Improved Searching for Spatial Features in Spatio-Temporal Data*

Kurt Stockinger and Kesheng Wu

Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

September 27, 2004

Abstract

Scientific data analysis often requires mining large databases or data warehouses to find features in space. One important task is to find regions of interest such as stellar objects in astrophysics or flame fronts in combustion studies. Typically, this task is performed in two steps. The first step (searching) identifies records satisfying certain conditions specified by the user and outputs a set of cells. The second step (region-growing) groups these cells into connected regions. Most common approaches essentially perform a brute-force scan for the searching step. A number of indexing schemes have been proposed to speed up the searching step. Because they usually also slow down the region-growing step, these schemes have not reduced the overall time.

In this article, we propose an approach based on compressed bitmap indices. Our approach speeds up not only the searching step, but also the region-growing step. In the literature, the time complexity of the region-growing step is demonstrated to be linear in the number of records in the dataset. In our tests, we show that the response time of our region-growing algorithm is linear in the number of records close to the surface of the regions of interest which is a small subset of all cells.

*The authors thank Ekow Otoo, Doron Rotem, and Heinz Stockinger for their constructive comments during the writing of this article. This work was supported by the Director, Office of Science, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

1 Introduction

Many scientific datasets are spatio-temporal in nature because they measure physical quantities in space and time. For example, a simulation of the combustion process computes the concentrations of all chemical species along with pressure and temperature [3, 6]. A satellite image of the surface of the Earth [9, 10] measures quantities such as temperature, wind speed, humidity and so on. One common operation in mining these datasets is to derive some quantities on regions of interest, e.g., the total heat output from an ignition kernel in the combustion simulation, and the average rain fall of a state over a specified period of time. To support this operation one needs to efficiently identify regions of interest.

In this paper, we concentrate on datasets with regular discretization of space, such as the Direct Numerical Simulations of combustion on uniform 2D or 3D meshes[3, 6], and the raster images of the surface of the Earth from satellite observation [9, 10]. In these cases, the space is discretized into small cells, and the quantities on each cell are computed or measured at some time values. All data from one time value is commonly known as a time step. To simplify the discussion, we only discuss the regions of interest are defined by users with range conditions such as "pressure $> 10,000$ and temperature $> 1,000$."

After the user specifies the conditions, the process of identifying the regions of interest is usually divided into two steps, the *searching step* to find the cells satisfying the conditions, and the *region-growing step* to group the cells into connected regions. This region-growing step is often called the connected component labeling in image processing literature [1, 4]. If the conditions for regions of interest are simple, such as "pressure $< 20,000$", the boundaries of the regions are the contour lines computed by the iso-contouring algorithms for "pressure $= 20,000$" [2, 8]. For visualization purposes, region-growing algorithms produce the same output as iso-contouring algorithms. Because of this, region-growing algorithms are sometimes compared with iso-contouring algorithms. However, one important difference is that region-growing algorithms produce the cells inside the regions but the iso-contouring algorithms only identify the cells on the boundaries.

One approach to identify regions of interest is to partition the cells according to spatial coordinates, such as Quad-tree and R-Tree [5]. These indexing schemes are only efficient for relatively low dimensional data, say the total number of attributes including the spatial dimensions is less than 10. If the number of attributes is more or if only a small number of attributes are involved in the conditions, these indices are not as efficient as the brute-force scan. Worse yet, these indexing schemes usually separate cells that are neighbors in space. Because of this, they slow down the region-growing step. For example, in a recent study by Shi and Jaja [9], the time spent by their region-growing step is significantly longer than the time spent by comparable connected component labeling algorithms [1, 4]. For these reasons, the most successful approach uses brute-force scan for the searching step. Because this allows the neighboring cells to be kept together, efficient algorithms for connect component labeling can be used for the region-growing step. Clearly, if we can use an indexing scheme that keep the neighboring cells together, we should reduce the time used by the searching step without increasing the time used by the region-growing step. This would reduce the total time.

We propose to use bitmap indices for the searching step because they do not reorder the cells [7, 11, 14]. Our implementation of the bitmap indices also uses compression scheme based the run-length encoding. In this case, the searching step produces a compressed bitmap to represent the cells satisfying the conditions. This compressed bitmap can be easily converted to blocks of connected cells. Working with these blocks to perform the region-grow step is in fact more efficient than using cells directly. Previously, we have presented some evidence that the bitmap-based approach works well for data a on 2D uniform mesh [12]. In this paper, we describe a set of efficient algorithms for region-growing in 3D. By taking full advantage of the compact output from the bitmap indices, our algorithms are orders of magnitudes faster than the generic algorithm required when the searching step produces cells in arbitrary order. As in the 2D case [12], we observe that our region-growing algorithm scales linearly in the number of blocks produced by the searching step. Since the number of blocks is less than the number of cells on the boundaries of the regions of interest, our algorithm scales better than the best iso-contouring algorithms [2, 8]. The best of connected component labeling algorithms scale linearly in the total number of cells [1, 4], our region-growing algorithm scales much better since the number of blocks are much less than the total number of cells.

