Indexing Query Regions for Streaming Geospatial Data

Quinn Hart

CalSpace
University of California, Davis
Davis, CA 95616, U.S.A.
qjhart@ucdavis.edu

Michael Gertz

Department of Computer Science
University of California, Davis
Davis, CA 95616, U.S.A.
gertz@cs.ucdavis.edu

Abstract

This paper introduces the Dynamic Cascade Tree (DCT), a structure designed to index query regions on multi-dimensional data streams. The DCT is designed for a stream management system with a particular focus on Remotely-Sensed Imagery (RSI) data streams. For these streams, an important query operation is to efficiently restrict incoming geospatial data to specified regions of interest. As nearly every query to an RSI stream has a spatial restriction, it makes sense to optimize specifically for this operation. In addition, spatial data is highly ordered in it's arrival. The DCT takes advantage of this trendiness. The problem generalizes to solving many *stabbing point* queries. While the worst case performance is quite bad, the DCT performs very well when the stabbing point exhibits certain trending characteristics that are common in RSI data streams. This paper describes the DCT, discusses performance issues, and provides extensions of the DCT.

1 Introduction

New methods for processing streaming data [1, 2, 5] have a great deal of potential impact for remotely-sensed geospatial image data originating from the various satellites orbiting the Earth. Besides its typically large bandwidth, Remotely-Sensed Imagery (RSI) data has a number of characteristics that are different from generic streaming data. One important difference is that streaming RSI data is highly organized with respect to it's spatial components. This organization varies for different data streams, but generally image data will arrive in contiguous packets of data. These packets may be individual pixels, rows of pixels, or small images, depending on the instrument. Within a packet, the organization of the pixels is well defined. Consecutive data packets from an RSI data stream are usually close to one another spatially. Also, an RSI data stream is arriving at a high data rate, but usually only at one or a small number spatial locations at a time. In addition, most queries against an RSI data stream include operations to restrict the geospatial data to be processed to specified regions of interest. Therefore, an RSI stream management system needs to efficiently intersect incoming geospatial image data with a possibly large number of query regions.

In this paper, we present a method for intersecting incoming geospatial image data with multiple spatial restrictions, that is, queries that request incoming data for particular regions only. For this, we introduce the Dynamic Cascade Tree (DCT), a structure to index query regions and to provide for efficient insertions and deletions of queries. The DCT supports *stabbing point queries* [3] for a single moving point. A stabbing query is a simple query that, for a given point, will identify all indexed regions that contain that point. For an incoming RSI data stream, the structure is used to efficiently determine what queries are interested in that data. The trendiness inherent to most types of streaming RSI data is exploited to build a small index that is especially efficient when multiple stabbing queries are in close proximity. Based on the information provided by the DCT, query plans can be generated and incoming data can be pipelined to respective query operators, thus providing the basis for multiple-query processing models for streaming RSI data.

The remainder of the paper is structured as follows. Section 2 describes related research for similar problem domains. Section 3 outlines the data and query model underlying RSI. Section 4 describes the *DCT* in detail. Section 5 discusses the performance of the *DCT*, and implications of input regions and stabbing point trends. Section 6 describes extensions and modifications of the *DCT*.

Copyright held by the author(s).

Proceedings of the Second Workshop on Spatio-Temporal Database Management (STDBM'04), Toronto, Canada, August 30th, 2004.

2 Related Work

Many data structures have been developed for one and two dimensional stabbing queries including, among others, interval trees, priority search trees, and segment trees [3]. Space partitioning methods for answering stabbing queries include quadtrees, hashes, and numerous variants of R-trees.

The two most common methods for solving stabbing queries in two dimensions are multi-level segment trees [3] and R-trees [4]. Using multi-level segment trees, one dimension of the region is stored in a segment tree, while the second dimension is indexed with an associated interval structure for each node in the first segment tree. Storage for these structures can be $O(n \lg n)$, with stabbing query times of $O(\lg^2 n)$. Dynamic maintenance of such a structure is more complicated and requires larger storage costs [12]. It is difficult to modify the multi-level segment tree to improve results for trending data. If the input stabbing point moves a small distance, which doesn't change the query results, it still would take $O(\lg n)$ time to respond. That is because even if every node in the multi-level segment tree maintains knowledge of the previous point, it would still take $\lg(n)$ time to traverse the primary segment tree to discover that no changes to the query occurred.

