A Comparative Study of Spatial Indexing Techniques for Multidimensional Scientific Datasets *

Beomseok Nam and Alan Sussman
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742
{bsnam,als}@cs.umd.edu

Abstract

Scientific applications that query into very large multidimensional datasets are becoming more common. These datasets are growing in size every day, and are becoming truly enormous, making it infeasible to index individual data elements. We have instead been experimenting with chunking *the datasets to index them, grouping data elements into small chunks of a fixed, but dataset-specific, size to take advantage of spatial locality. While spatial indexing structures based on R-trees perform reasonably well for the rectangular bounding boxes of such chunked datasets, other indexing structures based on KDB-trees, such as Hybrid trees, have been shown to perform very well for point data. In this paper, we investigate how all these indexing structures perform for multidimensional scientific datasets, and compare their features and performance with that of* **SH-trees***, an extension of Hybrid trees, for indexing multidimensional rectangles. Our experimental results show that the algorithms for building and searching SH-trees outperform those for R-trees, R*-trees, and X-trees for both real application and synthetic datasets and queries. We show that the SH-tree algorithms perform well for both low and high dimensional data, and that they scale well to high dimensions both for building and searching the trees.*

1 Introduction

In the past couple of decades, extensive research has been carried out on multidimensional indexing structures, to enable efficient range queries and nearest neighbor searches. However, most of the recent studies have focused on high-dimensional feature-based similarity searches into a relatively small number of point data items. Many scientific instruments, ranging from sensors on Earth orbiting satellites to light microscopes, can produce hundred of gigabytes of spatio-temporal daily, consisting of billions of individual data elements. Storing each data element in a huge scientific dataset into a multidimensional indexing tree is impractical, because the size of the index could be even larger than the raw dataset, and the performance of queries would be poor due to size of the index. Instead, using spatio-temporal information that is already present in the dataset, we can build a bounding box for parts of the array dataset with data elements having similar spatio-temporal coordinates, only storing the bounding box into the indexing tree to reduce the size of the index and make index searches faster. Storing bounding boxes for small subsets of scientific datasets into spatial indexing structures, such as R-trees [9] or its variants, allows for direct access to subsets of a dataset in order to improve data access performance.

In this paper, we focus on the problem of indexing rectangular objects (multidimensional bounding boxes), introducing the Spatial Hybrid tree (SH-tree), an extension of the Hybrid-tree [7], and perform a comparative study of SH-trees against other indexing techniques for multidimensional rectangular datasets. Multidimensional indexing trees can be classified into two groups. One group is so-called *space partitioning methods*, which are based on KD-trees [5] and have been shown to perform well for point data. In space partitioning methods, a single dimension and a single position in the dimension are used to split nodes that overflow as elements are inserted into the tree, regardless of the number of dimensions of the data. Hence the fan-out of internal nodes in the tree is independent of the number of dimensions of the dataset and the sub-partitions are mutually disjoint.

The second group is *data partitioning methods*, based on R-trees [7, 9], which have been shown to perform well for

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288, and NASA under Grant #NAG511994.

(hyper-)rectangular data. Data partitioning methods store all the bounding box information, namely the minimum and maximum coordinates in each dimension. As the number of dimensions increases, the disk space required for each bounding box also increases, and for a fixed size disk page the fan-out of internal nodes decreases, with the standard assumption that all the data for a tree node fits in one disk page. In data partitioning methods, the bounding boxes of tree nodes may overlap, unlike in space partitioning methods. Because overlapping regions are allowed for the data partitioning methods, rectangle data can be indexed without any modification of the data structure.

Some efforts have been made to make space partitioning methods work for non-point data and one such solution is to convert rectangles into higher dimensional point data [10]. This transformation approach does not need any alteration of the existing multidimensional indexing structure. However with this approach the distribution of the converted point data becomes extremely skewed, potentially causing a range intersection query to search a large fraction of the nodes in the tree [2]. Moreover, the transformation approach leads to the well-known *curse of dimension* problem - the exponential growth of hypervolume as a function of dimension [4].

Previously developed space partitioning methods include *SKD*-trees and *Hybrid*-trees. The Spatial KD-tree developed by Ooi et. al. [14] allows overlap of space partitions in KD-trees, so that non-point data can be inserted [5]. The Hybrid-tree developed by Chakrabarti et al. [7] is another commonly used space partitioning method. The Hybrid-tree is derived from the KDB-tree, but allows overlapping regions to avoid the *downward cascading split problem* (described in Section 3), which is a notorious performance problem for KDB-trees that decreases the node utilization. To take advantage of the best features of both SKD-trees and Hybrid-trees, we have designed the *Spatial Hybrid tree (SH-tree)*. An SH-tree is a disk based multidimensional indexing structure for non-point data that enables the fan-out of each node to be independent of the number of dimensions of the rectangles it stores. To the best of our knowledge, there has not been a thorough performance evaluation of commonly used spatial indexing structures on non-point data. In the rest of this paper, we will discuss the costs of building and searching existing index structures, comparing them to that of the SH-tree.

The rest of the paper is organized as follows. In Section 2 we discuss data chunking, which creates bounding boxes from the data in a multidimensional scientific dataset. In Section 3, we compare existing multidimensional index structures against SH-trees for hyper-rectangle data. The SH-tree algorithms are discussed in detail in Section 4. In Section 5, we show performance results for the SH-tree against several other indexing structures, examining the costs of both building the trees and searching them. We conclude in Section 6.