RID	**I**	bitmap index			
		=0	=1	=2	=3
1	0	1	0	0	0
2	1	0	1	0	0
3	3	0	0	0	1
4	2	0	0	1	0
5	3	0	0	0	1
6	3	0	0	0	1
7	1	0	1	0	0
8	3	0	0	0	1
		b_1	b_2	b_3	b_4

Figure 1: A sample bitmap index where RID is the record ID and **I** is the integer attribute with values in the range of 0 to 3.

2 Bitmap-Based Approach

For read-only or read-mostly data, the bitmap index is one of the most efficient indexing schemes for speeding up range queries [7, 14]. For an attribute with c distinct values, the basic bitmap index generates c bitmaps each with N bits, where N is the number of records (cells) in the dataset. Each bit in a bitmap is set to 1 if the attribute in the record is of a specific value, otherwise 0. For example, the integer attribute **I** shown in Figure 1 can be one of four distinct values, 0, 1, 2, and 3, and the corresponding bitmap index has four bitmaps. Since the value in record 5 is 3, the fifth bit in b_4 is set to 1 and the same bits in other bitmaps are 0. In short, 4 bitmaps are required to encode 4 distinct attribute values.

Using a bitmap index, answering a range query, such as $\mathbf{I} < 2$, requires some bitwise logical operations on the bitmaps. Since bitwise logical operations are well-supported by computer hardware, we can expect to answer range queries efficiently with bitmap indices. With a bitmap index for each attribute, conditions involving multiple attributes, such as "$\mathbf{I} < 2$ and $\mathbf{J} < 3$", can also be efficiently answered by combining the partial solutions computed using indices on attributes **I** and **J**.

One major concern about the bitmap index is that for attributes with a large number of distinct values, the indices require too much space to store. Recently, it has been shown that even in the worst case, the bitmap indices can be compressed to a size that is comparable with a typical B-tree index. The time required to answer a range query using a compressed bitmap index is in fact optimal. In the worst case, the response time is proportional to the number of hits of the query [14].

In our implementation of the bitmap index for spatio-temporal data, we preserve the spatial order of the cells. This avoids reordering of the raw data and reduces the time required for building the bitmap indices. Another benefit is that the compressed bitmap produced as the result of the searching step can be easily turned into blocks of connected cells. In [9] the authors report that a majority of the compute time is spent in grouping the cells identified by the searching step into horizontal line segments. The time required in our approach to convert a compressed bitmap into line segments and other blocks is usually below the accuracy of the common timing functions of about 0.01 seconds.

On a 3D regular mesh, the cells can be identified by three coordinates along three spatial dimensions, x, y, and z. A common way of representing the cells during computation or measurement is to order the cells according to their z-coordinates first. For cells with the same z-coordinates, order them according to y-coordinates. For cells with the same z- and y-coordinates, order them according to x-coordinates. This is usually called the raster scan order. Since most of the efficient bitmap compression schemes are based on run-length encoding, the result produced by the searching step naturally represents consecutive cells that either all satisfy the user specified conditions or not. These consecutive cells can be easily mapped to blocks along the x-axis. There are three types of blocks (see Figure 2):

1. Line segments, where all cells have the same y and z coordinates.

2. Whole lines, where all cells have the same z coordinates, consecutive y coordinates and all possible values in the x-axis.

3. Whole planes, where cells have consecutive z coordinates and have all possible x and y coordinates in the given range of z coordinates.

To simplify the discussion, we describe all algorithms as if there are only line segments. In the actual implementations all three types are considered.

Both experiments and analyses have shown that the time spent in the searching step is proportional to the number of blocks identified, and the size of the result produced by the searching step is also proportional to the number of blocks [14, 12, 13]. The most efficient iso-contouring algorithms are shown to have an execution time that is proportional to the number of cells touching the contour lines [2, 8]. Since each block has at least one point touching the boundaries of the regions, the searching time using bitmap indices is not