R-trees solve the stabbing query problem by recursively traversing through successive minimum bounding rectangles that include the extent of all regions in the sub-tree, generally with good performance. Since these rectangle regions can overlap, there can be no savings from knowing the previous stabbing query, as there is no way to know if an entirely new path through the segment tree needs to be traversed. R^+-trees[11] can have better performance for these trending stabbing points, since the minimum bounding rectangles are not allowed to overlap and so maintaining the previous query can help verify a query hasn't left a particular region. R^+-trees have problems with redundant storage, dynamic updates, and potential deadlocks [7].

Probably the approach most similar to the *DCT* described in this paper is that of the Query index [6, 9]. The Query index builds a space partitioning index on a set of static query regions, and at each time interval, it allows a number of moving objects to probe the index to determine overlapping queries. Main memory implementations show that grid-based hashing of query regions generally outperform R-tree or quad-tree based methods. SINA [8] describes an incremental method to solving the problem of intersecting moving objects. However, much of the approach involves efficient integration with disk-based static queries, and a complete main-memory implementation would be more similar to the query index approach.

All the indices described above anticipate a large number of moving objects to be indexed against the query regions. The RSI application described above is different in the sense that the *DCT* index is designed for a single or small number of moving objects, where the input rate of data for that moving object is very high. In this application, the desire is for a small index that can efficiently route a high volume data stream to the query regions, rather than indices that are interested in the location of the moving objects.

In one sense, the *DCT*is basically a method for dynamically maintaining a region around a current point for which the current set of query regions is valid, and identifying where this result set is no longer valid. Another method for dynamically describing a neighborhood of validity for a stabbing query using R-trees was proposed by Zhang et al. [14]. This method builds an explicit region of validity around a current point, which can then be used to verify that a stabbing point will not result in a different response. The technique makes a number of additional queries to the R-tree index in order to build this region. Like the technique described in this paper, this could result in cost savings if many subsequent stabbing queries are located within the region of validity.

3 Data and Query Model

Our data model for RSI data is based on raster images. To allow for different types of objects called image, we employ some concepts from the Image Algebra [13] and extend these concepts to account for the specifics of streaming RSI data.

An image consists of a set of points and values associated with these points. The *point set* of an image is a set of points and an associated measure of distance between points. As our interest is in RSI, we choose as point set $\mathbf{X}$ a subset of $\mathbb{R}^3$, with a point $\mathbf{x} \in \mathbf{X}$ of the form $\mathbf{x} = \{x, y, t\}$. The pair $\{x, y\}$ denotes a spatial location in some spatial reference system, and t denotes a timestamp. Thus, a point set exhibits spatio-temporal characteristics. For example, weather satellites continuously transmit images of clouds over one hemisphere of the earth. Given such an image, the *point set* corresponds to the actual location for each point in the image, along with the time that the image was acquired.

A *value set* $\mathbb{V}$ provides the values associated with points in a given point set. For the weather satellite example, the *value set* includes all the intensity levels in the image. Value sets can be complex, in the case of color images, $\mathbb{V}$ is a subset of $\mathbb{Z}^3$ for the red, green, and blue components. For gray-scale images, it is a subset of $\mathbb{Z}$. Based on the concepts of point and value sets, we can now give a functional representation of an image.

Definition 3.1 Given a point set $\mathbf{X}$ and value set $\mathbb{V}$. A $\mathbb{V}$-valued image $\mathbf{i}$ is a function from $\mathbf{X}$ to $\mathbb{V}$, denoted $\mathbf{i} = \{(\mathbf{x}, \mathbf{i}(\mathbf{x})) \mid \mathbf{x} \in \mathbf{X}\}$. The pair $(\mathbf{x}, \mathbf{i}(\mathbf{x}))$ is called a *pixel* of $\mathbf{i}$. $\mathbf{x}$ is the spatio-temporal component of the pixel and $\mathbf{i}(\mathbf{x}) \in \mathbb{V}$ is the *pixel value* at point $\mathbf{x}$.

Different types of RSI have different orderings and structures. Figure 1 shows these structures. Airborne cameras obtain imagery in an image-by-image manner. Some sensors, such as NOAA's Geostationary Operational Environmental Satellite (GOES), obtain RSI data basically in a row-by-row fashion. Although conceptually the data collected by GOES can be viewed as a stream of images, the images are actually obtained in a *row-scan order* in which pixels are delivered a few lines at a time. Still other types of sensors gather data on a pixel-by-pixel basis.

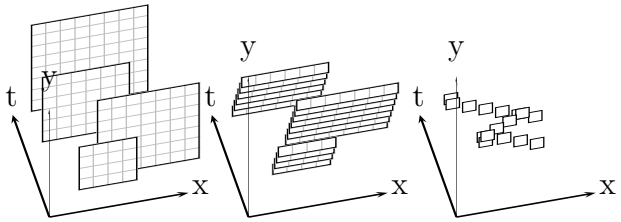

Figure 1: Examples of different point set orderings for streaming RSI data: image-by-image, line-by-line, and point-by-point.