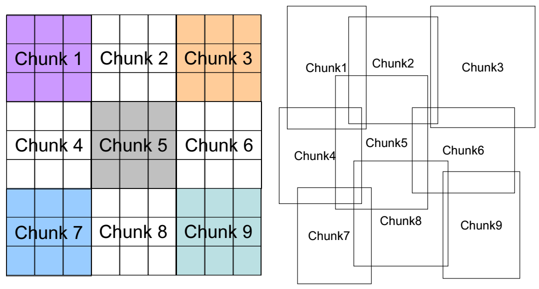


Figure 1. *Dataset with nine chunks and corresponding bounding boxes in problem space*

2 Data Chunking

In many scientific datasets, including satellite sensor data, the coordinate metadata is stored as separate fields, along with the sensor values. When a range query is to be performed, to extract the set of sensor values that fall within the specified coordinate ranges, the entire coordinate dataset must be scanned to find elements falling inside the query range if no indexing structure exists. This can be a very expensive operation that should not be executed often, but unfortunately there are many scientific libraries that perform such a brute force range query operation. Creating multidimensional indexing structures can allow performing range queries much more efficiently. However, scientific datasets may contain many billions of individual data elements, each with its own spatio-temporal coordinates. In previously described experiments [13], we showed that it could take several hours to build an R*-tree index for relatively small datasets containing a few million data elements. Another serious problem with indexing every element in a dataset is that the index file size can be very large. If we build an index using every data element, the size of the index file can be much larger than the dataset it is indexing.

Data chunking refers to partitioning a multidimensional dataset into coarse-grained hyper-rectangular blocks, as shown in Figure 1. The contents of scientific data files typically are a collection of multidimensional arrays. In scientific datasets, data elements that are nearby in a stored array (i.e. their indices are close) usually are also nearby in spatio-temporal coordinates, because the sensor data is stored in the same order it is acquired. By grouping data elements into chunks, we can get a relatively tight bounding box for the spatio-temporal coordinates (meaning that the boxes for different chunks do not overlap much). Note that data chunking may cause data elements not within the requested query range to be retrieved, because if the bounding box of a data chunk overlaps the query range all the the elements in the chunk must be accessed. There is therefore an overhead from data chunking - filtering out data elements not within the query range after they are read from disk. However, we have shown in earlier experiments [13] that the overhead of filtering the additional data elements is negligible compared to the cost of retrieving the data from disk.

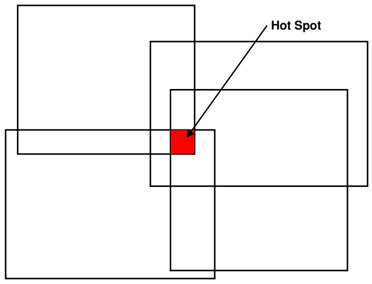


Figure 2. *Disjoint partitioning is not possible due to a hot spot*

3 Spatial Indexing Structures

We discuss several widely used variants of *space partitioning* and *data partitioning* multidimensional indexing data structures, concentrating on issues related to performance for range queries into rectangle data elements.

3.1 Space partitioning methods

KDB-tree: Robinson has developed a balanced B-tree version of the binary KD-tree [5], the KDB-tree [15]. Unfortunately, minimum space utilization is not guaranteed for KDB-trees because of the *downward cascading split* problem. A KDB-tree does not allow overlapping partitions, as does the standard KD-tree, but when a tree node must be split it is not always possible to find disjoint partitions in a KDB-tree. In such cases, some sub-partitions must be split at the same split value as for the parent node, even if the sub-partitions do not meet the minimum storage utilization requirement for a node. The split can propagate all the way down to the leaf nodes, which can make range query performance poor.

Spatial KD-tree: A Spatial KD-tree (SKD-tree) [14] is another variant of the binary KD-tree designed for non-point spatial objects. An SKD-tree allows sub-partitions to overlap, by having two split positions in one split dimension. Each split position represents the boundary of the lower or upper sub-region, respectively. However, the SKD-tree is a memory-based, not a disk based data structure, thus is not suitable for very large databases.

Object duplication methods: Matsuyama's KD-tree [12] is another variant of the binary KD-tree for non-point spatial data. In Matsuyama's KD-tree, an extensive object duplication strategy is used, hence objects can be stored in multiple leaf nodes. The R+-tree [16] is a disk based indexing method that uses the object duplication strategy. However, object duplication methods may create infinite recursive loops when inserting rectangles into the tree, if there is at least one non-point region, denoted as a *hot spot*, that falls completely inside all the child partitions of a node, as shown in Figure 2. In such a case, no matter what split dimension or split position is selected, either or both of the two resulting nodes will overflow again because the hot spot will belong to at least one of the resulting sub-partitions, so the resulting nodes must duplicate all the child bounding boxes that cover the hot spot. For this reason, disjoint partitioning methods based on the object duplication strategy are not feasible for non-point data.