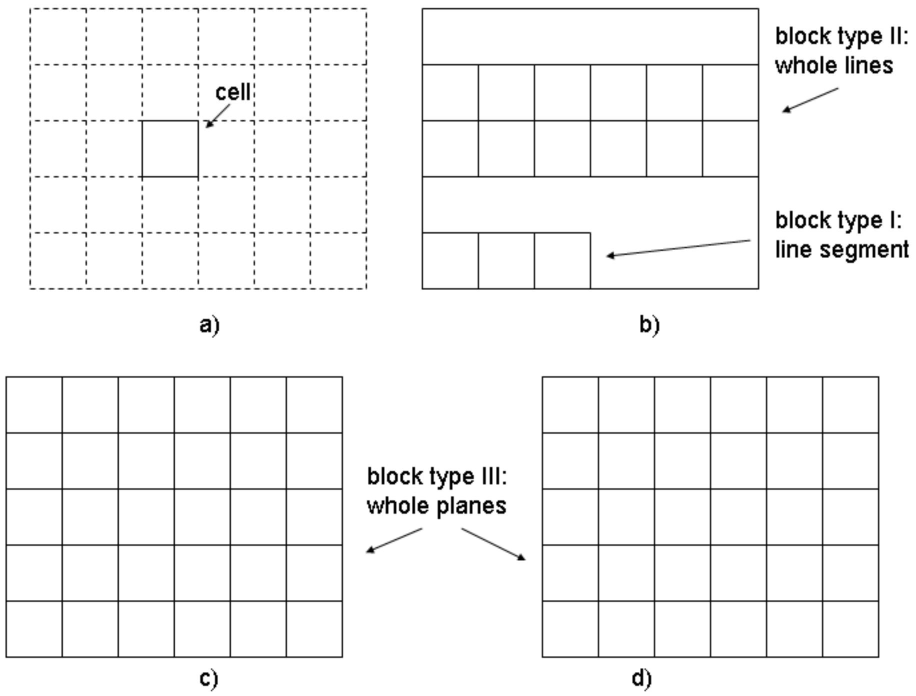

Figure 2: a) Cell with grid lines of the mesh. b) Block type I and II without grid lines. c)-d) Block type III without grid lines.

worse than linear in the number of cells touching the contour lines. On 2D data, we have observed that the region-growing time is proportional to the number of blocks [12]. Next we will show the same is true for 3D data. If both steps have linear complexity, then the whole process has linear time complexity. This demonstrates that our bitmap-based approach to identify regions of interest is theoretically optimal.

3 Region-Growing Algorithms

Each block in the 3D space is characterized by a pair of points for each dimension. Each pair specifies the lower and upper bound in a given dimension (bounding box). For instance, $block[7]$ in Figure 3 is characterized by the points $< 0, 1 >< 3, 4 >< 1, 2 >$, where $< 0, 1 >$ refers to the coordinates $< x_{min}, x_{max} >$, $< 3, 4 >$ refers to $< y_{min}, y_{max} >$, and $< 1, 2 >$ to $< z_{min}, z_{max} >$. In the following discussion, we take the input for region-growing algorithms to be a list of blocks. The output of the region-growing algorithm are lists of connected regions. Next, we outline algorithms for connecting blocks to form regions of interest.

3.1 Simple Region Search

This algorithm is the most generic one and does not assume any particular order of blocks. Note that blocks are the output of the search step discussed in Section 2. This algorithm is used for indexing schemes that do not preserve spatial ordering among the cells, for example, [9].

The search algorithm works as follows. The first block *block*[0] is considered as the first connected region *connectedRegions*[0]. Next, the algorithm loops over each block and checks whether it is part of an already identified connected region. If it is not part of an existing connected region, a new region is created. If the block is part of a connected region, it gets added to this one. If the block is part of several connected regions, these regions are merged.

```
numberOfBlocks is given from initial search step   // always greater than 0
connectedRegions = 0
connectedRegions[0]->add(block[0])  // vector of vectors with block indices;
                                    // each block-vector holds the indices
for i = 1 to numberOfBlocks-1       // of the blocks of a specific region
  mergeRegions.clear()
  numberOfFoundRegions = 0
  findMore = true
  for j = 0 to connectedRegions.size()-1
    for k=0 to connectedRegions[j].size()-1
        AND (searchMore == true)
      if block[i] connected with connectedRegions[j][k] then
        numberOfFoundRegions++
        connectedRegions[j]->add(block[i])
        mergeRegions->add(j)
        searchMore = false
  if numberOfFoundRegions == 0 then
    createNewRegion(block[i])         // connectedRegions->add(block[i])
  else if numberFoundRegions > 1 then
       mergeConnectedRegions in mergeRegions
```

For indexing schemes that do not preserve spatial order among the cells, it is necessary to use this algorithm. Another possibility is to sort the blocks first and then use one of following algorithms.

3.2 Improved Region Search I

The improved algorithm *FastRegionSearch I* takes advantage of the fact that the blocks are partially sorted. Rather than searching through all connected regions, only a subset has to be searched, namely those regions that are close to the block. In this case, close means that the respective block is on the same plane within the axis of our coordinate system. See Figure 3.