All the above scenarios describing how RSI data is obtained show an important characteristics we aim to exploit in our approach: *the points in a point set exhibit certain trends*. That is, consecutive points in a stream of RSI data have a close spatial and temporal proximity. The only exception is where the last point of a line in an image is followed by the first point of a new image (scenario on the left in Figure 1). As we will show in the following section, knowing about the *trendiness* of incoming geospatial point data can have a significant influence on how queries against a stream of RSI data are processed.

Queries against a stream of RSI data are typically continuous queries that run for a long time and may include complex operators, such as spatial transforms or aggregates (see [13] for operations on point sets). However, since most applications are not interested in the complete region covered by a sensor, *spatial restriction* is a type of operation common to all queries. A spatial restriction specifies a *region of interest*, primarily in the form of a rectangle, and typically precedes other operations on point data. The Dynamic Cascade Tree (DCT) includes an index structure and algorithms to efficiently determine what query regions are affected by incoming RSI image data, and to pass those images to the appropriate queries.

4 The Dynamic Cascade Tree (DCT)

The problem of quickly answering multiple queries on a stream of RSI data is basically solving a normal *stabbing query* [3] for a point. That is, as query result, a stabbing query determines all query regions that contain the current point delivered by the RSI data stream. For RSI data, the stabbing points are special in that the next stabbing point is typically very close to the previous stabbing point. The goal is to take advantage of the trendiness of stabbing points and to develop index structures that improve the search performance for subsequent stabbing queries.

The structure proposed in the following builds an index that is dynamically tuned to the current location of RSI data. For a given point, the *DCT* maintains the regions around that point where the query result will change. Stabbing queries can be answered in constant time if the new stabbing point has the same result as the previous query and will incrementally update a new result set based on the previous set when the result is different. The structure is designed to be small and quickly allow for insertions and deletions of new query regions. It assumes some particular characteristics of the input stream, notably that the stream changes in such a way that many subsequent incoming RSI data will contribute to the same result set(s) to region queries as the current point. Therefore, the cost of maintaining a dynamic structure can be amortized over a large set of queries. Section 5 describes in more detail the performance implications of the regions and input data stream.

4.1 DCT Components

Figure 2 gives an overview of the data structure employed, which we term a *Dynamic Cascade Tree (DCT)*. The figure shows a set of query regions $a, b, \ldots, f$, the node *cn*, denoting the most recent stabbing point from the data stream, and the associated structures for the *DCT*. The figure describes a *DCT* that indexes two dimensions. There is no required order in how the dimensions are referenced, and the example shows the vertical (y) dimension being the first dimension indexed in the *DCT*.

The components of *DCT* are pleasantly simple extensions to a binary tree. In the example and following pseudo-code, we assume that we have two simple search structures, *List* and *2-Key-List*. *List* supports INSERT(KEY,VALUE), DELETE(KEY), and ENUMERATE(). Keys in *List* are unique for each value. In our approach, we use a simple skip list [10] to implement *List*. The *2-Key-List* is incrementally more complex. It supports INSERT($key_1, key_2, value$), DELETE(key_1, key_2) and ENUMERATE(key_1) using two keys. The combination of two keys is unique for each value. ENUMERATE(key_1) enumerates all the values in the *2-Key-List*, entered with key_1. An implementation of *2-Key-List* could be a skip list using key_1, where each node has an associated skip list using key_2. With this implementation, for *2-Key-List* .DELETE(key_1, key_2), if the deletion causes an empty set in the associated key_1 node, then that entire node is deleted.

