325 SCSI disk. The 3/75 is approximately a 2 MIP machine.

4. Results and Discussion

We measured the performance for all four access methods and the interesting test results turned out to be those of the R-trees and R+ trees. The performance of K-D-B-trees, as implemented, was uninterestingly slow. As discussed in Section 2.2, it could have been improved by spending substantial time tuning up the splitting strategies and implementing a way to rearrange the nodes when too many empty ones appear. It did not seem worthwhile to spend time doing so since K-D-B trees are not likely to match the performance of the similar R+trees when the data is solid objects and not points. Both access methods share the problems associated with partitioning space into disjoint regions; splits can propagate both up and down the tree and it is not possible to guarantee a minimum space utilization.

The advantage of R+ trees is that they do not map the data into points but rather, maintain and make use of the properties of 2-D data objects. The mapping to points handicaps K-D-B trees in several ways. Twice as much storage is required to store an enclosing region than is needed for an enclosing box. The former requires 8 object data type coordinates, (in our case floats), and the latter requires 4. Overlap queries in a K-D-B tree require twice as much computation since instead of a single low to high range on each axis, there are two sets for each axis. Lastly, a straight mapping of rectangles into points will result in a nonuniform distribution of points and this is bad for performance. This occurs because there will never be any points below the diagonals since it is always true that $x_{low} <= x_{high}$ and the same for the y values.

After testing the static 2D-Isam structure, we noticed that with suitable preprocessing of the input data, R-trees could be made to duplicate their performance. 2D-Isam takes box input data, sorts it on the low x values, and distributes the data evenly into a set number of "x-level" buckets. The number of buckets is a function of the logical node page size, it is the number of buckets that the root node page can hold pointers to. Each "x-level" bucket is then sorted on low-y and and the data is distributed evenly into "y-level" buckets. The resulting structure is identical to an R-tree that has only nonoverlapping vertically split regions at the root level, and nonoverlapping horizontal splits at the second level. The vertical splits effectively divide the data into "x-level" buckets and the horizontal splits create the "y-level" buckets.

An R-tree with this data distribution and thus equivalent structure, can be created if the data is appropriately presorted and a special build algorithm is used to pack the nodes. The data is sorted along the low-x values and then divided into (n = fill-factor * MAXENTRIES) groups which are in turn sorted on the low-y values. This data is then inserted, in order, into the leaves with n entries per leaf. Finally, the appropriate enclosing boxes and intermediary nodes are be propagated up to the root. Such a build will result in a structure identical to 2D-Isam. It will have the significant advantage, however, that the data is now in a dynamic index and can accept updates without risking overflow and requiring reorganization.

The only advantage of 2D-Isam over R-trees is one of space considerations. 2D-Isam only needs to store bounds along a single axis at a given level. R-trees must store the enclosing box and thus twice as many bounds. This results in approximately a 50% increase, (814 bytes to 1214 bytes for a 50 entry node page), in data structure size over 2D-Isam. This size increase did not significantly change the performance and so the 2D-Isam results are not comparatively interesting.

The results for R-trees and R+ trees are presented in Table 1 and in Figures 10 through 15. Table 1 is a comparison of the tree structure properties for both access methods.

TREE PROPERTIES FOR VARYING BOX SIZES								
	Box Size in a 100 X 100 World							
	.01 X .01		0.1 X 0.1		1.0 X 1.0		10.0 X 1.0	
Property	Rtree	R+	Rtree	R+	Rtree	R+	Rtree	R+
# Nodes	349	302	339	298	324	610	318	451
# Splits	346	288	336	290	321	457	315	375
R+ Tree Duplicates	-	23	-	200	-	3010	-	1491
R+ Tree Down Splits	-	11	-	8	-	1470	-	355
Rtree,retr. E(root)	29.5	-	31.27	-	14.60	-	6.563	-

Table 1

Tabulated in table 1 are the tree structure properties for R-trees and R+ trees after 10,000 box insertions. The boxes range in size and include .1%, 1%, 100% and 1000% coverage of the region. The .1% through 100% coverage refers to square boxes. The 1000% coverage is for boxes with a 10 to 1 aspect ratio, i.e. boxes that are long and narrow along the x-axis.

