Efficient Techniques for Range Search Queries on Earth Science Data

(Preliminary Report)

Qingmin Shi and Joseph F. JaJa
Institute for Advanced Computer Studies,
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742, USA
{qshi,joseph@umiacs.umd.edu}

February 22, 2002

Abstract

We consider the problem of organizing large scale earth science raster data to efficiently handle queries for identifying regions whose parameters fall within certain range values specified by the queries. This problem seems to be critical to enabling basic data mining tasks such as determining associations between physical phenomena and spatial factors, detecting changes and trends, and content based retrieval. We assume that the input is too large to fit in internal memory and hence focus on data structures and algorithms that minimize the I/O bounds. A new data structure, called a Tree-of-Regions (ToR), is introduced and involves a combination of an R-tree and efficient representation of regions. It is shown that such a data structure enables the handling of range queries in an optimal I/O time, under certain reasonable assumptions. Experimental results for a variety of multi-valued earth science data illustrate the fast execution times of a wide range of queries, as predicted by our theoretical analysis.

1 Introduction

Considerable amounts of spatio-temporal data sets are generated on a daily basis with the amount of remotely sensed data alone expected to exceed several terabytes per day within the next few years. The sources of geospatial data are quite diverse and include satellite imagery, geographical information systems, census data, and environmental assessment and planning. These data sets offer unprecedented opportunities for exploring associations between environmental phenomena and spatial factors, building environmental models, detecting changes and finding interesting spatio-temporal patterns and trends. In spite of a significant progress in the development of geospatial data mining techniques, the exploration of large amounts of geospatial data by content remains quite difficult. The NASA supported Earth Science Information Partnership (ESIP) Federation, that includes all the major data centers for earth sciences, developed a list of major scenarios for which content-based retrieval techniques will be critical [3]. Almost all of these scenarios involve the fundamental problem of determining spatio-temporal regions over which a certain number of parameter values satisfy certain constraints, for example the values fall within certain ranges or increase within certain bounds over a time period. This paper develops efficient techniques for addressing the core problem of determining regions within a large scale raster geospatial data set whose parameters' values fall within specified ranges. A forthcoming paper will show how these techniques can be extended to handle a time series of such data. The techniques developed here have a strong theoretical foundation and are coupled with an extensive set of experimental results that illustrate the efficiency of these techniques.

Briefly our main contributions are:

- The development of an efficient representation of raster data sets consisting of a combination of an R-tree built around the parameter values and a decomposition of the spatial-space into regions described by their boundaries. The overall complexity to build this structure is dominated by two external sorting steps.
- The querying over arbitrary value ranges of the parameters can be done very efficiently in time that is approximately proportional to the time it takes to read the output from external memory.
- Extensive experimental tests on remotely sensed data confirmed the efficiency of our representation in terms of fast execution times of a wide variety of random queries.

The remainder of this paper is organized as follows. In Section 2, we define the problem and the computational model used for analyzing our algorithms. The related work is discussed in Section 3, while our data organization structure is described in Section 4. Sections 5 and 6 present the algorithms for building our structure and handling general queries. The experimental evaluation of our methods is summarized in Section 7.

2 Problem Definition and Computational Model

We assume that we are given a grid G of size $N_x \times N_y$ representing a spatial region decomposed into $N = N_x N_y$ cells. A k-tuple $(f_1^{(i,j)}, f_2^{(i,j)}, \cdots, f_k^{(i,j)})$ is associated with each cell (i,j) in G such that each parameter $f_l^{(i,j)}$ is a certain numerical attribute corresponding to cell (i,j). We assume that G is too large to fit in internal memory and the result of a query may or may not fit in memory. The problem is to develop a representation of this grid in such a way that the following query can be answered very quickly (in time proportional to reading the output from external memory):

> *Determine all the regions in which the parameters' values fall within specified ranges: $a_l \le f_l \le b_l$, for all $1 \le l \le k$ (a* query window*).*

A region is defined as the maximal set of connected cells where the parameter values satisfy the constraints in each cell. The output to our query consists of a list of all the cells in these regions such that all the cells in the same region are assigned the same label. Our techniques will carry out to more robust definitions of regions such as density-base regions in [11].