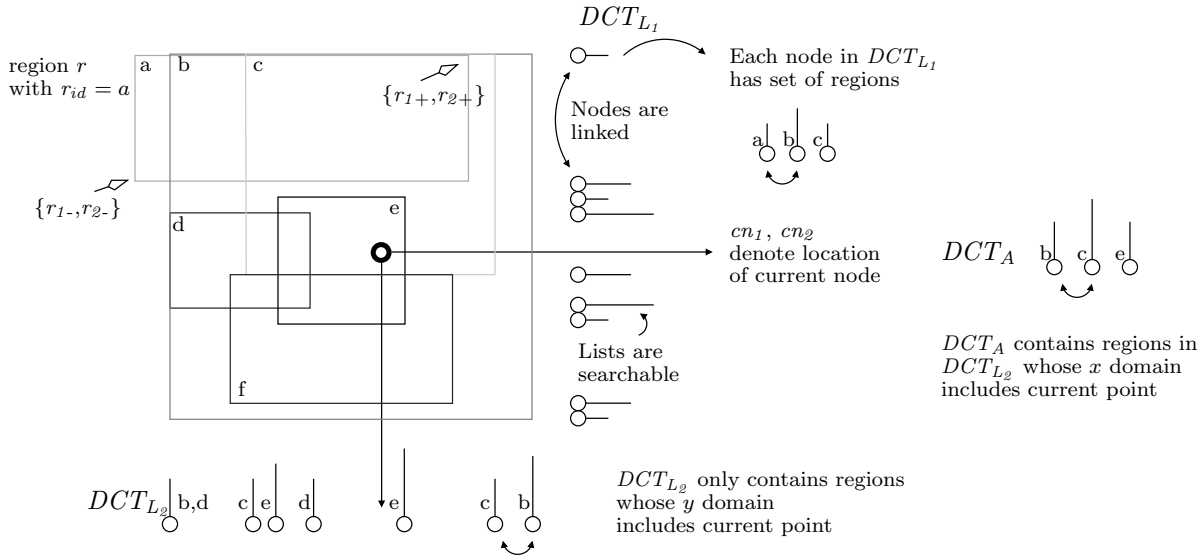

Figure 2: Dynamic Cascade Tree (DCT) with query regions $a, b, \ldots, f$; a query region, r, is described by it's minimum (lower left) corner $\{r_{1-}, r_{2-}\}$ and maximum (upper right) corner $\{r_{1+}, r_{2+}\}$

List leaf nodes contain region ids, r_{id} as keys with a pointer to the query region, r as the value. For the *2-Key-List*, key_1 is the value of the endpoint, and key_2 is the id of the query region, denoted r_{id}. The leaf nodes of a *2-Key-List* correspond to the half-open line segments between two endpoints. Leaf nodes of a *2-Key-List* also have pointers to the next and previous nodes in sorted order, allowing for linked list traversal to leaf nodes. One reason for choosing a skip list implementation is that the forward pointers already exist, and only an additional back pointer is added to a normal skip list.

The *DCT* maintains a separate *2-Key-List* for each dimension of the individual query regions. In Figure 2, these are DCT_{L_1} and DCT_{L_2}. In addition, the *DCT* maintains a *List*, DCT_A, of all query regions that overlap the current stabbing point. A second structure, *cn*, maintains pointers to nodes within each *2-Key-List*, corresponding to the most current stabbing point.

The *2-Key-List* for the first dimension, DCT_{L_1} contains the minimum and maximum endpoints in the 1st (y) dimension, $\{r_{1-}, r_{1+}\}$, for every query region r.

The next *2-Key-List*, DCT_{L_2} contains keys on the endpoints in the 2nd (x) dimension and r_{id}. DCT_{L_2} does not contain the endpoints of all the regions in *DCT*, but only the regions whose 1st dimension (y) overlap with the current node, cn_1.

If the query regions contain more dimensions, additional *2-Key-List* structures are added to the *DCT*, where each subsequent *2-Key-List* only indexes those regions that overlap the current point up to that dimension.

In Figure 2, *cn* contains two pointers, cn_1 and cn_2 to nodes within both DCT_{L_1} and DCT_{L_2} corresponding to the location of the most recent stabbing point.

DCT_A is the final *List* that contains all the currently selected query regions that correspond to the current stabbing point query. Just as DCT_{L_2} contains only a subset of the regions of DCT_{L_1} that contain the cn_1 node, DCT_A contains the subset of DCT_{L_2} where the cn_2 node is contained by the 2nd dimension of each region. The DCT_{L_1}, DCT_{L_2}, and DCT_A structures make up a cascade of indexes, each a subset of the previous index.

4.2 Updating query regions in the DCT

The *DCT* is initialized by creating the *2-Key-List* and *List* structures, adding a starting node for DCT_{L_2} and DCT_{L_1} outside their valid range, and assigning cn_2 and cn_1 to those nodes.

Algorithm 1 shows the pseudo-code for inserting query regions into *DCT*. Insertion and deletion are simple routines. For insertion, a region is first inserted into DCT_{L_1} and then successively into DCT_{L_2} and DCT_A if the region overlaps the current node *cn* in the other dimensions. DELETE-REGION is similar to the insertion, taking a region r as input. It should be clear that the structures DCT_{L_2} and DCT_A need to be maintained when new regions are inserted and deleted, and for each new stabbing point. Since DCT_{L_2} contains regions overlapping the current node *cn*, when a new stabbing point arrives where a y boundary for any region in the *DCT* is crossed, then the DCT_{L_2} structure needs to be modified to account for the regions to be included or deleted from consideration. A similar method needs to be associated with boundary crossings in the x dimension while traversing DCT_{L_2} and modifying DCT_A.