Hybrid-tree: To solve the downward cascading split problem for KDB-trees, several variants have been proposed, such as hB-trees [11] and Hybrid-trees [7]. The Hybrid-tree solves the downward cascading split problem by allowing overlap of the two sub-regions after a node is split, as in data partitioning methods [7]. While the internal nodes for the data partitioning methods are lists of bounding boxes and pointers to child nodes, each internal node for the disk based space partitioning methods is a binary KD-tree, with each leaf of the KD-tree containing the sub-partition of a child internal node in the top-level tree and a pointer to the child node in the top-level tree. An internal node in a Hybrid-tree is also a binary KD-tree, whose nodes contain both a splitting dimension and two splitting positions in that dimension. By having two splitting positions instead of one, the Hybrid-tree allows overlapping regions when a downward cascading split is unavoidable. However, the Hybrid-tree allows overlap only in non-leaf nodes, and the overlapping region is created or extended only when a node overflows during object insertion, so must be split. Therefore, non-point spatial objects cannot be indexed in a Hybrid-tree.

3.2 Data partitioning methods

R-tree and R*-tree: Instead of duplicating objects, spatial objects can be indexed by allowing overlapping regions, as in R-tree based index structures [9]. Although R-trees can be used for non-point data, a large amount of overlap between internal nodes in R-trees leads to search performance problems. To reduce overlapping regions for R-trees, Beckmann et al. proposed an optimized version of R-trees, called R*-trees [3]. The R*-tree insertion algorithm reinserts elements from a node that overflows, instead of splitting the node. This forced reinsertion feature of R*-trees improves search performance, but node insertion can become very expensive.

X-tree: Berchtold et al. developed another variant of the R-tree, called an X-tree [6], which avoids highly overlapping bounding boxes via the use of *supernodes*. A supernode is a tree node that spans multiple pages on disk, thus has a larger capacity than a normal node. When a node must be split and a large amount of overlap between sub-partitions is unavoidable, the X-tree algorithm increases the capacity of the node instead of splitting it. If there would be a large amount of overlap between two nodes after a split, the probability that both nodes would be accessed by a search operation is high. Hence, sequential access to supernodes should be faster than random access to two separate nodes. However, supernodes have the overhead of additional disk management costs at index creation time. Therefore, before the X-tree insertion algorithm creates a supernode, it tries to find an overlap-free split based on past split history. For more details on supernodes, see [6]. However, split history is not useful for non-point spatial objects, because an overlap-free split is not always possible for non-point data. Even if an overlap-free split can be found, in most cases it will not be acceptable since it will not meet minimum node utilization requirements.

4 Spatial-Hybrid Tree

The Spatial Hybrid-tree (SH-tree) is a new multidimensional indexing structure that supports efficient range queries on non-point data objects, in both low and high dimensional spaces. The SH-tree combines the properties of the SKD-tree and the Hybrid tree, both of which are based on space partitioning methods, and allows overlapping sub-regions by having two split positions in one split dimension. The SKD-tree allows overlapping sub-regions only when a mutually disjoint partition is not possible because of the volumes of the data objects, whereas the Hybrid-tree allows overlapping sub-regions when a downward cascading split is unavoidable. In other words, the Hybrid-tree creates a new overlapping region when an node that overflows must be split, while the SKD-tree adjusts overlapping regions so that one region will fully contain a new object that is to be inserted. The SH-tree employs the node splitting algorithm of the Hybrid-tree and the insertion algorithm of the SKD-tree.

4.1 Node Splitting

To allow overlap between nodes after splitting, two split positions are needed, one representing the minimum boundary of the upper (right) region in the split dimension (denoted as $minU$) and the other the maximum boundary of

procedure
$Insert(Object * o, KDnode\ currNode)$

```
if object is inside the left sub-region then
    Insert(o, currNnode.left)
else if object is inside the right sub-region then
    Insert(o, currNode.right)
else if object is not inside left nor right sub-region then
    if left sub-region requires less enlargement then
        currNode.maxL := o.High(splitDim)
        Insert(o, currNode.left)
    end if
    if right sub-region requires less enlargement then
        currNode.minU := o.Low(splitDim)
        Insert(o, currNode.right)
    end if
end if
```

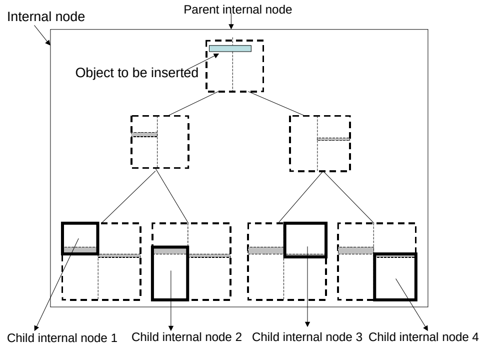
end procedure

Algorithm 1:Default Insertion Algorithm

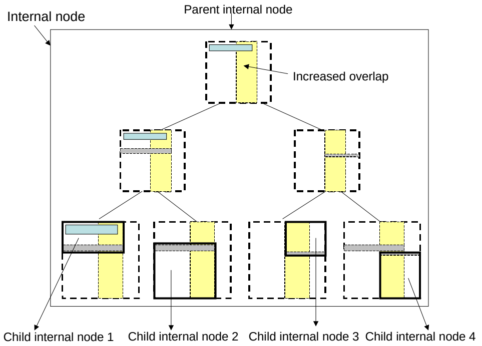
the lower (left) region in the split dimension (denoted as $maxL$). The $minU$ and $maxL$ values, and the split dimension (denoted as $splitDim$), are locally optimized to reduce the overlap when a node that overflows must be split. The goal of the node split algorithm for SH-trees is to minimize the distance between the two split positions $(maxL - minU)$. For an N-dimensional dataset, only one of the dimensions is used as a split dimension. For each dimension, the bounding boxes of the child sub-regions of the node to be split are sorted twice, based on their lower and upper boundaries in the split dimension. The sub-region with the lowest upper bound and the sub-region with the highest lower bound are selected and put into the lower and upper resulting regions, respectively, until the minimum required node utilization is reached. When the minimum required node utilization for both regions is reached, it must be determined which region will increase in size the least if all the remaining sub-regions are inserted into that region. In this way, all the children are placed into two resulting regions to achieve minimal overlap in the split dimension. This process is performed for each dimension, and the dimension that causes the smallest overlapping region is chosen to be split.

