Evaluation of a Dynamic Tree Structure for Indexing Query Regions on Streaming Geospatial Data

Quinn Hart[1], Michael Gertz[2] and Jie Zhang[2]

[1] CalSpace, University of California, Davis, CA 95616, U.S.A.
[2] Dept. of Computer Science, University of California, Davis, CA 95616, U.S.A.

Abstract. Most recent research on querying and managing data streams has concentrated on traditional data models where the data come in the form of tuples or XML data. Complex types of streaming data, in particular spatio-temporal data, have primarily been investigated in the context of moving objects and location-aware services. In this paper, we study query processing and optimization aspects for streaming Remotely-Sensed Imagery (RSI) data. Streaming RSI is typical for the vast amount of imaging satellites orbiting the Earth, and it exhibits certain characteristics that make it very attractive to tailored query optimization techniques. Our approach uses a Dynamic Cascade Tree (*DCT*) to (1) index spatio-temporal query regions associated with continuous user queries and (2) efficiently determine what incoming RSI data is relevant to what queries. The *DCT* supports the processing of different types of RSI data, ranging from point data to more general spatial extents in which the incoming imagery can be single pixels, rows of pixels, or discrete parts of images. The *DCT* exploits spatial trends in incoming RSI data to efficiently filter the data of interest to the individual query regions. Experimental results using random input and Geostationary Operational Environmental Satellite (GOES) data give a good insight into processing streaming RSI and verify the efficiency and utility of the *DCT*.

1 Introduction

The current interest in data stream management [1, 2, 6] has driven various new processing methods and paradigms for streaming data, such as adaptivity [11, 15] and operator scheduling [3, 5]. Some proposed approaches have extended to the realm of spatio-temporal data, where new methods defining the spatial relationships between queries and data streams are being investigated, in particular in the context of continuous queries over moving objects (e.g., [12, 18, 19]). A new area of research where these streaming data paradigms can have a great impact is in applications surrounding remotely-sensed geospatial image data originating from the various satellites orbiting the Earth.

Figure 1(a) gives an overview of a data stream management system (DSMS) for Remotely-Sensed Imagery (RSI) data. Such data has a number of characteristics that are different from traditional streaming relational or spatio-temporal

data typically described in a DSMS context. One aspect is that the incoming RSI data stream is very large, usually arriving as discrete parts of binary image data. Another important point is that streaming RSI data is more highly organized with respect to its spatial and temporal components than is usually assumed for more generic types of spatio-temporal data. This organization has some profound effects on how data and queries can be processed in a RSI DSMS. The organization of the incoming RSI data affects the query evaluation and the index structures used. In this paper, we discuss in detail an indexing scheme for queries in such a system. This structure, the Dynamic Cascade Tree (*DCT*) utilizes the characteristics of the RSI data and facilitates the efficient routing of relevant portions of the streaming data to query operators.

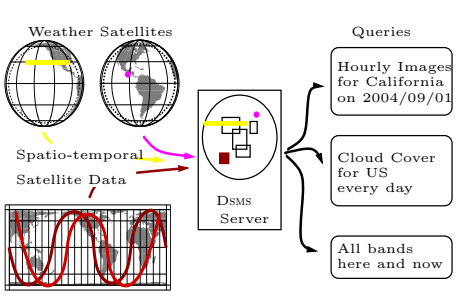
Exactly how data arrives varies for different RSI data streams, but generally image data arrives as discrete contiguous packets of data. The packets may be individual pixels or organized sets of pixel data, depending on the instrument. Different types of RSI data have different orderings and structures. Figure 1(b) illustrates the common structures, including image-by-image, row-by-row, and pixel-by-pixel. Many satellite instruments, such as the Geostationary Operational Environmental Satellite (GOES) weather satellite [7], obtain RSI data in a row-by-row fashion. Although conceptually the data collected by GOES is represented as a set of complete images, the images are incrementally received in *row-scan order* where pixels are delivered a few lines at a time. Other types of sensors gather data on a pixel-by-pixel basis. These include imaging sensors with large pixel sizes, or active sensors such as Light Detection and Ranging (LIDAR) sounding instruments.