Algorithm 1 Inserting Query Regions in DCT

```
INSERT-REGION(DCT, cn, r)
1  ▷ Input: DCT,
2  ▷        current node cn= {cn_2,cn_1}
3  ▷        region, r = {r_2-, r_1-, r_2+, r_1+}
4  INSERT-ITH(DCT, cn, r, 1)

INSERT-ITH(DCT, cn, r, i)
1  ▷ Input: DCT,cn,r same as INSERT-REGION
2  ▷        dimension, i
3  I ← DCT_{L_i} ▷ ith 2-Key-List
4  if (i > dimensions of r)
5     then DCT_A.INSERT(r_id, r)
6     else I.INSERT(r_i-, r_id, r)
7          I.INSERT(r_i+, r_id, r)
8          if (r_i- ≤ cn_i . key and r_i+ > cn_i . key)
9             then INSERT-ITH(DCT, cn, r, i + 1)
```

4.3 Querying the DCT

The algorithm for reporting selected (active) query regions for a new stabbing point $np = \{np_1, np_2\}$ begins by traversing the *2-Key-List* DCT_{L_1} in the y direction from the current node $cn = \{cn_1, cn_2\}$ to the node containing np_1 going through every intermediate node using the linked list access on the leaf nodes of DCT_{L_1}. At each boundary crossing, as regions are entered or exited, those regions need to be added to or deleted from the DCT_{L_2} *2-Key-List*. When the point has traversed to the node containing np_1, traversal begins in the x direction, moving from cn_2 to the node containing np_2. As with DCT_{L_1}, when the traversal hits x boundary points, the entered query regions are added to DCT_A and the exited regions are deleted from DCT_A. When the traversal reaches np, cn contains pointers to the nodes containing np, DCT_{L_2} contains x endpoints to all the regions with y domains that contain np_1, and DCT_A lists all regions that contain np. DCT_A is then enumerated to report all the query regions that are affected by the new stabbing point np.

Figure 3 shows an example of an update of the structures within DCT on reporting regions for a new stabbing point. This extends the example of Figure 2. In this example, the new point has crossed a y boundary that contains two region endpoints, c and f. As the current point traverses in the y direction to this new point, the x endpoints of region c are removed from DCT_{L_2}, and the endpoints of f are added to DCT_{L_2}. When the endpoints of these regions are deleted, the regions themselves are also deleted from DCT_A. In the example, c is deleted from and f inserted into DCT_A. After reaching np_1, DCT_{L_2} is traversed in the x direction. In the example, this results in e being deleted from DCT_A. Finally, DCT_A is enumerated, completing the procedure.

Algorithm 2 describes the REPORT-REGIONS procedure, which reports query regions for a new stabbing point, while updating the structures of the DCT. REPORT-REGIONS simply calls UPDATE-ITH on the first dimension and then reports all regions in the DCT_A *List*. The procedure UPDATE-ITH recursively visits each dimension in the DCT and adds and deletes regions from the associated *2-Key-List* for that dimension. In UPDATE-ITH, Lines 7 to 12 traverse the dimension backwards. At each endpoint, the corresponding region is either added or removed from *2-Key-List* in the next dimension. Lines 13 to 18 execute a similar traversal in the forward direction, also adding and removing regions from the next *2-Key-List*. Only one of the **while** loops is executed at each invocation. After traversing to np_i, UPDATE-ITH is called again for the next indexed dimension, $i + 1$. This continues through all dimensions of the DCT. Note that INSERT-ITH will add regions into the DCT_A *List*when traversing the final dimension of the DCT. When all dimensions have been traversed, DCT_A contains all the regions that overlap np in all dimensions.

Algorithm 2 Stabbing queries in DCT

```
REPORT-REGIONS(DCT,cn,np)
1  ▷ Input: DCT
2  ▷        current node(s) cn= {cn_1,cn_2,...}
3  ▷        new stabbing point, np= {np_1,np_2,...}
4  ▷ Output: List of query regions containing np.
5  UPDATE-ITH(DCT,cn,np,1)
6  return DCT_A.ENUMERATE

UPDATE-ITH(DCT, cn, np, i)
 1  ▷ Input: same as REPORT-REGIONS
 2  ▷        dimension, i
 3  ▷ Output: List of query regions containing np.
 4  if (i > max dimension of r)
 5     then return
 6  I ← DCT_{L_i} ▷ i-th 2-Key-List of DCT
 7  while (np_i < cn_i . key)
 8        do for r ∈ I.ENUMERATE(cn_i)
 9           if r_i- = cn_i . key
10              then DELETE-ITH(DCT,cn,r,i + 1)
11              else INSERT-ITH(DCT,cn,r,i + 1)
12        cn_i ← cn_i .PREV
13  while (np_i > cn_i .NEXT. key)
14        do cn_i ← cn_i .NEXT
15           for r ∈ I.ENUMERATE(cn_i)
16           if (r_i+ = cn_i . key)
17              then DELETE-ITH(DCT,cn,r,i + 1)
18              else INSERT-ITH(DCT,cn,r,i + 1)
19  UPDATE-ITH(DCT,cn,np,i + 1)
```