Solving the above problem involves efficient identification of the cells whose parameters' values fall within the specified ranges and fast groupings of cells into connected regions. The main focus here is on minimizing the query time. In addition, the storage of the new structure is also important since the size of the raw data is assumed to be large. In general, we aim to achieve the following three properties:

- The size of the new representation should be comparable to the raw data.
- The construction time of the new representation should be efficient in the sense that the input data should only be scanned a few times.
- Queries should be answered very quickly in time proportional to the output size.

To analyze our algorithms, we will use the standard two-level I/O model [4] defined by the following parameters:

N: the size of the input;

M: the internal memory size; and

B: the size of a disk block.

We assume that $B^2 < M < N$. An I/O operation is defined as the transfer of one block of contiguously stored data between disk and internal memory. Hence scanning an input of size N stored contiguously on a disk takes $O(N/B)$ I/O operations in this model.

3 Related Work

A major component of our problem requires the handling of multidimensional range queries of point data. A large number of external data structures and algorithms have been proposed to deal with this problem. In contrast to the two-dimensional case where solutions that provide provable good performance exist (see for example [20, 16, 6]), most data structures for high dimensional data are aimed at achieving good practical performance (A recent survey can be found in [12]). Among them, the R-tree [13] has been widely accepted as an efficient external tree structure for handling multidimensional data sets. Many dynamic R-tree variations have appeared in the literature (see for example [19, 7, 15, 9]) and they differ mainly in the heuristics used to split or merge nodes when node overflows or underflows occur. More relevant to the work reported here is the static case where the entire data set is known beforehand. Techniques that deal with this type of data are sometimes called tree-packing or bulk-loading. Various tree packing techniques aimed at improving node utilization and minimizing the *minimum bounding rectangles (MBRs)* of nodes have been introduced. They either sort data based on some spatial orders and recursively pack them into tree nodes level-by-level from bottom up (such as in [18, 14, 17]), or recursively partition data using various heuristics (such as in [23, 8]). Techniques for efficiently constructing dynamic R-trees for static data sets have also been explored. They view the construction of an R-tree for a static data set as a batched insertion problem and use lazy buffering strategy to achieve optimal I/O complexity [10, 5].

All the above techniques report the individual points that satisfy the range query. For raster data, one needs to find regions for which the attributes fall within the query window. Very few attempts have been made to address this problem. Most past work revolves around organizing the data objects hierarchically according to their spatial locations and summarizing their parameter values at different levels. STING [22] stores statistical information about the types and parameters of distributions for subsets of data in a hierarchical grid structure. This information can be used during a query to identify *relevant* cells that are later clustered using for example density based methods [11]. Yang et al. [24] addressed the same problem as in this paper but for a single parameter, and proposed a two level hierarchical structure that uses histograms to summarize the distributions of data values. The histograms of the high level cells are then clustered. The representative histogram summarized from histograms in the same cluster is used to decide whether that cluster of cells should be checked for a given query. These previous methods provide approximate answers without any guaranteed accuracy.

In the following three sections, we will discuss how our proposed data structure is used to solve this problem.

4 Data Structure

The proposed data structure, called the *Tree-of-Regions* (ToR) is an extension of the R-tree structure, although the same technique can be used to extend other tree structures. Each node of the R-tree defines a k-dimensional value range. For each node, we associate a set of regions such that the cells in each region have attributes that fall within the corresponding value range. Each leaf node of a ToR corresponds to a unique k-tuple from G and contains a pointer to the regions whose parameter values are equal to the k-tuple. Each internal node occupies an entire block and has $O(B)$ children. It contains a *minimum bounding rectangle (MBR)*, which is the tightest bounding rectangle of the union of the minimum bounding rectangles of its children. For leaf nodes, MBRs reduce to k-tuples. It is clear that the spatial region induced by the MBR of an internal node is the union of the spatial regions induced by its children. Figure 1 shows a ToR with $B = 3$ for a data set with two parameters. The top part of the figure illustrates the tree structure. Distinct k-tuples and their corresponding leaf nodes are depicted as dots. Internal nodes are represented by their MBRs shown as rectangles. The bottom part consists of four spatial regions $R(a)$, $R(b)$, $R(c)$, and $R(d)$ that are associated with leaf nodes a, b, c, and internal node d, respectively.

There are two benefits of storing regions at higher level nodes. First, during a range query, if the MBR of a node is covered entirely by the query window, the regions corresponding to that node can be reported immediately, with no need to explore the descendants of that node. Second, higher level nodes tend to have larger regions. By pre-storing these larger regions, we can compute the connected regions much more efficiently.