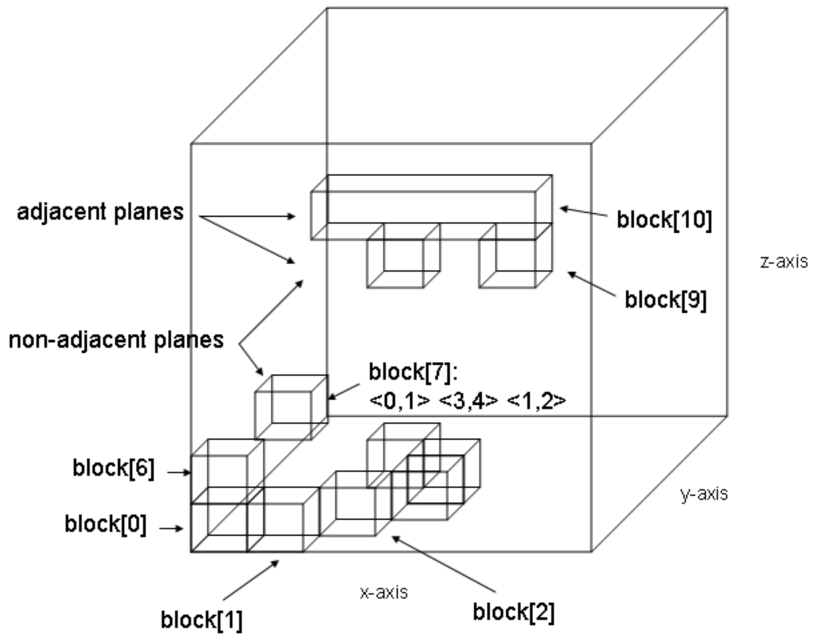

Figure 3: Blocks in space with 3 spatial dimensions. Note: For simplicity, not all blocks are indexed in this figure.

FastRegionSearch I iterates through the blocks plane-wise with respect to the z-axis. The algorithm first checks all the blocks on the x-y-plane for a given z-coordinate (see Figure 4 (a)). Once all blocks on this plane are searched, the next x-y-plane gets searched ((see Figure 4 (b) - (d)).

Note that some blocks might not be considered as connected on the x-y-plane for a given z-value. However, by searching x-y-planes on a higher z-value, these blocks can get connected. This is true for the blocks $block[8]$ and $block[9]$ in Figure 3. They are only identified as being connected after $block[10]$ is processed.

Like for the simple algorithm, the first block is considered to be the first connected region. Next *FastRegionSearch I* searches through the blocks on the x-y-plane with z-coordinate 0 and checks for connected regions in the same way as the simple algorithm does. All blocks on this plane are stored in an auxiliary data structure called *candidateRegions*. This is a two-dimensional data structure. Its first dimension

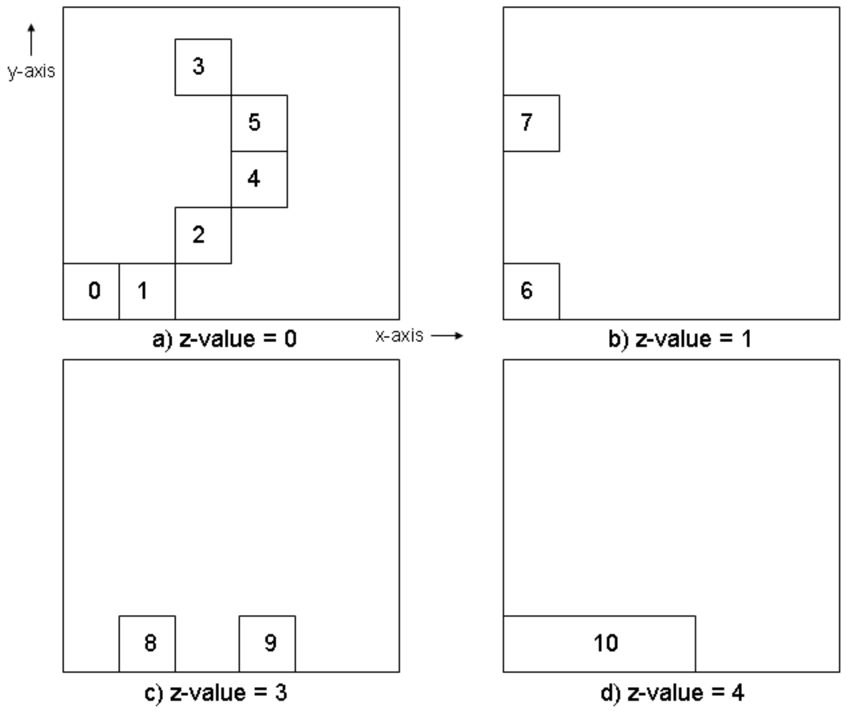

Figure 4: Blocks of Figure 3 mapped to two spatial dimensions.

holds the index of the original block. The second dimension holds the index of the connected region, e.g. $candidateRegions[1][0] = 7$ refers to block 7, $candidateRegions[1][1] = 1$ being part of connected region 1 (see Figure 3).

Once all blocks on the x-y-plane with the z-coordinate 0 are checked, the next plane on a higher z-coordinate is searched. If the next block is on a plane adjacent to the previous plane, i.e. the difference of the z-coordinate is 1, then we know that there are candidate regions that might be connected to previous candidates regions. The first step is to rename all candidate regions of the previous plane to the auxiliary data structure *activeRegions* and clear *candidateRegions*. *activeRegions* is the same two-dimensional data structure as *candidateRegions*. Assume we are searching through the x-y-plane with the z-coordinate = 1 (see Figure 4 (b)). All searched blocks on this plane are considered as candidate regions. On the other hand, all blocks on the previous x-y-plane with z-coordinate = 0 are considered as active regions.