The number of nodes and number of splits which resulted from 10,000 inserts, are listed in rows 1 and 2. The number of nodes in the R+ trees is always greater because of the duplicate entries.

When the boxes are virtually without overlap, R+ trees have 17% fewer splits. The opposite is true when the coverage is high and the boxes overlap more. This is largely due to the increase in the number of duplicates occurring at the R+ tree's leaf level. Notice that at 100% coverage 30% of the R+ tree entries are duplicated and the number of splits requiring downward propagation rises dramatically to 3 times the number of splits. Since the level of the tree is 3, this implies that, on the average, every split generates at least one downwards split propagation.

The last column is interesting because it shows that R+ trees improve if the data is skewed in one direction. With 100% coverage and square boxes, R-trees performed 30% fewer splits. Then, with 1000% coverage but high aspect ratios, this number is halved with R-trees performing 16% fewer splits. This can be explained because uniform data promotes clean splits and nonoverlapping entries at the leaf level. With the 10 by 1 boxes, only 15% of the entries were duplicated.

The last row is a measure of the amount of overlap between sibling nodes in the R-trees. It is the expected value of the number of nodes that will be visited in order to find all data overlapping a random point. This number was calculated recursively as follows:

for each node:

E(current_node) += E(child)*P(child)

E(leaf) = 1

P(child) = area of child / enclosing box of parent node

Notable is the fact that, as the amount of search space coverage goes up, the partitioning of the search space improves. This is indicated by the drop in the expected value figure. This must occur because insertion of densely populated region will mean densely packed nodes and thus smaller overlaps between siblings. The spacing between inserted boxes is small and so the enclosing boxes will not have much empty space. Also the splitting algorithm we used always picks the two most extreme seeds as the starting points for the two new nodes. This strategy will be effective in packing the data if it already tends to be dense since distant boxes will always end up in different enclosing regions eventually.

The graphed results reflect measurements made after inserting 10,000 boxes into the data base. Figures 10 and 15 graph the insert and retrieve results for R-trees and R+ trees over a range of box sizes. Figure 11 graphs the effect of page size on insert and retrieve performance for R-trees.

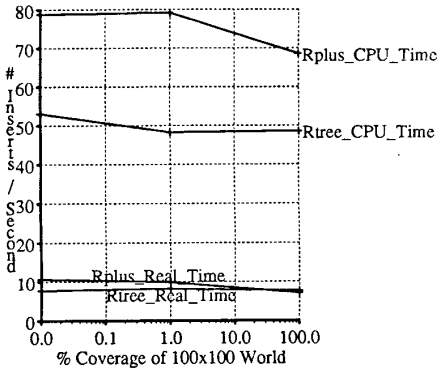


Figure 10

The y-axis of Figure 10 graphs the number of boxes inserted per cpu and real seconds for R-trees and R+ trees. The gap between CPU and real time reflects the fact that execution was I/O bound. Since the tests were run single user, performance can be judged by the number of inserts per real time second. By this metric R-trees and R+ trees do not show significantly performance differences. R+ trees are slightly faster at minimal % area coverage and at 100% coverage there is a crossover point where R-trees start to outperform the R+ trees.

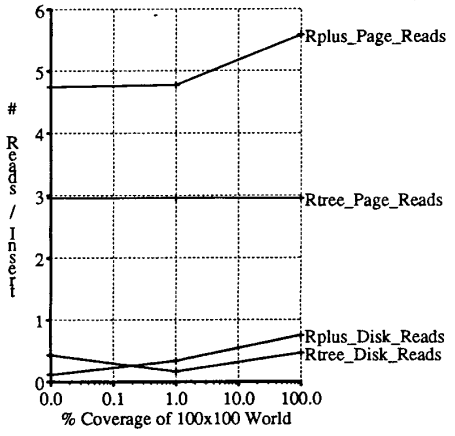


Figure 11

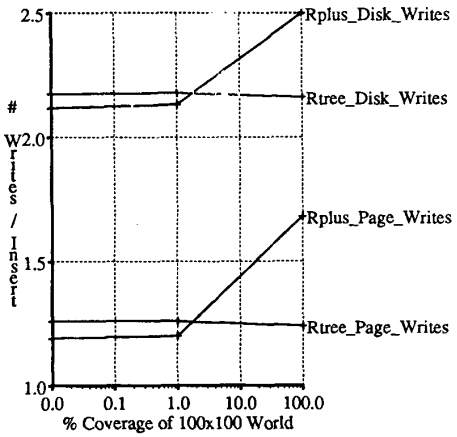


Figure 12

Figures 11 and 12 graph the number of logical and physical page reads and writes respectively. The y-axis indicates the number of reads or writes which occur per insert. R+ trees execute fewer I/O's at .1% coverage but at 100% coverage, R-trees achieve better performance.