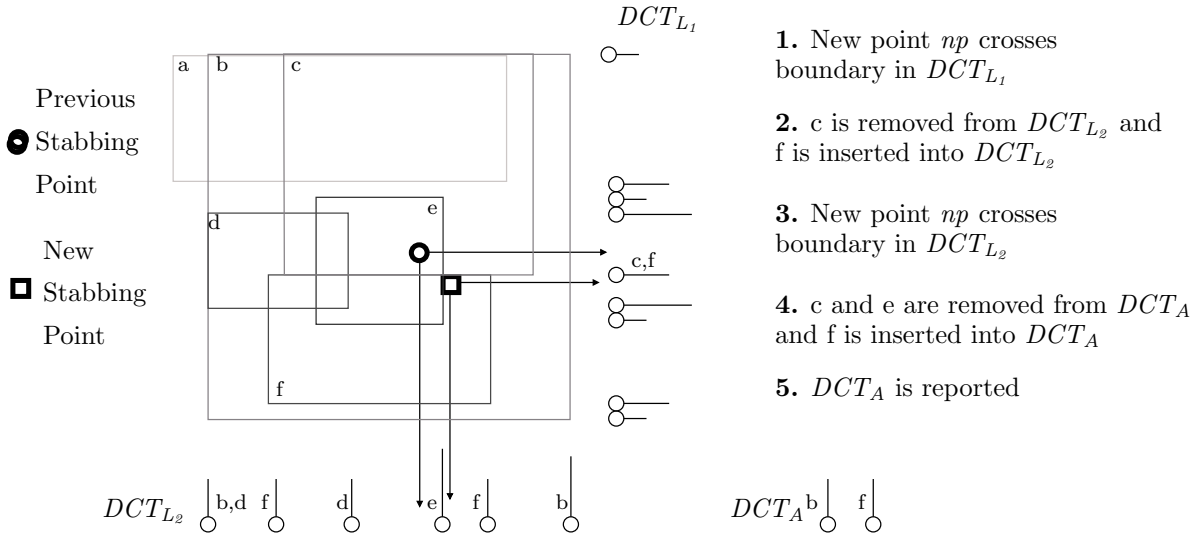

Figure 3: Stabbing point moving in REPORT-REGIONS

5 DCT Performance

The performance of REPORT-REGIONS is highly dependant on the location of the regions, the trending properties of the stream data, and the interaction of the two parameters. For executions of REPORT-REGIONS, with n being the total number of query regions and k being the average number of resultant regions, the average time of execution can range from $O(k)$ in the best case to $O(n \lg n + k)$ in the worst case. Reasonable experiments could be designed that would approach either of these limits. Instead, there are rules to consider for the application of the DCT.

5.1 Insertions and deletions of query regions

The DCT data structure is small and robust to many insertions and deletions of query regions. Insertions and deletions take $O(\lg n)$ time as the query region is potentially added to the structures DCT_{L_1}, DCT_{L_2}, and DCT_A. *List* and *2-Key-List* are simple to maintain dynamically in $O(n)$ space.

5.2 Number of boundaries crossed

The DCT is designed for trending data, which can most quantitatively be measured by the number of region boundary crossings from one stabbing point to the next. This structure works best when the number of query boundaries crossed on subsequent input points is not large. When no boundaries are crossed, then no internal lists are modified, and REPORT-REGIONS runs in $O(k)$ time. When a boundary is crossed in the i-th dimension, then each region in the crossed DCT_{L_i} node needs to be inserted into or deleted from the $(i+1)$-th *2-Key-List* structure. This is true for regions whose domains in subsequent dimensions do not overlap the new stabbing point *np*, and thus do not contribute to the DCT_A structure. The cost of REPORT-REGIONS in this case can be as high as $O(n \lg n + k)$, since many non-overlapping regions are inserted into the DCT_{L_2} structure. There is no performance difference in whether the crossing occurs at a single boundary with many regions at that node, or over many boundary crossings with few regions.