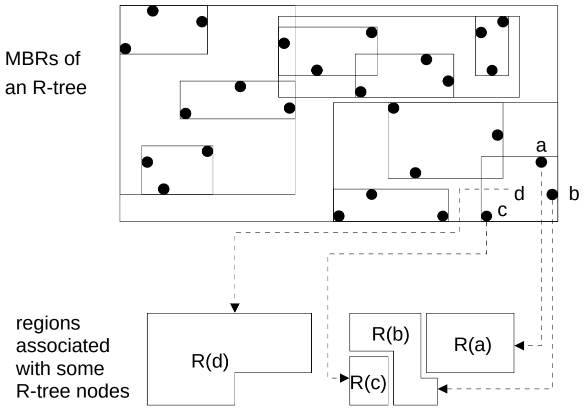


Figure 1: An example of the ToR

These potential benefits come at a storage cost as cells are duplicated along the path from the root to the leaf nodes. A *light* version of the ToR that associates regions only with the leaf nodes is another possible choice. However, it turns out that the query performance of the light ToR is much inferior without introducing any significant space savings. Related experimental results will be reported in the full paper.

A region associated with a ToR node is represented as a list of non-overlapping horizontal segments, each consisting of a maximal set of horizontally connected cells. Each segment is stored as a triple (y, x_left, x_right), where x_left and x_right are the x-coordinates of its leftmost and rightmost cells and y is its y-coordinate. Segments in the list are sorted using y as the primary key and x_left as the secondary key. Using segments to represent regions have several benefits. First, this representation maintains all boundary information of a region. Second, the amount of storage required is proportional to the perimeter of the region, which is much smaller than the area. Third, merging regions reduces to merging segments, which can be done quite efficiently as we will show soon. Figure 2 shows the merging of two regions represented using segment lists.

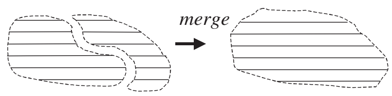


Figure 2: Merge of two regions represented by segment lists

Our overall data structure consists of three files: the *segment file*, the *leaf file*, and the *tree file*. Each file contains elements of the same size. The segment file contains the set of segment lists corresponding to the tree nodes. The list of segments corresponding to the same node are always stored contiguously on disk. The leaf file consists of the leaf nodes. The tree file is used to store the internal nodes. The reason we separate the leaf nodes from the internal nodes is that they have different structures. A leaf node does not contain the array of child pointers as does an internal node. Since we target our solution for very large data sets, we do not make the assumption that either of the three files will fit entirely in memory.

5 Tree Construction

The construction of a ToR consists of three steps:

1. The creation of the leaf nodes, where each leaf node corresponds to a distinct k-tuple of attribute values. This step includes the creation of all segments corresponding to the leaf nodes.

2. The creation of the internal nodes. Exactly how the internal nodes are constructed depends on the type of R-tree used. In this paper, we will use the packed Hilbert R-tree [14].[1]

3. The determination of the segment lists corresponding to the internal nodes.

5.1 Construction of Leaf Nodes

The objective of the first step is to find distinct k-tuples and their corresponding segment lists. These k-tuples form the leaf nodes of the tree structure. We assume that the raw data consists of a set of records, one for each cell. The record for cell (i, j) is in the form of $(f_1^{(i,j)}, f_2^{(i,j)}, \cdots, f_k^{(i,j)}, j, i)$. (Without causing confusion, we will call such record cell as well.) Reformatting is needed if the raw data are stored in a different format. Finding distinct k-tuples is achieved by sorting all cells using the key sequence $(f_1, f_2, \cdots, f_k, j, i)$. Since we are making the assumption that the data set resides on a disk, an external merge sort algorithm [21] is used. This sorting guarantees that cells having the same values are stored contiguously and, furthermore, horizontally adjacent cells that have the same values are also stored contiguously. This allows us to use a single scan through the sorted cells to create both the leaf nodes and the associated segment lists. Cells corresponding to the same k-tuple are merged into horizontal segments which are then stored contiguously in a *segment file*. Leaf nodes are created for distinct k-tuples in the same sorted order and stored in a *leaf file*. Each leaf node contains a k-tuple, an integer indicating the number of segments in the associated segment list, and a pointer to the beginning of that list in the segment file. Clearly, this step involves the external sorting of N cells whose I/O complexity is $O(N/B \log_{M/B} N/B)$.