4.2 Insertion

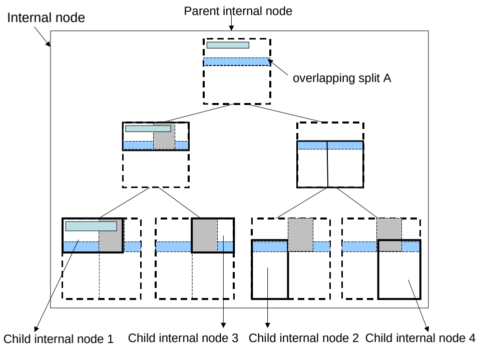
$minU$ and $maxL$ for an SH-tree internal node are computed for a newly created child node, based on how the node and its sibling node are split. However, while the split values in the KD-tree in an internal node of a hybrid tree never change, in the SH-tree they must be updated when an object is inserted (or deleted), as shown in Figure 3. Figure 3(a) shows an internal node of an SH-tree. This internal node has four child nodes, with each of their sub-regions represented



(a) Overlapping region must be adjusted when a new data object to be inserted is not fully covered by any sub-region. (Shaded regions represent the overlaps.)



(b) Cascading overlap problem - enlarging the sub-region of child internal node 1 enlarges the sub-region of child node 2.



(c) Greedy reorganization of overlapping regions - enlarging the sub-region of child node 1 does not enlarge any other sub-region.

Figure 3. *Dynamic adjustment of overlapping sub-regions in an internal node of an SH-tree*

by the bold outlined rectangles. When a new data object that does not fit completely inside any of the four children is inserted into the node, the algorithm must compare the object with the split positions of each level in the KD-tree of the node, and adjust the positions accordingly.

Algorithm 1 shows one way to extend the sub-regions, which is similar to how it is done in the SKD-tree insertion algorithm [14]. Suppose we are inserting an object *o* whose boundary is $(o.Low(splitDim), o.High(splitDim))$ in the split dimension (splitDim). If $o.Low(splitDim)$ is less than $minU$ and $o.High(splitDim)$ is greater than $maxL$, as seen in the root level node of Figure 3(a), either $minU$ or $maxL$ must be updated to minimize the increase in the overlapping region. However this algorithm has a potential performance problem, since when a split position is changed it not only has an effect on the boundaries of the child node that contains the inserted object, but may also increase the boundaries of the other child sub-regions. In Figure 3(b), the sub-region for child internal node 2 must also be extended, although the new data object will be inserted into child internal node 1. We refer to this unique problem for SH-trees as the *cascading overlap problem.* A basic property of KD-trees, which is that split positions are shared among child sub-regions, causes the cascading overlap problem. While the benefit of sharing split positions is to allow the node fan-out to be independent of the number of dimensions of the bounding boxes, this causes the cascading overlap problem for non-point data.

4.3 Reorganization Of An Internal Node

When an object to be inserted spans almost the full range of two sub-regions, as seen in Figure 3(b), Algorithm 1 causes one sub-region to almost completely overlap another sub-region. Even worse, the extended sub-region may overlap more than one other sub-region. This problem can occur for real datasets, for example for satellite remote sensing datasets. In that example, because the satellite orbits the earth's poles, sensor data recorded near the poles has extremely skewed latitude/longitude extents. For data collected near each pole, the range of longitudes in a data chunk ranges from 0 to 360 degrees, while the latitude values converge to either 90 or -90 degrees.

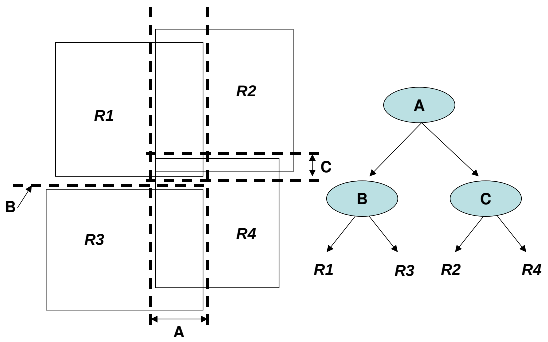
While we have not found an optimal solution to the cascading overlap problem, we would like to determine the benefits that can be gained from better addressing the problem. We have therefore designed and implemented a greedy method that is similar to other methods used to solve optimization problems. Finding the minimal overlapping SKD-tree style partitions, given N rectangular objects, is a hard problem. For an internal node with N d-dimensional child rectangles, there are 2N boundary positions in each dimension. Since the algorithm must select 2 split positions, there are $d \cdot \binom{2N}{2}$ possible partitions. After selecting one of them, the algorithm must repeat this process for each of the two resulting sub-regions recursively, so that the binary SKD-

procedure
$GreedyInsert(SHnode\ n, Object * o)$
$SubRegions$:= set of sub-regions from all the KD-tree leaf nodes
$choice$:= the index of the sub-region that needs minimum enlargement to contain o
$SubRegions[choice] :=$ $RegionMerge(SubRegions[choice], o)$
$SplitUpdate(SubRegions, n.KDtreeRoot)$
end procedure