The DCT data structure indexes lazily in the sense that for insertions of new regions that do not overlap cn, it only indexes a new region on its first dimension, and not on all dimensions. Thus, boundary crossing costs must take some time to index on these dimensions of the regions. The problem with the DCT is that these costs can occur many times in the travel of the input stabbing points. Rather than indexing these values once, the DCT re-indexes a subset of points multiple times as boundaries are crossed. The hope is that in a new set of regions, many subsequent stabbing points will be in these areas, and the low cost of those stabbing points will make up for the extra cost of maintaining a dynamic index.

5.3 Trajectory of the trending data point

Another aspect affecting performance is the trajectory of the input stabbing points. For example, consider a DCT in two dimensions, like the one shown in Figure 2, with trajectories that are increasing or decreasing monotonically in the x and y dimensions. In these cases, regions are put into the DCT_{L_2} structure at most one time. The total time maintaining the DCT_{L_2} structure then is at most $O(n \lg n)$, fixing a bound on the dynamic maintenance costs of the DCT. The total cost of m stabbing queries over that trajectory would be $O(n \lg n + mk)$, where k is the average number of regions per stabbing point. For a two-dimensional segment tree implementation, the total cost would be $O(n \lg n + m \lg^2 n + mk)$, which includes the static cost of maintaining a segment tree and does not include extra costs for dynamically maintaining that tree.

On the other hand, data with a more erratic trajectory can result in poor performance. Consider a point that repeatedly crosses a single boundary containing all n region boundaries. Again, each iteration would require $O(n \lg n)$ time, as the DCT_{L_2} and DCT_A structures are both repeatedly made up and torn down.

Also, the DCT as described in the Figures above, which indexes on y and then x, favors stream data points that trend in the x direction over data that trend in the y direction. The reason for this is that more regions added into the DCT_{L_2} structure end up being reported, and the dynamic structure building is not wasted. Also, there are fewer insertions and deletions in the DCT_{L_2} structure in the first place. When the stabbing point np crosses a boundary in the x direction, it still takes $O(\lg n)$ time to insert, as it updates DCT_A, but this is more useful work in updating DCT_A than a y crossing boundary, which can spend wasteful time adding points into DCT_{L_2} that might never be used. Where a stabbing query for a y trending trajectory can have a worst case time in Report-Regions of $O(n \lg n + k)$, the worst case time in x trending stabbing points is $O(n + k)$, when worthless insertions into DCT_A are skipped, as described below.

This shows that order in the cascade is very important, and dimensions that see more boundary crossings for subsequent stabs into the DCT should be pushed deeper into the structure. Boundary crossings are of course dependent on the trajectory of the stabbing point and the organization of the regions in the DCT.

5.4 Skipping worthless insertions

As mentioned above, when the next point of the input stream trends a long way with respect to the number of query boundaries traversed, then the time for Report-Regions goes up to at least the number of regions contained in all the boundaries crossed. Regions that are both entered and exited in the course of a single traversal to np are even worse. Their endpoints are needlessly added, then deleted from DCT_{L_2}, at a cost of up to $O(\lg n)$, and never queried. This can easily be remedied, but for clarity was left out of the initial Update-Ith algorithm. When encountering a region at a boundary crossing, check that the region will remain a valid region when np has finished it's traversal before inserting into the *2-Key-List* structure. This prevents wasted index modifications, but does not help with the basic problem of long traverses of np or input points that cross back and forth across expensive boundaries. Algorithm 3 shows the modifications made to Update-Ith in lines 7 to 12 in Algorithm 2. A similar modification would be made to the forward loop in Update-Ith.

Algorithm 3 Stabbing query modification

```
 8  ...
 9  ▷ replaces lines 7 to 12 in UPDATE-ITH
10  old = cn_i.key
11  while (np_i < cn_i . key)
12        do for r ∈ I.ENUMERATE(cn_i)
13           if (r_i- = cn_i . key) and (old < r_i+)
14              then
15                   DELETE-ITH(DCT,cn,r,i + 1)
16           elseif (r_i+ = cn_i . key) and (r_i- < np_i)
17              then
18                   INSERT-ITH(DCT,cn,r,i + 1)
19        cn_i ← cn_i .PREV
20           ...
```

6 DCT Extensions and Modifications

In Section 4, the discussion centered on answering a simple stabbing query for a single point and a number of regions in a two dimensional space. This basic framework can undergo some simple modifications to handle a number of similar types of queries.