Next, all blocks on the current plane are checked if they are connected with the regions in *activeRegions*. The checking for connection and the possible merging of connected regions is analogous to what we discussed for the simple algorithm.

If the blocks are on a x-y-plane that is not adjacent to the previous one, i.e. the difference in the z-coordinate > 1, then we know that these regions are not connected with any regions on the previous plane, e.g. *block*[8] and *block*[9]. If the block is the first on the non-adjacent x-y-plane, then a new connected region is created, otherwise the block is compared with all the blocks on the same plane, i.e. *candidateRegions*.

```
numberOfBlocks is given from initial search step   // always greater than 0
connectedRegions = 0
connectedRegions[0]->add(block[0])

for i = 1 to numberOfBlocks-1
  mergeRegions.clear()
  numberOfFoundRegions = 0
  if (block[i] is on the same x-y-plane) then
    findConnections() // see below
  else if (block[i] is on the adjacent x-y-plane) then
    activeRegions = candidateRegions
    candidateRegions.clear()
    findConnections() // see below
  else if (block[i] is first of non-adjacent x-y-plane) then
    candidateRegions.clear()
    createNewRegion(block[i])

findConnections():
  numberOfConnections = 0
  for j=0 to candidateRegions.size()-1
    if block[i] is connected with candidateRegions[j] then
      numberOfFoundRegions++
      if (numberOfFoundRegions == 1) then
        connectedRegions[candidateRegions[j]]->add(block[i])
      mergeRegions->add(j)

  for j=0 to activeRegions.size()-1
    if block[i] is connected with activeRegions[j] then
      numberOfFoundRegions++
      if (numberOfFoundRegion == 1) then
        connectedRegions[candidateRegions[j]]->add(block[i])
      mergeRegions->add(j)

  if (numberOfFoundRegions == 0) then
    createNewRegion(block[i])
  else if (numberOfFoundRegions > 1) then
    mergeConnectedRegions in mergeRegions
    update region index in candidateRegions and activeRegions
  candidateRegions->add(block[i])
```

3.3 Improved Region Search II

Next we study an even further improved algorithm called *FastRegionSearch II*. The main difference is that we group the regions *activeRegions* and *candidateRegions* into connected regions. Recall that *candidateRegions* refers to all regions on the same plane as the new block. By grouping them into connected regions, not all blocks of a particular region have to be searched for finding possible connected regions. The search can be stopped after identifying the first one. However, the search through *activeRegions* involves more steps. Since an input block can be connected with multiple *activeRegions*, all subgroups need to be searched. Both data structures are one-dimensional. The first element holds the index of the connected region. The remaining elements hold the index of the original block. Assume the z-coordinate is 4 (see Figure 4 (d)). In this case the two blocks *block*[8] and *block*[9] of the previous plane (see Figure 4 (c)) belong to the following active region: $activeRegions[0][0] = 2$ (index of connected region), $activeRegions[0][1] = 8$ (*block*[8]) and $activeRegions[0][2] = 9$ (*block*[9]).

Due to the one-dimensional structure of *activeRegions* and *candidateRegions* *FastRegionSearch II* has another advantage over *FastRegionSearch I*. Only the first element needs to be updated for changing the index of the connected region after regions get merged.

The first part of this algorithm is identical to *FastRegionSearch I*. The difference, however, is the method *findConnections* that is described below:

```
findConnections():
  numberOfConnections = 0
  for j=0 to candidateRegions.size()-1 AND searchMore
    if block[i] is connected with candidateRegions[j] then
      numberOfFoundRegions++
      searchMore = false                                  // Search stops after one
      connectedRegions[candidateRegions[j]]->add(block[i]) // connected region is found
      candidateRegions[j]->add(block[i])                  // in candRegions.
      mergeRegions->add(candidateRegions)

  for j=0 to activeRegions.size()-1
    if block[i] is connected with activeRegions[j] then
      numberOfFoundRegions++                              // Since a block can be connected
      if (numberOfFoundRegion == 1) then                  // with several regions, all
        candidateRegions[activeRegions[j]]->add(block[i])  // connected regions in
        connectedRegions[activeRegions[j]]->add(block[i])  // activeRegions need to be
      mergeRegions->add(candidateRegions)                 // searched.

  if (numberOfFoundRegions == 0) then
```

```
    createNewRegion(block[i])
  else if (numberOfFoundRegions > 1) then
    mergeConnectedRegions in mergeRegions
    update region index in candidateRegions and activeRegions
```

3.4 Improved Region Search III

A further improvement to the algorithm is not to search through all candidate regions and through all regions on the active plane, but only through the ones in close proximity to the block. *FastRegionSearch III* searches through all connected regions on the same y-plane and one y-plane below the block. In addition, it searches through the adjacent z-plane with regions adjacent to the y-plane of the block. Assume the block is *block*[6] on the x-y-plane with the z-coordinate = 1 (see Figure 5 (b)). The algorithms introduced in the previous sections would search through all regions of the x-y-plane with the z-coordinate = 0. However, *FastRegionSearch III* only searches through the two columns with the y-values y = $y_{block[7]}$ and y = $y_{block[7]}$ + 1.