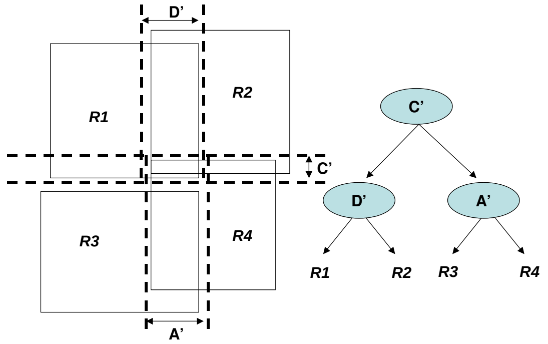
procedure
$SplitUpdate(Rect * SubRegions, KDnode\ currNode)$

$splitInfo := FindMinOverlapSplit(SubRegions)$
$currNode.splitdim := splitInfo.splitdim$
$currNode.minU := splitInfo.minU$
$currNode.maxL := splitInfo.maxL$
for all $R \in SubRegions$ **do**
  if $R.High(splitdim) < minU$ **then**
    $leftSubRegions.appendToList(R)$
  else if $R.Low(splitdim) > maxL$ **then**
    $rightSubRegions.appendToList(R)$
  else
    Put R into the sub-region that has fewer rectangles.
  end if
end for
$SplitUpdate(leftSubRegions, currNode.left)$
$SplitUpdate(rightSubRegions, currNode.right)$
end procedure

Algorithm 2:Greedy Insertion Algorithm



(a) Split B is overlap free.



(b) Applying the greedy algorithm, B and C merge into C', and C' is selected as the minimal overlap split in the root level, since the width of C' is less than that of A. However this does not decrease the overall overlap, because it creates unnecessary overlap between R1 and R3.

Figure 4. *Greedy algorithm fails to find the optimal solution*

tree ends up with N leaves, one for each rectangle. Since the number of internal nodes in a binary tree with N leaves is always $N-1$, the complexity of finding all possible splits is $O((dN)^{2N-2})$. Before building a binary SKD-tree with N rectangles, we do not know how much overlap will result. To find an optimal solution, we may need to build every one out of the $O((dN)^{2N-2})$ possible SKD-trees. The greedy method shown in Algorithm 2 does not guarantee selecting optimal overlapping sub-regions, but the heuristic assumes that less overlap in the higher levels of the KD-tree of an internal node will result in less overall overlap between leaf nodes in the KD-tree, thereby minimizing the cascading overlap problem. The greedy method tries to find a minimal overlapping split in the root level of the KD-tree, and then recurses down the tree.

Insertion using the greedy algorithm works as follows. A data rectangle R to be inserted into the tree is compared with the root level sub-regions. If R does not fit into either of the two children of the root node, instead of updating the split positions of those two children immediately, the inserted object is compared with the sub-regions of *all* the child internal nodes, and the bounding box that needs minimum enlargement is selected and extended only enough to hold R. Then a new binary SKD tree is built using the updated sub-regions and the greedy method described in Algorithm 2.

In the example shown in Figure 3(a), child internal node 1 is selected and its sub-region is extended to include the new data object. After the four sub-regions of the child internal nodes are updated, the greedy algorithm tries to find a minimal overlapping split in the root level. Among the many possible splits, the minimal overlapping split (split A in Figure 3(c)) is selected and is stored in the root level. Child internal nodes 2 and 4 end up in the lower region and child internal nodes 1 and 3 end up in the upper region. Then the same process is repeated recursively for both the lower and upper regions. Figure 3(c) shows the result of the greedy insertion algorithm. In this example, the greedy algorithm reduces the overall size of the overlapping regions.

The complexity of the default insertion algorithm is $O(log(N))$, where N is the number of rectangles, and the complexity of the greedy insertion algorithm is $O(N^2)$.

Unfortunately, the greedy insertion algorithm can result in greater overlap between sub-regions than the default insertion algorithm. One such example is shown in Figure 4. To avoid those cases, rather than applying the greedy insertion algorithm alone, the complete insertion algorithm we have implemented compares the overlap resulting from the default insertion algorithm with the overlap resulting from the greedy insertion algorithm and selects the best result.

5 Experiments

We measured the performance of both index creation and search using the SH-tree, R-tree, R*-tree, and X-tree algorithms. The experiments were run on a SunBlade 100 workstation with a 500MHz Sparcv9 processor, 256MB memory, and a 7200RPM IDE disk with a seek time of 9ms. Before we present the experimental results, we describe the Kronos dataset that was used as a representative three-dimensional scientific dataset, and the synthetic datasets used to evaluate the algorithms in a systematic way.