5.2 Construction of Internal Nodes

The Hilbert R-tree packing algorithm packs as many children into a parent node as possible while trying to make sure that children of the same parent are spatially close by using the Hilbert "space-filling" curve. The tree is constructed from bottom up. N_0 leaf nodes (at level 0) are first sorted according to their ascending Hilbert values. The first B leaf nodes in the sorted list are removed from the list and grouped under the same parent node at level 1.

[1] There are three reasons for choosing the Hilbert R-tree. First, it is has been widely regarded as very competitive among all the R-tree variations [12]. Second, constructing such a tree structure can be done very efficiently, since it involves only one sort of the data set. Third, it has served as a base of performance comparison for many recently proposed data structures [17, 8, 5]. Note that the Hilbert R-tree can be replaced by any other static R-tree without affecting the remaining tree construction algorithm and the query algorithm.

The next B leaf nodes are again chosen and put under another parent node. This continues until there are no leaf nodes left. After all internal nodes at level 1 are created, they are grouped similarly into nodes at level 2. The only difference is that, the internal nodes are no longer sorted based on their Hilbert values. Instead, they are grouped according to the order in which they are created. Tree nodes thus are created level by level until there is only one node that becomes the root of the R-tree.

The complexity of this step is dominated by the Hilbert sorting of the leaf nodes, which requires $O(N_0/B \log_{M/B} N_0/B)$ I/O operations. N_0 normally is much smaller than N.

5.3 Creation of Internal Segment Lists

The creation of the segment lists for internal nodes is done by recursively merging the segment lists of their children, starting from the leaf level.

Note that the segments in each list have been sorted in increasing order using keys y and x_left, and stored contiguously on the disk. Merging horizontal segments can be done in a similar way as the merging phase of the external sorting, while combining horizontally adjacent segments. Segments in a list are always brought into memory in blocks. A buffer of size B is allocated for each list. (Note that we have at most B lists per node and $B^2 < M$.) The smallest segment among the first segments of all the lists is repeatedly removed and added to the output segment list until all lists become empty. During this process, whenever a buffer is empty, another block of segments in the corresponding list is retrieved from the disk. The output segments are also buffered and added to the segment file in blocks.

Suppose the total number of segments associated with the leaf nodes is S, then the creation of the segment lists for the internal nodes just above the leaf nodes requires $O(S/B)$ I/O operations. As a result, the I/O complexity of the entire process of creating internal segment lists could be $O(S/B \log_B N_0/B)$, which may seem to be worse than the external sorting. However, in practice both S and N_0 are much smaller that N. Furthermore, the number of segments often decreases rapidly as the tree level gets higher. As a result, this step is normally dominated by the previous two steps.

6 Range Queries

Given a query window w, an *allocation node* in the ToR is a node whose MBR is covered entirely by w and whose parent is not an allocation node. Figure 3 shows the allocation nodes, depicted as dashed rectangles for internal nodes and gray dots for leaf nodes, for the dotted window describing a range query.

Answering a range query consists of determining the set of allocation nodes followed by merging the segment lists of these allocation nodes horizontally. Finally, this list of segments is merged vertically to create the output regions.

6.1 Identifying Allocation Nodes

The search for the allocation nodes starts from the root with the set of allocation nodes initialized as empty. If a node has no intersection with w, then no action is taken. If a

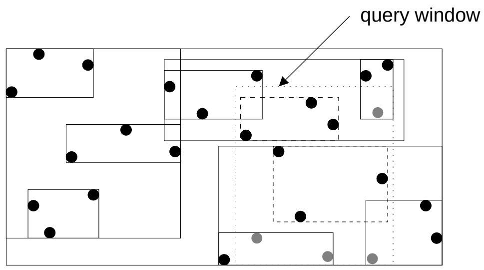


Figure 3: Allocation nodes

node is fully covered by w, then it is identified as an allocation node and added to the set of allocation nodes. Otherwise, if the node intersects w, the same procedure is repeated for each of its children.

Since in practice, many of the cells will share the same values, it is reasonable to assume that the segment list of each of the leaf nodes contains at least $B \log_B N_0$ cells. Under this assumption, the total number C of cells in the output regions is at least $fB \log_B N_0$, where f is the number of allocation nodes. Therefore, $f \le C/(B \log_B N_0)$. Clearly, only allocation nodes and their ancestors need to be accessed in this step. Since the height of the R-tree is $O(\log_B N_0)$, the total number of nodes accessed is $O(f \log_B N_0) = O(C/B)$.