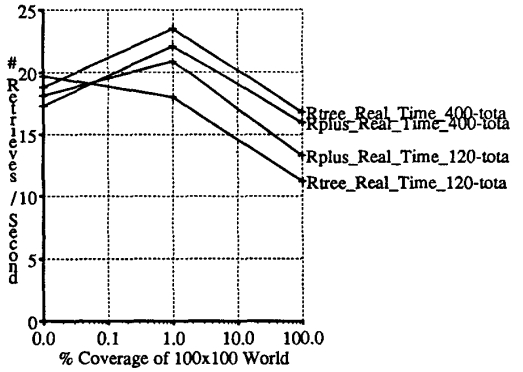


Figure 13

Figures 13, 14 and 15 graph retrieve results on a data base of 10,000 records. 10a shows the average number of retrieves per second and figures 14 and 15 show the number of reads and writes. Again, there is no decisive difference between the performance of the two access methods.

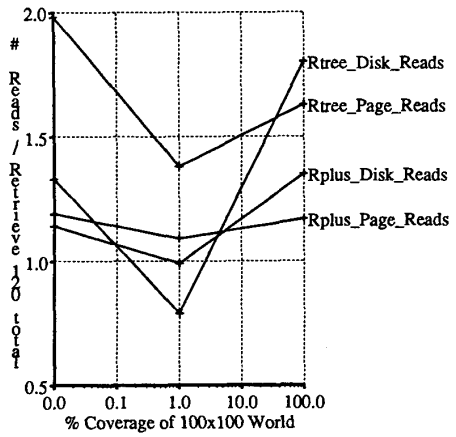


Figure 14

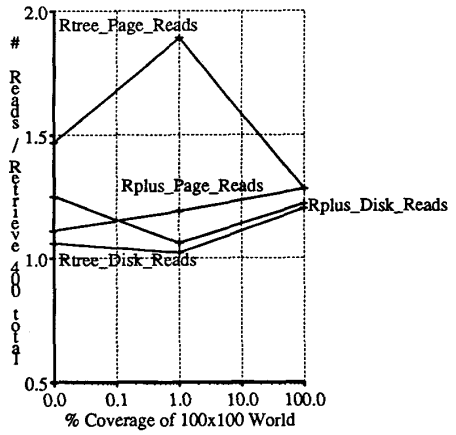


Figure 15

The size of the node page for the tests discussed thus far was arbitrarily picked to have MAXENTRIES set to 50. Figure 16 graphs the effect of varying page size from 25 to 300 maximum entries per node. The tests were run on R-trees, the two plots are for number of inserts and retrieves per real time second. Ten thousand insertions were made using the data set of .1 X .1 boxes.

Inserts have to examine all of the entries in a page during downward traversal of the tree to choose the best entry to add the new data to and also whenever a split occurs in order to redistribute the entries into the two new resulting nodes. If numerous splits occurs then a large node page size could be expensive, however, the number of splits decrease as the page size increases. Larger page size also means that fewer page accesses will occur and also that the tree will have fewer levels.

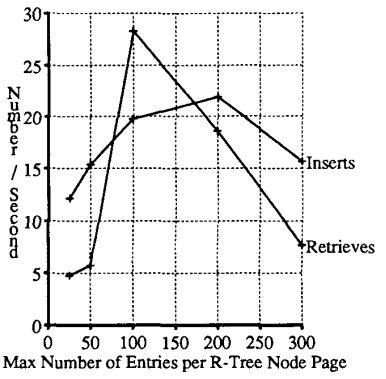


Figure 16

Retrieve queries spend most of their time checking the entries in a page for overlap. A smaller page size would tend to partition the search space better so that fewer total entries would probably end up being examined. On the other hand, as page size decreases, overlaps increase and hence more paths have to be traversed.