5.1 Kronos Dataset

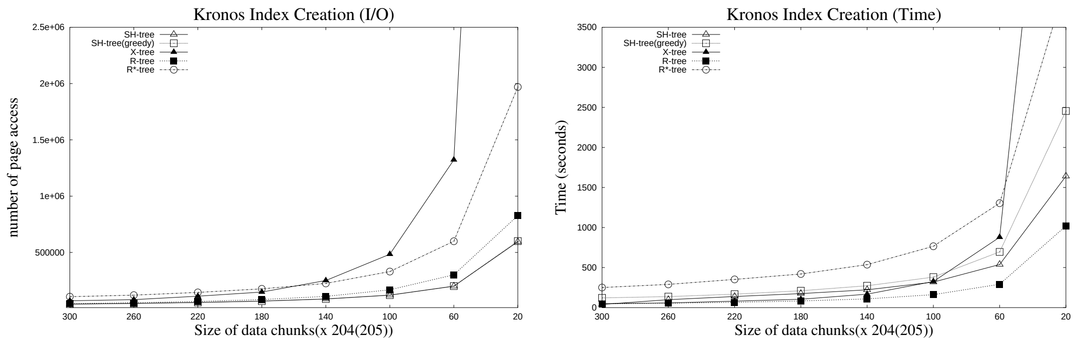
The Kronos dataset is used to evaluate low-dimension (3D) multidimensional indexing trees. Kronos is an application that processes remotely sensed AVHRR (Advanced Very High Resolution Radiometer) GAC (Global Area Coverage) level 1B datasets. The raw data is continuously collected by multiple satellites orbiting Earth. AVHRR level 1B sensor data is stored as a set of arrays, and also includes geo-location fields, time fields, and some additional metadata. As the satellite moves along a ground track over the earth, it records longitude, latitude, and time values, as well as sensor values. Because the sensor swings across the ground track, the sensor values and meta values are stored as two dimensional arrays. We partitioned those arrays into equal sized rectangular chunks. The length of a ground track can grow indefinitely, but the length of the cross track is fixed at 409 values. Hence we divided the cross track into 2 unequal partitions - 404 and 405, as was done for the same data in a previous study several years ago [8], and evaluated the performance of the indexing trees with various sized data chunks along the ground track. The Kronos dataset used was collected over one month (January 1992), and has a total size of more than 30GB, with the volume of data for a single day about 1GB. In order to create range queries, we employed a variation of the *Customer Behavior Model Graph (CBMG)* technique to match them to a realistic workload. We modeled common query behaviors, including hot spots in the data corresponding to areas of high interest (e.g., Iraq or Afghanistan). More details about the Kronos dataset and query workload generator can be found in [1].

We evaluate both the insertion and search times for the SH-tree, X-tree, R*-tree, and R-tree algorithms. [1] We turned off OS disk file caching, via the Solaris *directio* system call, to allow ignoring the effects of file caching, which also makes the execution times of the algorithms more consistent with the number of disk page accesses.

Figure 5 shows the time and the number of page writes for inserting bounding boxes, with various data chunk sizes. As the chunk size decreases, the number of leaf nodes in the indexing trees increases, since we are partitioning a fixed size dataset. When the data chunk size is 300x404 (or 300x405), we insert about 40,000 data chunks into the indexing trees, but when the chunk size is 20x404, there are about 560,000 data chunks. Measuring the number of page accesses, the SH-tree algorithm writes the fewest number of pages for inserting all the rectangles, in all cases. When the chunk size is small, the X-tree algorithm writes approximately 20 times as many disk pages as the SH-tree in the worst case, because of the large size of the supernodes. The X-tree algorithm shows better performance than the R*-tree algorithm when the tree is small. However, as the tree grows, the X-tree algorithm writes up to 5 times as many disk pages as the R*-tree algorithm. When a bounding box (or sub-regions) of an internal node must be updated, all other trees access only one page, but several pages must be written for a supernode of the X-tree, although those pages are adjacent on disk. Because of the multiple page accesses, the X-tree index creation algorithm has even worse performance than the notoriously expensive R*-tree algorithm.

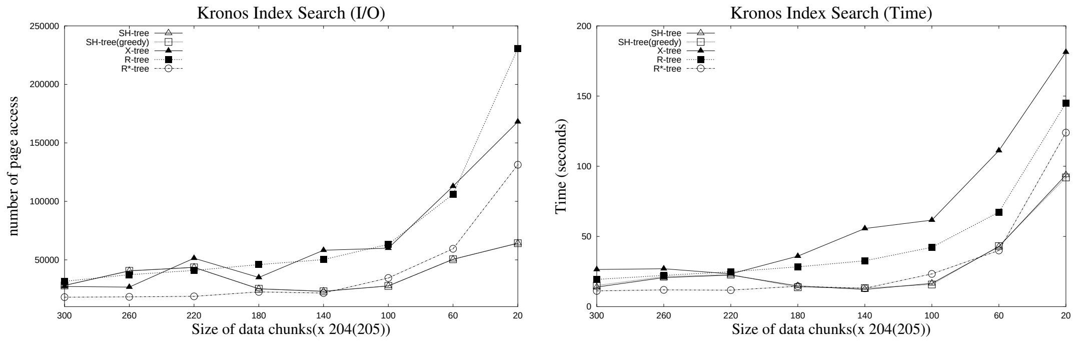
The timing results presented in Figure 5(b) show the elapsed wall clock time for inserting the bounding boxes of all data chunks into the index. In the experiments, the SH-tree algorithm spends 43%-51% of its insertion time in user level computation, and the rest in system level disk I/O. The time to create index files correlates well with the number of disk accesses, except that the R-tree algorithm is somewhat faster than the SH-tree algorithm. From this observation, we see that the number of disk accesses is not sufficient to evaluate the overall performance of the indexing structures. Accessing disk pages sequentially is another factor that influences the overall response time. Moreover, especially when inserting data into the index, computation costs can be quite high, because of node splitting or other algorithms used to reduce overlapping regions in an index. If we employed the OS file cache, user level computation would account for most of the execution time for doing the insertions. Therefore, the performance of the multidimensional indexing structures also depends on code optimization. In

[1] We modified the R-tree and R*-tree implementations by Marios Hadjieleftheriou, and the X-tree implementation by Hans-Peter Kriegel's group for the experiments. Those implementations are available at http://www.rtreeportal.org/code.html. We modified the X-tree implementation so that it writes dirty pages whenever they are updated, as do all the other tree implementations.