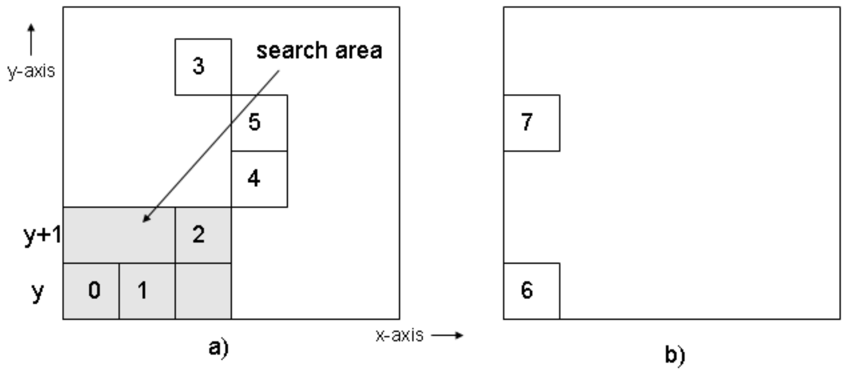

Figure 5: Reduced search area for *FastRegionSearch III*.

In order to retrieve only those blocks on a given y-axis, we introduce a new hash-based data structure which keeps track of the y-values of the connected regions. The data structure has the following format:

```
<y-value>;
<indices of connected regions 1 to n>;
<indices of blocks of region 1>
<indices of blocks of region 2> ...
<indices of blocks of region n>
```

A typical example would be:

```
<4>; <1, 2, 4>; <2, 3> <4> <8, 9, 10>
```

The first column refers to the connected regions with the y-value 4. This value is also the hash-value for fast lookup. The second column holds the index of the connected regions 1, 2 and 4. This means, that the blocks 2 and 3 make up region 1. Block 4 corresponds to region 2. Finally, the blocks 8, 9 and 10 refer to the connected region 4. The advantage of this data structure is that the lookup for connected regions with a given y-value is of complexity O(1). In addition, if two connected regions get merged, say regions 2 and 4, then this can easily be indicated by updating the corresponding entries of the second column. Assuming that region 4 gets merged into region 2, the updated column would be <1, 2, 2>.

Due to space limitations we do not state all the details of the whole algorithm here. The main difference to *Regions Search II* is that rather than searching through all candidate or active regions, only those are checked, that are adjacent to the y-plane of the block. For instance, if the y-value of the block is 4, then all regions are searched on the y-planes 3, 4 and 5.

4 Experimental Results

We tested our algorithms on astrophysics data which consists of 110 million records. We selected three attributes from our data set and performed one-dimensional queries with various selectivities ranging from 5% to 95%. The experiments were carried out on a 2.8 GHz Intel Pentium IV with 1 GB RAM. The I/O subsystem is a hardware RAID with two SCSI disks.

Figure 6 depicts the time for finding connected regions as a function of the blocks that vary between 38,000 and 400,000. The processing time for the simple algorithm takes 12 to 1,250 seconds. The algorithm *FastRegionSearch I* is up to a factor of 100 faster. One the other hand, *FastRegionSearch II* is again up to a factor of 5 faster than *FastRegionSearch I* or even up to factor of 500 faster than the simple algorithm. For a low number of blocks, *FastRegionSearch III* is slower than *FastRegionSearch II*. However, for a large number of blocks, *FastRegionSearch III* shows the best overall performance.

Figure 7 depicts the average time for the nine region search queries shown Figure 6. For instance,