Within a packet of data, the spatial organization, represented as image pixels, is well defined. The ordering of the packets is also well defined, and consecutive data packets from an RSI data stream are in close spatial and temporal proximity. Data from an RSI stream usually arrive only at one or a small number of spatial locations for a given instrument.

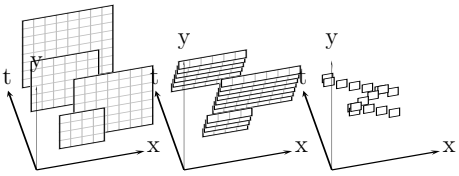
The above scenarios illustrate an important characteristic exploited in the following approach. Consecutive packets in an RSI data stream have close spatial and temporal proximity. In the *DCT*, this spatial trend is used to influence how multiple queries against a stream of RSI data are processed.

1.1 Problem description and objectives

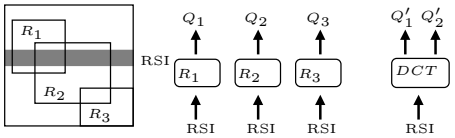
Most continuous queries against an RSI data stream include operations to restrict the spatio-temporal data to specified regions of interest. Such *Continuous Query (CQ) regions* are part of more complex queries users issue against a DSMS. Other, more complex operators of a query against an RSI data stream, such as map projections or spatial and value transforms, typically follow a spatial restriction as they are much less selective. Clearly, query regions specified by different users may overlap. This is typical for RSI streams that have geographic *hot spots* or regions that are of interest to many users. An RSI stream management system needs to (1) efficiently intersect incoming image data with a possibly large number of query regions, and (2) it should provide a means that allows queries



(a) A Remotely-Sensed Imagery (RSI) DSMS. High bandwidth input image streams are handled by the system. The figure shows imagery from both polar orbiting and geostationary sources. Typical queries request image data or other products for specific times and spatial extents.



(b) Different pixel orderings for streaming RSI data: image-by-image, row-by-row, pixel-by-pixel



(c) RSI data intersecting with query regions

Fig. 1. Geospatial data stream system and characteristics of streaming RSI data

to share the incoming data for further processing. These aspects are illustrated in Fig. 1(c), where one region query R_i is associated with each of the user queries Q_i. In the figure, incoming RSI data intersects with two query regions R_1 and R_2 (left). Instead of filtering incoming data for each individual query Q_1, Q_2, and Q_3 (middle), a mechanism is needed to determine what incoming image data is relevant to what queries and pipeline the relevant data to subsequent query operators. The *DCT* is such a mechanism that filters and streams relevant data to subsequent operators of individual queries, here only Q'_1 and Q'_2 (right). To efficiently process these spatial restrictions, the organization of the incoming data stream needs to be considered.

The Dynamic Cascade Tree (*DCT*) is a space efficient structure that indexes query regions that are part of more complex queries against RSI data streams. In particular, the *DCT* supports the efficient processing of a *moving data stream*. A data stream query is one that, for a window describing the spatial extent of the incoming image data (pixel, row, or image), will identify all Continuous Query (CQ) regions that spatially and temporally overlap the data window. This allows the pipelining of image data to those queries to which the data is of interest, and it facilitates the sharing of image data among queries in the case of non-disjoint query regions. The design of the data structures and algorithms underlying the *DCT* is guided by some important requirements, which are typical for RSI data streams: (1) The *DCT* indexes CQ regions and is sufficiently small to be kept in main memory. It also has to support efficient insertion and deletion of CQ regions associated with user queries. (2) The geospatial data stream comes from a single

source corresponding to the real-time streaming satellite data. Sequential stream data are usually in close proximity and might have a regular trajectory through space. The *DCT* has to account for both size and spatio-temporal trends of the incoming data. (3) Because of the size of the CQ regions and the size and shape of the incoming stream data, selectivity is high. For example, incoming RSI data packets intersect about 20% of CQ regions for typical GOES applications.