6.2 Merging Segments Horizontally

After the allocation nodes are determined, their associated segments are merged so that horizontally connected segments are combined into a single segment. A segment list merging algorithm similar to the one used in the tree construction can be used here. There is one difference, however. Since the number of allocation nodes f could be larger than M/B, multiple iterations might be needed as follows. In each iteration, every M/B segment lists are merged into a single list. There will be $O(\log_{M/B} f)$ iterations. Let F be the total number of segments associated with the allocation nodes. The I/O complexity for each iteration is $O(F/B)$. The total complexity for the horizontal merge is then $O(F/B \log_{M/B} f)$. We denote the list of segments after the horizontal merge as L and its cardinality as T.

6.3 Merging Segments Vertically

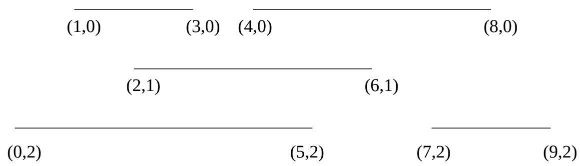
To identify the connected regions, we need to assign a label to each output cell such that cells from different connected regions have different labels. Using the sorted list L, finding connected regions can be done very efficiently, in fact in $O(T/B)$ I/O time.

Note that horizontally connected output segments have already been merged in the horizontal merge phase. What remains to be done is to merge the segments vertically to create regions.

We first use $O(T/B)$ I/O operations to scan L once to partition it into T_y sublists, T_y being the number of different y-coordinates of these segments. Each sublist contains segments with the same y-coordinate. This is possible since segments in L have already been sorted using their y-coordinates as the primary keys.

If L fits in internal memory, then we can apply an internal merging algorithm as follows. First, a graph H is created, whose vertices correspond to the segments in L. If two segments are adjacent to each other vertically, their corresponding vertices are connected by an edge in H. H is represented as a set of adjacency lists, one for each vertex. Second, a connected components algorithm based on depth-first search is used to find the regions.

Each sublist with y-coordinate y has one sublist *above* (*below*) it if there exists a sublist with the y-coordinate equal to $y-1$ $(y+1)$. To construct the adjacency list for the first segment s in a sublist L_s, we scan the sublists above and below until all segments vertically adjacent to s are found and the first such segments are recorded in s's adjacency list. Then we continue to scan the same two sublists for the next segment in L_s and keep doing so until the adjacency lists for all the segments in L_s are created. Figures 4(a) and (b) give a simple example of a segment list L consisting of five segments and its corresponding adjacency list. Given a segment, its adjacent segments can be found by scanning the corresponding *above* and *below* sublists, starting from the segments recorded in its adjacency list.



(a) A list of sorted segments

	segment	*above*	*below*
0	$(0,1,3)$	NULL	2
1	$(0,4,8)$	NULL	2
2	$(1,2,6)$	0	3
3	$(2,0,5)$	2	NULL
4	$(2,7,9)$	NULL	NULL

(b) The corresponding adjacency lists

Figure 4: Adjacency lists

If L does not fit in internal memory, we determine the connected regions as follows. We read as many sublists as the internal memory size allows, starting from the one with the smallest y value. We call this set of sublists a *stripe*. The internal merging algorithm described above is then applied to label all the segments in that stripe. These labeled segments are then written back to disk. Next, we again read as many sublists as possible, but starting from the lowest sublist in the last stripe (we call this sublist the *lower boundary* of the last stripe and the *upper boundary* of the current stripe). Since the segments in that boundary (the *boundary segments*) have already been labeled, their labels are propagated to other segments connected to them. New labels will be assigned to segments that do not connect with any of these boundary segments. If, during the labeling process, we find out that two segments from the upper boundary with different labels are actually connected then the label of one of them is changed. This change is kept in an *label-update table (LUT)* for that boundary. LUT is also written to disk after the current stripe is processed. The same process continues as we read the sublists stripe by stripe with two contiguous stripes sharing a boundary until all segments are labeled.

After the downward labeling process, an upward updating operation is performed as follows. We repeatedly read a stripe and the LUTs of its upper and lower boundaries, starting from the stripe that is just above the lowest stripe. For each stripe, we update the labels of the segments in it using the label changes maintained in the lower LUT. These changes are also used to update the upper LUT. This process continues until the labels of the segments in the first stripe are updated. Details will appear in the full paper.