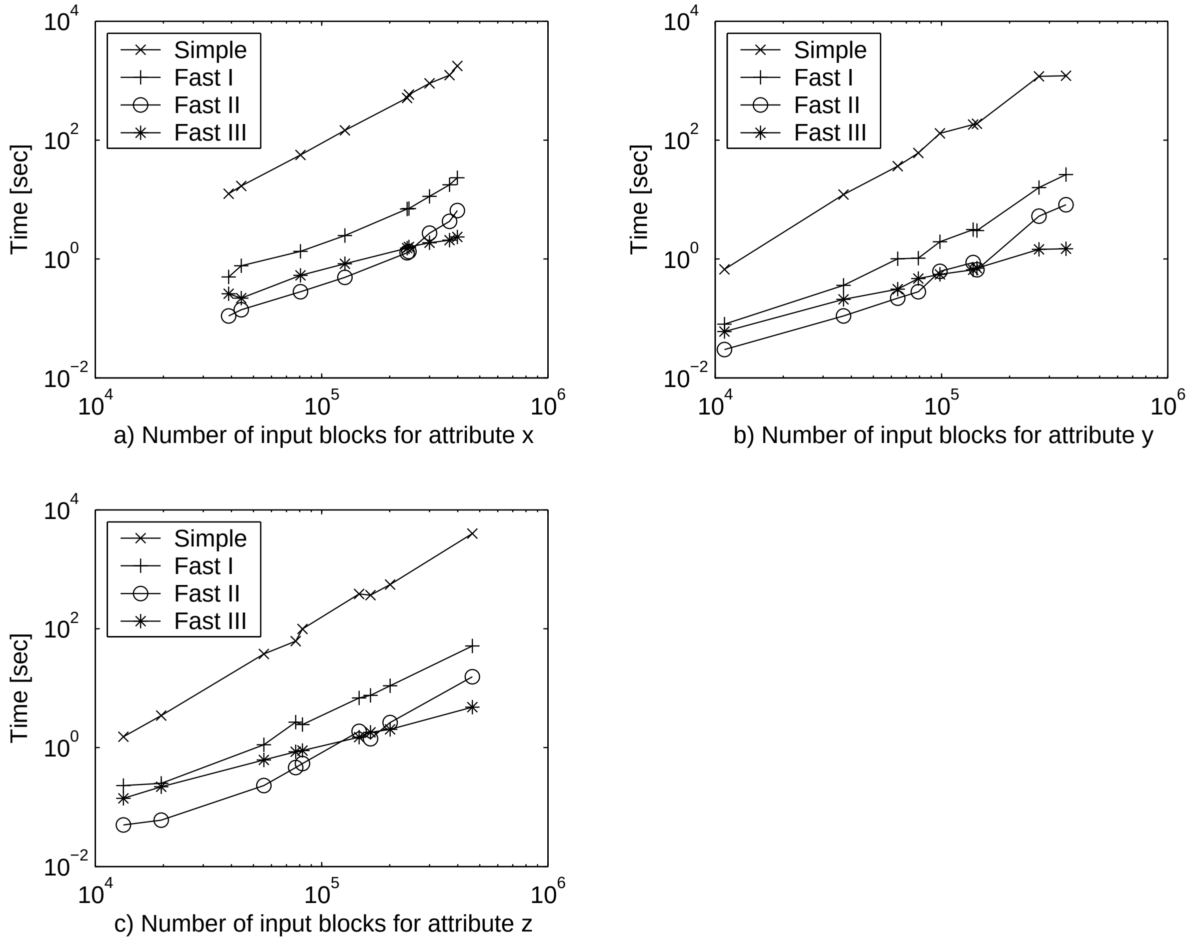

Figure 6: Time [in seconds] for finding connected regions as a function of blocks.

for attribute y the average search time for the simple algorithm is 333 seconds. For the algorithms *FastRegionSearch I, II* and *III* the search times are 5.9, 1.8 and 0.65 seconds respectively. This shows that *FastRegionSearch III* significantly outperforms all other algorithms of a factor of 3 up to 500.

5 Conclusions

In this paper, we demonstrated that compressed bitmap indices can be used efficiently to speed up identifying regions of interest. The process of identifying regions of interest can be accomplished with a searching step and a region-growing step. Compressed bitmaps are well-suited for the searching step [11, 14]. The key contribution of this paper is to demonstrate that the output from the searching step can be efficiently used