The results indicate that the optimal page size lies somewhere between 100 and 200 entries per node. The maximum for retrievals occurred at 100 entries per node and the peak for insertions, much less decisive, occurred at 200 entries per node. The difference between 100 and 200 entries per node for insertions probably has more to do with the fragmentation of the physical page than it has to do with properties of the access method itself.

5. Summary and Future Work

We have described the implementation and performance analysis of four different spatial data access methods. It was shown that 2D-Isam could be simulated by R-trees and hence the latter method is more powerful. 2D-Isam is a static index while R-trees are dynamic. R+ trees were shown to be an improvement over K-D-B trees since they do not require a mapping into points. Both K-D-B trees and R+ trees suffer from having to handle splitting of nodes up and down the tree. In addition to being difficult to code, this results in poor space utilization, empty nodes can exist at the leaf level. R-trees on the other hand, may have space utilization as low as 50%.

The tests showed that, for the data sets tested, there is not much difference between the performance of R-trees and R+ trees. R+ trees perform better when the boxes do not overlap and a small percentage of the space is covered. R-trees perform slightly better when the boxes overlap and cover most of the search space.

The reason R+ trees perform better when inserting non-overlapping box data is that R-trees tend to generate enclosing boxes with widely differing areas. The nodes with large area enclosing boxes tend to get all of the inserts and hence more splits occur. R+ trees tend to maintain a more balanced distribution of entries within the nodes. As the amount of overlap between boxes increases the situation reverses and R-trees work better. R+ trees have to maintain a large number of duplicates and the tree becomes larger. Also the denseness of the data tends to make tightly paneled square-like enclosing boxes and this increases the likelihood that splits will propagate down the tree when they occur. This has the opposite effect in R-trees where, as the data density increases, the nodes tend to pack themselves more optimally and hence sibling overlap decreases and the performance improves.

R+ trees are significantly more difficult to code than R-trees. In addition they have the problem of requiring chaining at the leaf level in order to handle the case when a leaf is full, has an insert request, and all of its current entries overlap. For this reason and because the performances are not significantly skewed one way or another, R-trees appear to be the better alternative for a general spatial data access method.

The performance of the access methods is highly dependent on the implementation. A range of split and insertion algorithms needs to be tested and classified as to what character of data they are suited to. Also, algorithms for rearranging the indices during idle periods should be investigated. It would be useful to test the access methods over an even broader range of data and in particular, to use real world data sets such as VLSI data. Lastly, the issue of deletion algorithms needs to be addressed.

6. Acknowledgements

Michael Stonebraker, my advisor, initiated the project, and made many helpful suggestions and critiques.

7. References

[BANA86] Banerjee, J., W. Kim, "Supporting VLSI Geometry Operations in a Database System," *Proc. Int. Conf. on Data Engineering*, pp. 409,415, February 1986.

[FALO87] Faloutsos, C., T. Sellis, and N. Roussopoulos, "Analysis of Object Oriented Spatial Access Methods," *Proc. ACM SIGMOD*, pp. 426-439, May 1987.

[GUTT84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spail Searching, " *Proc. ACM SIGMOD*, pp. 47-57, June 1984.

[HINR83] Hinrichs, K. and J. Nievergelt, "The Grid File: A Data Structure to Support Proximity Queries on Spatial Objects," Tech. Report 54, Institut fur Informatik, ETH, Zurich, July 1983.

[NIEV84] Nievergelt, J., Hinterberger, and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM TODS*, 9(1), pp. 38-71 , March 1984.

[OREN86] Orenstein, J., "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD*, pp. 326-336, May 1986.

[ROBI81] Robinson, J.T., "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes, " *Proc. ACM SIGMOD*, pp.10-18, 1981.

[ROUS85] Roussopoulos, N. and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," *Proc. ACM SIGMOD*, May 1985.

[SELL87] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R+_Tree: A Dynamic Index for Mult-Dimensional Objects," *a Proc. of the 12th VLDB Conf.*, 1987.

[STON83] Stonebraker, M., B. Rubenstein, and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," Tech. Report UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley, January 1983.

[STON87] Stonebraker,M., L. Rowe, "The POSTGRES PAPERS," Tech. Report UCB/ERL M86/85, Electronics Research Laboratory, University of California, Berkeley, June 25, 1987(Revised).