Under the reasonable assumption that the size of each of the T_y sublists is less than $O(M)$, we can make sure that a stripe and its upper and lower LUTs can be loaded into memory simultaneously, thus guaranteeing that the operations described above are possible. It is obvious that both the downward labeling and the upward updating processes require $O(T/B)$ I/Os for reading and writing the segments. The additional cost for reading and writing the LUTs is clearly less than $O(T/B)$ because each LUT is only accessed O(1) times and the total size of the LUTs is less than the total size of the boundaries, which is less than T.

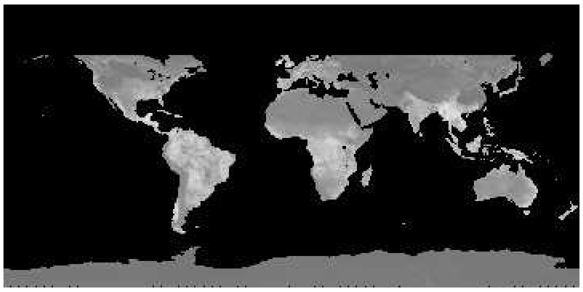
7 Experimental Results

We tested our new approach on a number of raster data sets generated from satellite data. We describe here two types, a global coarse resolution and a fine resolution. The first type consists of the standard AVHRR (Advanced Very High Resolution Radiometers) data products that form a 1-degree by 1-degree of global coverage generated from 10 day composites. Geophysical parameters contained in the data set include: Normalized Difference Vegetation Index (NDVI), two reflectance channels (channels 1 and 2), three brightness temperature channels (channels 3, 4, and 5), and date and hour of observation [2]. NDVI is the ratio of the contrast between the response of the two reflectance channels. We used three of these parameters (NDVI, channel 1 and channel 4). The total number of cells in each AVHRR data set is 64K. The second type is the TM (Thematic Mapping) data [1]. Each TM data is a 7200-by-8192 grid representing a region with 30 meter resolution. Each cell has seven parameters (bands), of which we used five (bands 1, 2, 4, 5, and 7) on a 1000×1000 grid. A total of 44 data sets are used in our experiments. 24 of them are AVHRR global 1-degree by 1-degree data and the remaining 20 are the TM data.

The tree construction and query answering algorithms were coded in C. All the experiments were conducted on a Pentium III 550Mz machine with 1GB Memory running Linux 2.2.19. The page size B was chosen to be 8192 bytes.

7.1 Sample Query Results

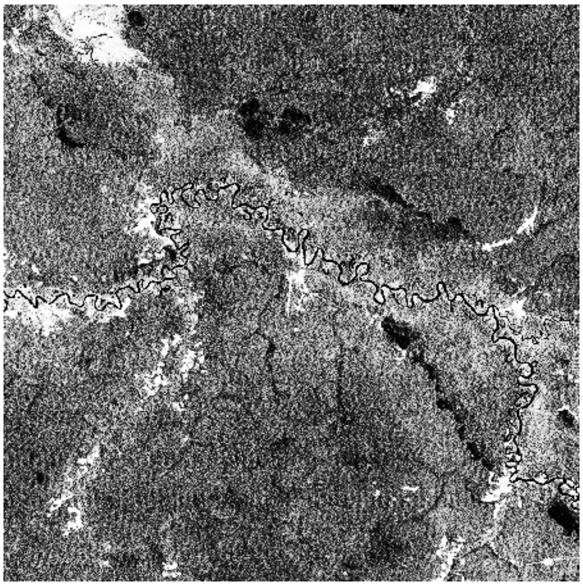
Figures 5 and 6 give two sample query results. Figure 5(a) is the global 1-degree by 1-degree NDVI map. Figure 5(b) shows the areas with high temperature and high NDVI values, which approximately correspond to the rain forests and the wooded grasslands that mainly locate in Central America, Central Africa, South Asia, and the east coast of Australia. Different colors are used to denote different connected regions. Figure 6(a) is band 7 for part of a TM scene in Columbia. Figure 6(b) shows the query result that largely corresponds to *nonforests*, which typically have high values in bands 4 and 7.