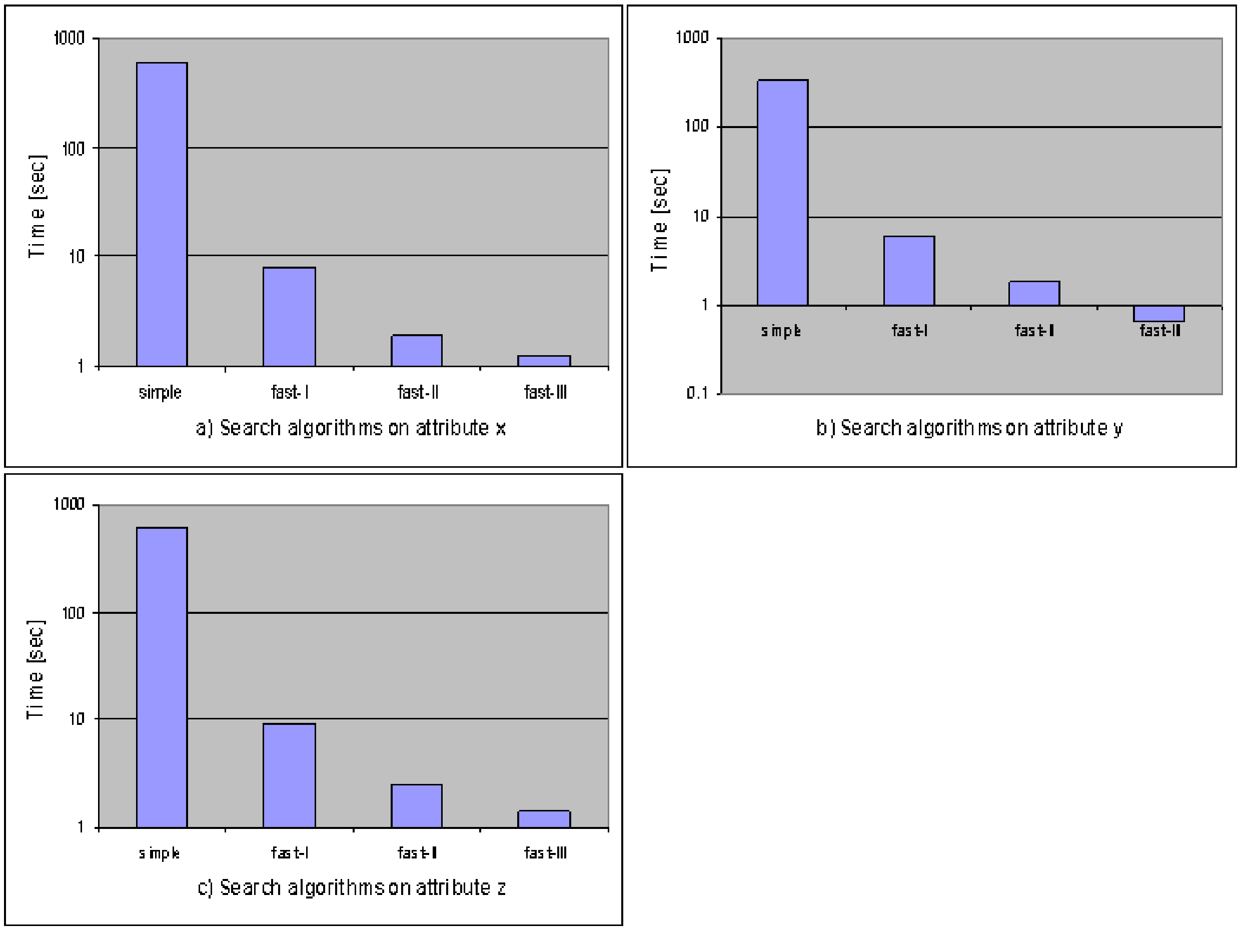

Figure 7: Average time [in seconds] for finding connected regions.

for the region-growing step. Because the output of the searching step can be easily organized into blocks of consecutive cells, the region-growing step is observed to scale linear in the number of blocks. Since the number of blocks is much smaller than the number of cells on the boundaries of the regions of interest, and also much smaller than the total number of cells, our algorithm scales better than the best known connected component labeling algorithms [1, 4] and the iso-contouring algorithms [2, 8].

In the future, we plan to analyze the theoretical complexity of the new region-growing algorithm and conduct performance tests against the best known connected component labeling and iso-contouring algorithms.

References

[1] F. Chang, C.-J. Chen, and C.-J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Comput. Vis. Image Underst.*, 93(2):206–220, 2004.

[2] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Volume Visualization Symposium*, pages 31–38, 1996.

[3] T. Echekki and J. H. Chen. Direct numerical simulation of autoignition in non-homogeneous hydrogen-air mixtures, 2003. to be published in Combustion and Flame.

[4] C. Fiorio and J. Gustedt. Two linear time union-find strategies for image processing. *Theor. Comput. Sci.*, 154(2):165–181, 1996.

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

[6] H. G. Im, J. H. Chen, and C. K. Law. Ignition of hydrogen/air mixing layer in turbulent flows. In *27th International Symposium on Combustion, The Combustion Institute*, pages 1047–1056, Boulder, CO, 1998.

[7] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems*, pages 40–59, Asilomar, CA, 1987. Springer-Verlag.

[8] H. W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.

[9] Q. Shi and J. F. Jaja. Efficient techniques for range search queries on earth science data. In J. Kennedy, editor, *Fourteenth International Conference on Scientific and Statistical Database Management*, pages 142–151, Edinburgh, Scotland, 2002. IEEE Computer Society.

[10] M. Steinbach, P.-N. Tan, V. Kumar, S. Klooster, and C. Potter. Discovery of climate indices using clustering. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 446–455, Washington, D.C., 2003. ACM Press.

[11] K. Stockinger, K. Wu, and A. Shoshani. Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of DOLAP'02*, McLean, VA, 2002. ACM Press.

[12] K. Wu, W. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *Proceedings of SSDBM 2003*, pages 65–74, Cambridge, MA, 2003.

[13] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of SSDBM'02*, pages 99–108, Edinburgh, Scotland, 2002.

[14] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. Technical Report LBNL-54673, Lawrence Berkeley National Laboratory, Berkeley, CA, 2004. To appear in VLDB 2004.