1.2 Structure of the paper

Section 2 describes the *DCT* and shows how it filters incoming image data for multiple CQ regions. Section 3 details the performance of the *DCT* in general terms and discusses the parameters affecting performance. In Section 4, several experimental results are presented. These include experiments on random data to study the performance under changing input parameters, and experiments closer to real world scenarios using Geostationary Operational Environmental Satellite (GOES) data as a practical example. Section 5 describes related research for similar problem domains. Section 6 concludes the paper and highlights future research directions.

2 The Dynamic Cascade Tree (DCT)

The *DCT* data structure is designed to use a single rectangular window of pixel data from a Remotely-Sensed Imagery (RSI) data stream as input to a *window query* on a number of CQ regions associated with user queries, as illustrated in Fig. 1(c). The *DCT* uses the spatial extent of the most recent *input window* to quickly answer multiple queries on a stream of RSI data. That is, *for a given input window from the stream, the DCT returns all CQ regions that overlap this input*. For streaming RSI data, input windows correspond to individual packets of contiguous image data. The regions correspond to the spatial extents of the individual CQ queries registered in the data stream management system (DSMS).

The most important aspect for RSI data in terms of designing the *DCT* is that the incoming stream data comes as contiguous data packets that are typically in close spatial and temporal proximity to the previous data. Our goal is to take advantage of this proximity and to develop an index structure that improves the search performance for subsequent parts of the stream.

The *DCT* structure realizes an index that is dynamically tuned to the current location of an input RSI window. For the spatial extent of the most recent input window, the *DCT* maintains the CQ regions around that window where the result set will change. New RSI input can be processed very quickly if the new window has the same result set of CQ regions as the previous window; and will incrementally update the result set otherwise. The structure is designed to be small, and allow for fast insertions and deletions of new CQ regions registered by the DSMS. It assumes some particular characteristics of the input stream, notably that the stream changes in such a way that many successive data extents from the RSI stream will share the same result set of CQ regions. Therefore, the cost of maintaining a dynamic structure can be amortized over the incoming data

stream. Sections 3 and 4 examine in more detail the actual performance with different CQ regions and data stream parameters. First a general overview of the components of the DCT is given, followed with details on how the DCT is built, maintained, and queried. Hart [10] gives a more complete algorithmic description of the DCT, including a simpler point-based version of the DCT and algorithms for insertion, deletion, and queries.

2.1 DCT components

Figure 2 gives an overview of the DCT. The figure shows a set of rectangular CQ regions $a, b, \ldots, f$, and a rectangular input window corresponding to the most recent data in the RSI stream. Also shown are the associated structures that make up and maintain the DCT. This figure describes a DCT that indexes two dimensions, although the DCT can be extended to more. The figure shows the dimensions being cascaded first in the x dimension and then in the y dimension. The DCT maintains separate trees for both the minimum and maximum end-