(a) Disk Page Accesses (b) Response Time

Figure 5. *Index Creation for Kronos Dataset*

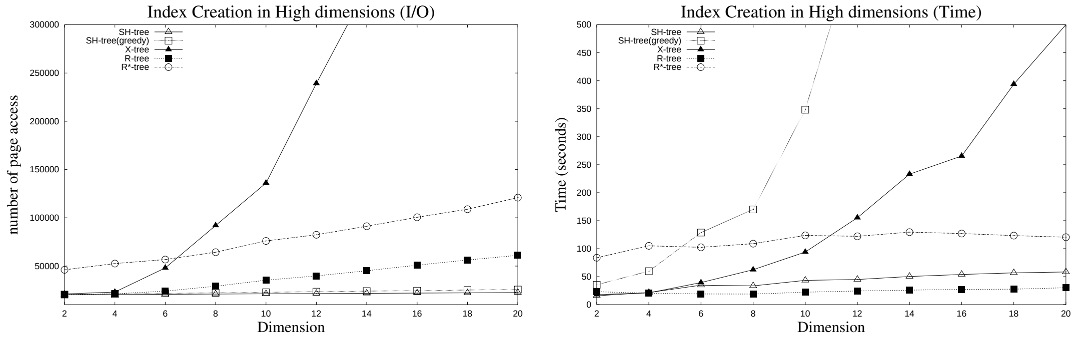


(a) Disk Page Accesses (b) Response Time

Figure 6. *Index Search for Kronos Dataset*

any case, the SH-tree algorithm shows stable and good performance in terms of both the number of disk accesses and overall response time.

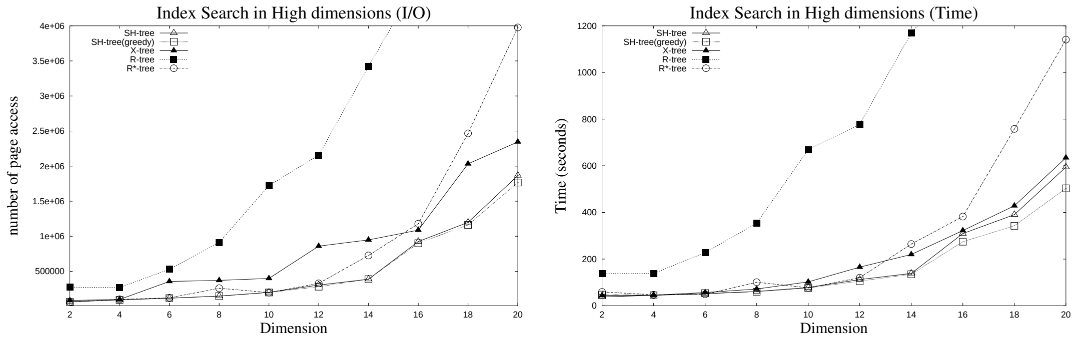
The SH-tree algorithm is fast not only for building the tree, but also for range queries. We generated 2000 range queries using the Kronos query workload generator. These queries are generated from the workload model, using 16 geographic places of interest (hotspots) at a randomly selected time. The average size of a query in latitude and longitude is approximately 18 degrees and the maximum time span of a query is 10 days. When the data chunks are small, the SH-tree algorithm accesses only 1/4 as many disk pages as the R-tree algorithm and half that of the R*-tree algorithm, as shown in Figure 6. However as the data chunk size grows, the SH-tree algorithm tends to generate large overlapping regions, due to the cascading overlap problem. Although the R*-tree algorithm outperforms the SH-tree algorithm for large data chunk sizes, the SH-tree algorithm shows better or almost equal performance compared to the R-tree and X-tree algorithms. An interesting result is that the X-tree algorithm does not perform well for the non-point Kronos dataset. In the worst case, the X-tree algorithm reads 2.7 times more disk pages than the R*-tree algorithm, and 25% more pages than the R-tree algorithm. For low dimensional datasets, the X-tree algorithm shows similar performance to the R-tree algorithm in disk page accesses, because of supernodes. We noted in Section 3 that the split history used by the X-tree algorithm does not produce better trees for non-point data objects. In three dimensions, the R-tree based trees have the same fan-out as the SH-tree for the same dataset. Although greedy insertion for the SH-tree algorithm increased overall insertion time, it does not improve search performance into the Kronos datasets. The partitions resulting from the greedy insertion algorithm are only slightly better than from the default insertion algorithm. In other words, the cascading overlap problem does not appear to be serious for low dimension datasets.