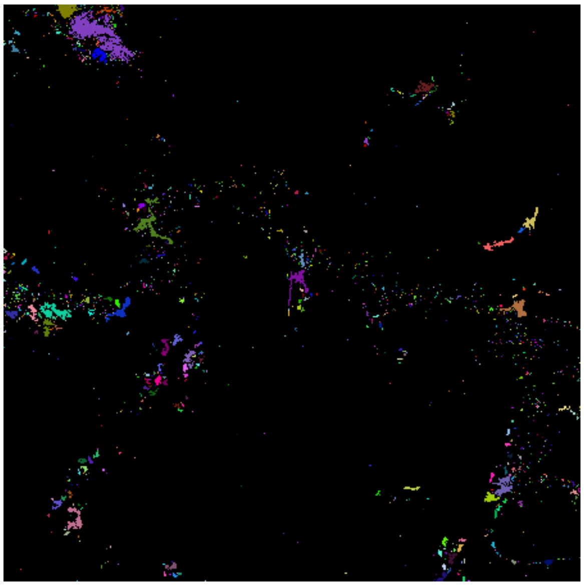
(a) NDVI values, 10-day composite
Jan. 01, 1989

(b) regions where NDVI ≥ 0.4 and channel 3 brightness temperature ≥ 260.0 *Kelvin*

Figure 5: Sample query results (AVHRR)



(a) Band 7, Part of Columbia (Path/Row: 6/66)
Oct 16, 1996



(b) regions where
band 4 ≥ 20 and band 7 ≥ 20

Figure 6: Sample query results (TM)

7.2 Query Performance

We will first examine the overall query performance and then focus on the main components of the query algorithm. We will also compare the number of output segments and output cells to demonstrate the importance of representing regions using segment lists.

For each ToR, we generated query windows of 5 different sizes ranging from 5% to 25% of the size of the MBR of the root node. For each window size, 30 query windows were randomly and uniformly generated within the root MBR. All performance numbers are averaged over these queries. We will report the experimental results for the TM data, which is much larger than the AVHRR data.

Figure 7 shows the overall query execution time as contributed by the three main steps. We can see that the amount of time it takes to identify the allocation nodes is very small comparing to the horizontal and vertical segment merge times. The horizontal merge step takes up most of the execution time, while the vertical merge step was done much faster. Overall, it can be seen that the queries are handled extremely fast, within 5 seconds for the TM data even for queries with large windows. Furthermore, the query time is proportional to the output size as had been indicated by our earlier analysis.

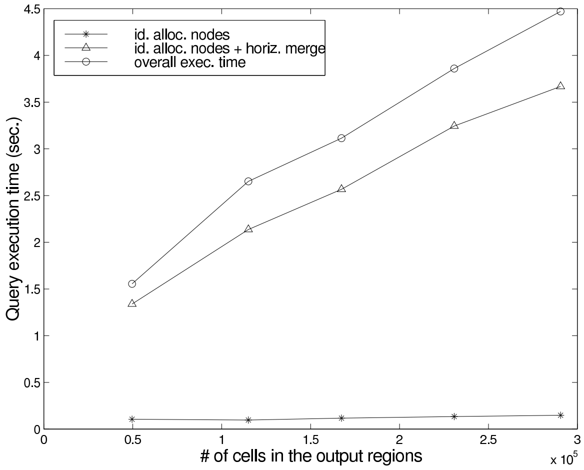
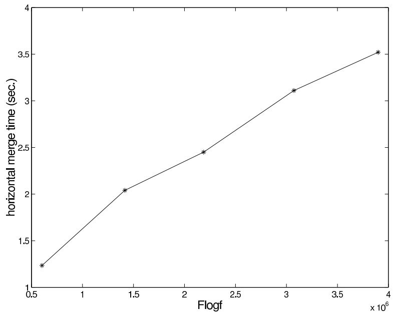
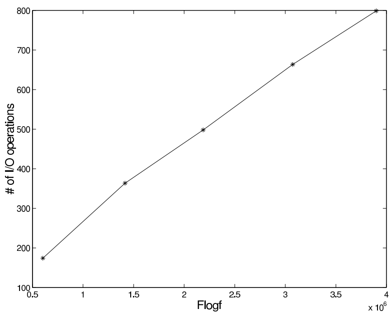