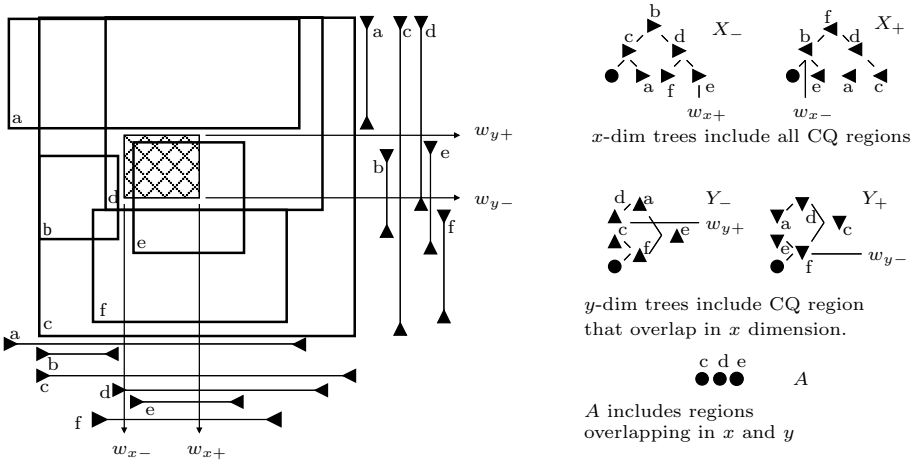


Fig. 2. Dynamic Cascade Tree (DCT) with six CQ regions, $a, b, \ldots, f$. Shown are the indexed regions, the current data stream input window, and the cascading trees, X_-, X_+, Y_-, Y_+, A, and the window node pointers, w_{x-}, w_{x+}, w_{y-}, w_{y+} that make up the data structure. $\triangle$ and $\triangleright$ represent the minimum endpoints of the regions in x and y dimension, $\triangledown$ and $\triangleleft$ the maximum endpoints of the regions in x and y dimension.

points of each of the CQ regions for each dimension. In Fig. 2, these are denoted as X_- and X_+ for the x-dimension, and Y_- and Y_+ for the y-dimension and are termed *endpoint trees* in the DCT. A final tree, A, maintains an index of all the regions that overlap with the current data in the input stream. The figure shows these trees notionally, emphasizing their ordering and structure; the endpoint trees using one endpoint and the unique identifier r_{id} for each indexed region, and the A tree only using r_{id}. The values of the nodes contain pointers back to the regions, or the associated CQ query in the Dsms.

Which of the CQ regions are included in the trees varies with the dimension. The trees for the first dimension, x, contain the minimum (in X_-) and maximum (in X_+) endpoints for *every* CQ region. Y_- and Y_+ do not contain the endpoints of all the regions in DCT, but only *the regions with x extent that overlap the current data stream's x extent.* This is easily expanded to more dimensions, where each additional dimension adds another set of trees to the DCT that hold the minimum/maximum endpoints of the CQ regions in this new dimension. Again, each of these new trees only indexes those regions that overlap the current window up to that dimension.

A is the final tree and contains all the regions that overlap with the current window in all dimensions. Just as the trees in the y-dimension contain only a subset of the regions that are indexed in the x-dimension, A contains the subset of the regions where the y-dimension of the window and the CQ regions overlap. These tree structures in each dimension make a cascade of indices, each a subset of the previous index.

In addition, pointers are maintained identifying where the current window is located. This is accomplished by pointing to nodes within each of the endpoint trees for each of the dimensions in the DCT. The pointers match the closest endpoint with a value less than or equal to the current window location. These are pointers to existing nodes and not the actual values of the window endpoints themselves. The nodes correspond to line segments and these pointers identify regions where the current result set is still valid.

Figure 2 shows these pointers in each dimension. They are designated as w_{x-} and w_{x+} for the x-dimension, and w_{y-} and w_{y+} for the y-dimension. Note that the minimum window pointers, for example w_{x-}, point into the tree of maximum endpoints for the CQ regions. Similarly, the maximum window pointers are in the trees of minimum endpoints. By moving these pointers along their corresponding trees, the DCT constructs a new result set of CQ regions from the previous result set for a new input window. A more detailed explanation is given in Sect. 2.4. In short, imagine as a spatial extent grows in size, new CQ regions are added to the result set as the maximum edge of the extent crosses the minimum edge of new regions. Similarly, regions would be added as the data extent minimum crosses new region maximum edges. The same idea applies for shrinking edges. Tracking the movement of the spatial extent of the incoming data stream from input window to input window is how the DCT incrementally maintains a result set of intersecting CQ regions.