(a) Disk Page Accesses

(b) Response Time

Figure 7. *Index Creation With Synthetic Datasets*



(a) Disk Page Accesses

(b) Response Time

Figure 8. *Index Search With Synthetic Datasets*

5.2 Synthetic Dataset

We present experimental results on synthetic datasets, looking at the effects of the dimensionality of the dataset on performance. We generated datasets of 20,000 uniformly distributed hypercubes in the unit hyper-rectangle, with the dimension of the datasets ranging from 2 to 20. As the number of dimensions of the dataset increases, the indexing methods based on data partitioning, including the R-tree, R*-tree and X-tree algorithms, all suffer from reduced fan-out. However, the SH-tree data structure scales well to high dimensional datasets, because of its dimension independent fan-out.

Figure 7 shows the performance of index creation for the various algorithms, for two to twenty dimensional datasets. Due to its node reinsertion strategy, the R*-tree algorithm takes much longer to create an index than either the R-tree or SH-tree algorithms. As was described for the Kronos experiments, the X-tree algorithm suffers from the supernode problem, especially in high dimensions. In low dimensions, the SH-tree, R-tree, and X-tree algorithms access a similar number of disk pages, about 40% of that for the R*-tree algorithm. However as the number of dimensions increases, the X-tree insertion algorithm becomes very expensive. In the worst case, the X-tree algorithm writes 37 times as many disk pages for the twenty dimension dataset as for the two dimension dataset, the R-tree and R*-tree algorithms access up to four times as many disk pages and the SH-tree algorithm requires only 1% more disk accesses.

Comparing the algorithms for twenty dimensions, the number of disk writes for insertion with the SH-tree algorithm is only 18% that of the R*-tree algorithm, and only 2.8% that of the X-tree algorithm. Overall, the tree insertion algorithm for SH-trees appears to be very efficient. The time to create an index, as shown in Figure 7(b), is mostly proportional to the number of disk accesses, except that the R-tree algorithm performs somewhat better than the SH-tree algorithm, because of lower computation costs and a more optimized implementation. However, for the SH-tree with the greedy insertion algorithm, the time to create the index is significantly higher than for the default algorithm, due to its $O(N^2)$ complexity

For index searches in high dimensions, we generated and submitted 20,000 uniformly distributed hypercube queries. The performance of the SH-tree algorithm for index search is very good compared to the other algorithms, both for execution time and disk accesses, as seen in Figure 8. The SH-tree and X-tree algorithms scale better than the R-tree and R*-tree algorithms to high dimensions. The R-tree algorithm accesses more disk pages than the other algorithms across all numbers of dimensions, and the performance gap grows as the number of dimensions increases. Although the X-tree algorithm reads more disk pages than the R*-tree algorithm for low numbers of dimensions, when the number of dimensions is higher than sixteen the X-tree algorithm begins to outperform the R*-tree algorithm. Both algorithms still perform worse than the SH-tree algorithm. The SH-tree algorithm accesses from 11% to 34% the number of disk pages as does the R-tree algorithm, from 46% to 101% those of the R*-tree algorithm, and from 32% to 94% those of the X-tree algorithm.

The performance improvement from the SH-tree greedy insertion algorithm was up to 5%, in disk accesses. However, in low numbers of dimensions, the greedy insertion algorithm can decrease search performance by a small amount. Given the high extra cost for building the tree of the greedy algorithm, a 5% improvement in search time is rather disappointing. However this result is interesting, because it is evidence that the default insertion algorithm does not cause a serious cascading overlap problem.

6 Conclusion

We have shown that the SH-tree outperforms several other commonly used spatial indexing techniques on both a real remote sensing dataset and for synthetic datasets, also showing that the SH-tree is more scalable to high dimensions than the other techniques. One future direction of this work is to determine the complexity of an optimal partitioning algorithm, to make SH-tree even faster. Another direction is to incorporate the SH-tree into a spatial indexing library that we are developing [13] to speed up access to self-describing scientific datasets, in formats such as HDF and netCDF.

References

[1] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Exploiting functional decomposition for efficient parallel processing of multiple data analysis queries. In *Proceedings of the 17th International Parallel & Distributed Processing Symposium*. IEEE Computer Society Press, Apr. 2003. Also available as University of Maryland Technical Report CS-TR-4404 and UMIACS-TR-2002-84.

[2] P. W. Andreas Henrich, Hans-Werner Six. The LSD tree: Spatial access to multidimensional point and non-point objects. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB89)*, pages 45–53, 1989.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD90)*, pages 322–331, May 1990.

[4] R. E. Bellman. *Adaptive Control Processes: A GuidedTour*. Princeton University Press, NJ, 1961.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM 18(9)*, 1975.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB96)*, pages 28–39, 1996.

[7] K. Chakrabarti and S. Mehrotra. The Hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE99)*, pages 440–447, 1999.

[8] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering (ICDE97)*, pages 375–384, Apr. 1997.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD84)*, pages 47–57, 1984.

[10] K. Hinrichs. *The Grid File System: Implementation and Case Studies of Applications*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1985.

[11] D. Lomet and B. Saltzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. In *ACM Transactions on Database Systems, vol. 15, no. 4*, 1990.

[12] N. M. Matsuyama T., Hao L.V. A file organization for geographic information systems based on the spatial proximity. In *Computer Vision, Graphics and Image Processing. Vol. 26, No.3*, pages 303–318, 1984.

[13] B. Nam and A. Sussman. Improving access to multi-dimensional self-describing scientific datasets. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid. (CCGrid)*, May 2003.

[14] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC Conference*, 1987.

[15] J. T. Robinson. The K-D-B tree: A search structure for large multi-dimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD81)*, 1981.

[16] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB87)*, pages 507–518, 1987.