Figure 7: Overall query performance

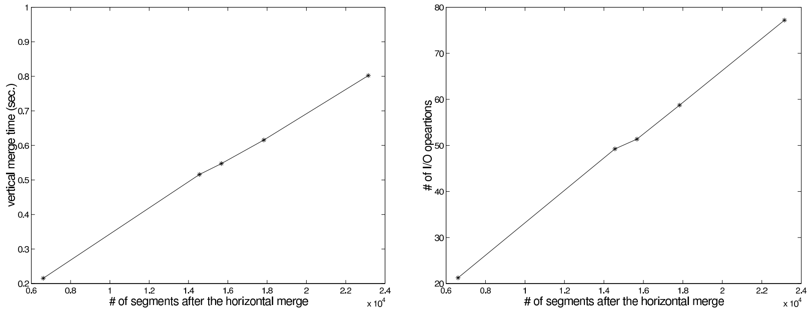




(a) execution time (b) I/O complexity

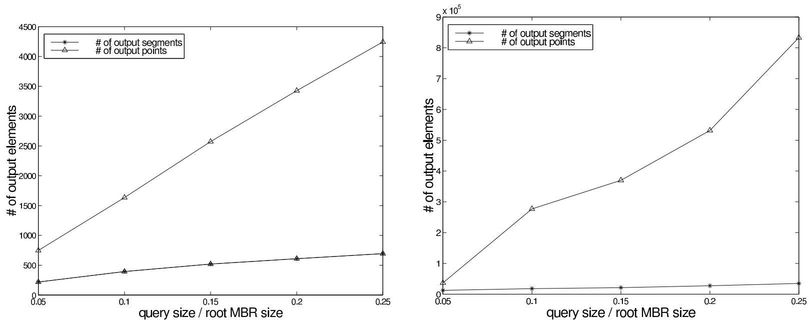
Figure 8: Complexity of Horizontal Merge

Figures 8(a) and (b) show the comparison of the theoretical bounds and the observed bounds in terms of execution time and number I/O operations. The X-axis represents the theoretical complexity $F/B \log_{M/B} f$, where f is the number of allocation nodes and F is the number of segments associated with them. Since B and M do not change in our experiments, this theoretical bound differs from $F \log_2 f$ by only a constant. Thus, using the latter will not affect the shape of the curves. These two figures demonstrate that our experimental results and the theoretical results are quite consistent. The performance of the



(a) execution time (b) I/O complexity

Figure 9: Complexity of Vertical Merge



(a) AVHRR data (b) TM data

Figure 10: Number of output segments v.s. number of output cells

vertical segment merge is shown in Figure 9. The results are consistent with our theoretical complexity analysis as well.

Finally, we compare the number of output cells and output segments to demonstrate the effectiveness of the segment representation of output regions. Figure 10 shows the average number of output segments and output cells for both AVHRR and TM data. Note that while the number of output cells increases quite fast as the size of the query window increases, the number of output segments increases very slowly in both cases. This has enabled the various steps in our query answering algorithm to be carried out quite fast.

References

[1] Landsat thematic mapper data. http://edc.usgs.gov/glis/hyper/guide/landsat_tm.

[2] Goddard DAAC NOAA/NASA Pathfinder AVHRR Land (PAL). http://daac.gsfc.nasa.gov/REFERENCE_DOCS/dataset_references/pal-_summary.html, 1999.

[3] Content-based search and data mining. http://www.esipfed.net/clusters/content_based/sci_scen.html, 2000.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[5] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, pages 328–348, Baltimore, MD, Jan. 1999.

[6] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 346–357, Philadelphia, PA, May 1999.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[8] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Proc. 6th Int. Conf. Extending Database Technology, EDBT*, pages 216–230, Mar. 1998.

[9] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *VLDB'96, Proceedings of 22nd International Conference on Very Large Data Bases*, pages 28–39, Mumbai (Bombay), India, Sept. 1996.

[10] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 406–415, 1997.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231, Portland, OR, Aug. 1996.

[12] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

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

[14] I. Kamel and C. Faloutsos. On packing R-trees. In *In Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[15] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 500–509, Santiago, Chile, 1994.

[16] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Science*, 52(3):589–612, 1996.

[17] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 497–507, Apr. 1997.

[18] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 17–31, Austin, TX, Dec. 1985.

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

[20] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.

[21] J. S. Vitter. External memory algorithms. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems*, pages 119–128, New York, NY, USA, June 1998.

[22] W. Wang, J. Yang, and R. Muntz. STING: a statistical information grid approach to spatial data mining. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 186–195, Athens, Greece, Aug. 1997.

[23] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, CA, 1996.

[24] R. Yang, K.-S. Yang, M. Kafatos, and X. Wang. Value range queries on earth science data via histogram clustering. In *First International Workshop on Temporal, Spatial, and Spatio-Temporal Data Mining*, pages 62–76, Lyon, France, Sept. 2001.