The endpoint trees of the DCT are implemented using simple binary tree structures that support insertion, deletion, and iteration of nodes in both forward and reverse directions. The keys for each node in the endpoint trees are made up of two values, one corresponding to an endpoint value for each CQ region in one dimension, and one corresponding to a unique identifier for each region, r_{id}. The two values are combined so that spatial order is maintained. Different CQ regions that share an endpoint value are differentiated by the r_{id}'s. Individual nodes correspond to the half-open line segments between two endpoints in a single dimension. The trees all have an additional node with a minimum endpoint

that is less than any possible region, shown as a dot in the trees of Fig. 2. These allow half-open line segments to completely cover any potential location of the incoming data stream.

2.2 DCT initialization

The DCT is initialized by creating the minimum/maximum endpoint trees for each dimension of the DCT. Initial nodes for each endpoint tree are created by adding a node with a value that is less than any possible value for a region in each dimension. These are shown as dots in the trees of Fig. 2. The current window location is then created by assigning the pointers w_{x-}, w_{x+}, w_{y-}, and w_{y+} to the appropriate minimum node for each of the endpoint trees in each dimension. The A structure is initially empty.

2.3 Inserting and deleting CQ regions

For inserting a new region r, first the x-dimension endpoints are inserted into X_- and X_+. The new region is checked for overlap with the current stream window extent, that is both $w_{x-} < r_{x+}$ and $w_{x+} >= r_{x-}$ hold, where r_{x+} and r_{x-} are the maximum and minimum values of the new region in the x dimension. If the new region does overlap the current window, then the region is also added into trees Y_- and Y_+. If the new region overlaps in this dimension, $w_{y-} < r_{y+}$ and $w_{y+} >= r_{y-}$, the region is added into A using r_{id} as the index. Insertion maintains the validity of the result set A with respect to the current stream window.

Deletion is similar, following the cascade of the DCT, with one modification. The current window pointers need to be checked to verify that they are not pointing to a node that is being removed. If they are, then they need to be modified. For example, to delete region r, if w_{x+} points to this region, then decrement w_{x+} to the previous node in X_-. If w_{x-} points to the maximum endpoint of r, then decrement w_{x-} to the previous point in X_+. These changes maintain the validity for Y_-, Y_+, and A. The endpoints of r are then deleted from X_- and X_+. If the region intersects the current window, it needs to be deleted in a similar manner from Y_- and Y_+, and potentially A as well.

2.4 Queries

Figure 3 describes the changes to the DCT for new stream data (windows) with new extent. As X_- and X_+ are not changed, only the values of Y_-, Y_+, and A are shown. Figure 3(a) shows the query change due to the movement of the window in the x dimension, and 3(b) shows the change due to the movement in the y dimension. Just as the trees for each dimension and A need to be maintained when new regions are inserted and deleted, each new stream window requires maintenance of these trees as the window changes its location. These changes are made incrementally on the existing structure of the DCT.

Since Y_- and Y_+ contain regions overlapping the current window's x extent, when new data arrives with a different x-extent, Y_-,Y_+, and A need to