6.1 Window queries

The first general area is for regions other than points. The stabbing query can easily be changed from a single point to a constant size rectangle. Constant size rectangles are common in RSI data. In this case, track the center location of the stabbing rectangle, and when inserting new regions of interest, extend the boundaries by half the width and height of the rectangle queries. Intersections of the modified regions and the stabbing point will coincide with intersections of the original regions and the rectangle query.

Queries on stabbing rectangles with more dynamic extents are possible, too. For example, in the two dimensional case, track both edges of the stabbing rectangle in the DCT_{L_1} and DCT_{L_2} structures. The leading edge of the lines in the y dimension will track insertions into DCT_{L_2}, and the trailing edges will track deletions from DCT_{L_2}. Leading and trailing are with respect to the previous stabbing rectangle. Rectangles that are growing in size from the previous rectangle may have two leading edges, and shrinking rectangles will have two trailing edges. A similar strategy is used for mapping the DCT_{L_2} structure to the DCT_A structure.

Many remote-sensing image data come in a row-by-row scheme (see Figure 1). For this special case, use and maintain the DCT_{L_1} structure as in the stabbing point example, and only modify the DCT_{L_2} structure for different lengths of the individual rows of data.

6.2 Adding dimensions

Adding an additional dimension is a simple extension by adding another intermediate *2-Key-List* to the DCT_{L_1}, DCT_{L_2}, and DCT_A cascade. For example, a time dimension on a rectangle query, or in this instance a cube query, could be added. As discussed

in Section 5, it is best to order the dimensions so the most varying is on the deeper levels of the *DCT*. The monotonic increase of time would make it a good candidate for the level before the DCT_A structure. However, if a system contains query regions with a mostly unbounded temporal dimension, then it could also be located at the first level. One nice feature of making time the first structure of the cascade is that it also does double duty in providing a structure to prune regions that have expired. For geospatial data streams, if incoming pixels are timestamped, then new stabbing points that have identified regions whose time extent has ended can be removed from the time structure as well as the other cascaded structures.

6.3 Non-spatial multi-dimensional data

The focus of the *DCT* has been on spatial data. However, these techniques could be applied to a general multi-dimensional data space. The obvious modifications could be made and extended to n dimensions as outlined above. One important issue to address is the order of the cascade of *2-Key-List* structures. As described in Section 5, if regions are dispersed equally, it is generally best to move the most varying parameter to the end of the cascade, and move the least varying to the top. This structure is also appropriate for range queries over a single dimension over trending data by maintaining only the top *2-Key-List*. Since the index sizes are relatively small, it is conceivable that a system could consistently maintain one dimensional *DCT* structures, and then dynamically begins to build two or n dimensional structures when queries requesting such regions are instantiated. The advantage of this method is that the structures are no longer maintained as the queries requesting those regions are deleted.

7 Conclusions and Future Work

In this paper, we have presented the Dynamic Cascade Tree (DCT), a simple data structure designed to follow trending geospatial data points that constitute streaming geospatial image data. The focus has been on two dimensional stabbing queries, but we have offered modifications to a number of related problems. Theoretical and rule of thumb performance bounds have been discussed. Initial further work will focus on more experimental tests of the *DCT* for various realistic scenarios.

We are currently implementing the *DCT* as part of a query processing architecture to support complex continuous queries over streams of remotely-sensed geospatial image data. The proposed *DCT* will build an important component to facilitate the optimization of multiple queries against such a stream.

Acknowledgments

This work was partially supported by the NSF grant IIS-0326517.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and issues in data stream systems. In *Proc. of the 21st Symposium on Principles of Database Systems*, 1–16, 2002.

[2] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB 2002*, 215–226.

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, 47–57, 1984.

[5] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[6] D. V. Kalashnikov, S. Prabhakar, S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distrib.& Parallel Databases*, 15(2):117–135, 2004.

[7] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, Y. Theodoridis. R-Trees have grown everywhere. Unpublished Techn. Report, 2003.

[8] M.F.Mokbel, X. Xiong, W.G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 623–634, 2004.

[9] S. Prabhakar, Y. Xia, D.V. Kalashnikov, W.G. Aref, S.E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. on Computers*, 51(10), 2002.

[10] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, 1990.

[11] T. K. Sellis, N. Roussopoulos, C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *VLDB 1987*, 507–518.

[12] M. J. van Kreveld, M. K. Overmars. Concatenable segment trees. Technical report, Rijksuniversiteit Utrecht, 1988.

[13] J. N. Wilson, G. X. Ritter. *Handbook of Computer Vision – Algorithms in Image Algebra*. CRC Press, 2nd edition, 2001.

[14] J. Zhang, M. Zhu, D. Papadias, Y. Tao, D. L. Lee. Location-based spatial queries. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, 443–454, 2003.