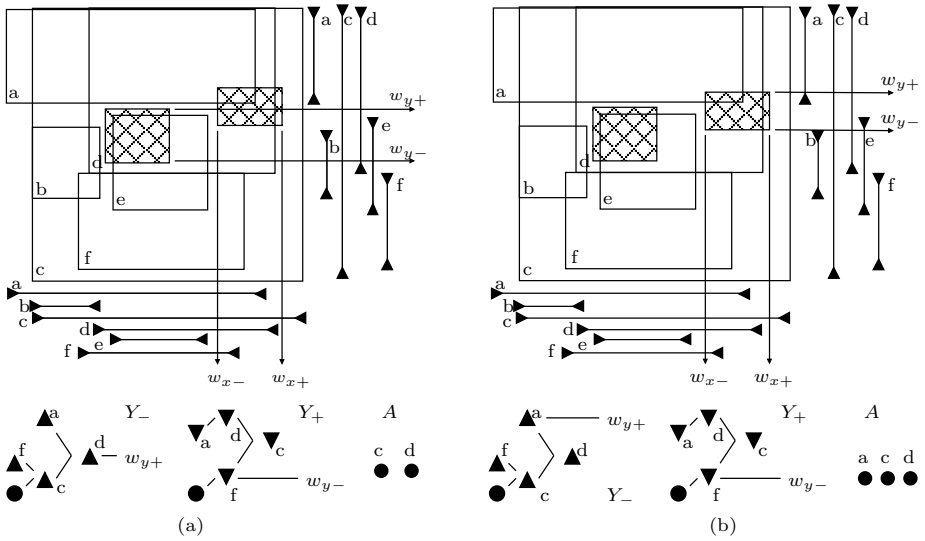


Fig. 3. Query on a new input stream window. The original window position (as in Fig. 2) is the hatched box on the left, and the region of the new window is the hatched box on the right. (a) shows the updated Y_-,Y_+, and A structures after moving the window in x dimension; (b) shows the final structures after moving in y dimension.

be modified. These modifications occur when the window region endpoints cross a boundary of a CQ region. For each x region boundary in the DCT that is crossed, the y-dimensional trees need to be modified to account for inclusion or deletion of this region. A similar method needs to be associated with boundary crossings in the y-dimension, requiring modification of A.

The algorithm for reporting overlapping CQ regions for a new window $w = \{(w_{x-}, w_{y-}), (w_{x+}, w_{y+})\}$ begins by traversing the endpoint trees in the x dimension. Figure 3(a) shows an example where the new window region has increasing minimum and maximum x edges. An increasing minimum edge implies that the movement can result in CQ regions being deleted from the result set A. The movement of w_{x-} is tracked by incrementing along the nodes of X_+. Moving w_{x-} to a new node corresponds to a boundary crossing of the window minimum with a region maximum, and in this case corresponds to one region, e, being removed from Y_- and Y_+. This deletion is cascaded to A as well. The increasing maximum, w_{x+}, corresponds to a movement that would add CQ regions. However, in this case, although the window maximum is greater, no minimum nodes are crossed, and w_{x+} points to the node that contains the minimum endpoint of the region d. The net result of moving the window in the x dimension is that region e is removed from Y_-, Y_+, and A.

Next, the movement of the window in the y dimension is accounted for as shown in Fig. 3(b). In this case, although the y minimum of the window has changed, no maximum boundaries are crossed, and w_{y-} remains pointing to the

maximum value of region f. Moving the window maximum in y, however, results in the minimum edge of region a to be crossed. w_{y+} now points to the minimum edge of a, and a is added to the result set A.

This example shows one type of movement of the window region, but the same algorithm works for all changes of the window region. The window can grow and contract on all sides, or move in any direction. The algorithm works the same way; each edge of the window query is handled separately either adding to or deleting from the subsequent trees and the final result, based on the direction of movement of the edge. There is one caveat to this algorithm for updating the DCT based on the movement of edges individually. Without modification of the above algorithm, the expansive movements of the new window need to be performed before the shrinking movements. Large movements of the window can produce movements across both edges of a CQ region. Expanding before shrinking prevents a region from being deleted first on a shrinking movement, and then erroneously inserted back on expansion. Executing deletions first, however, is preferable as they decrease the size of the trees in the DCT. To allow shrinking movements first, during the expansive movements, range checks need to verify that the region indeed belongs to the overlapping set before insertion into subsequent trees.

3 DCT Performance

The window query performance of the DCT, i.e., determining which CQ regions intersect with a new incoming stream window, is highly dependant on the location of the CQ regions, the properties of the input window, and the interaction of these parameters. For window queries over n CQ regions, with k being the result count, the execution time for window query can range from $O(k)$ in the best case, when the result set is the same as the previous set to $O(n \lg n + k)$ in the worst case, when every region is entered in a single movement of the spatial extent of the incoming window. Before looking at some experimental results, there are some simple guidelines to consider for the application of the DCT.

3.1 Region insertion and deletion cost

The DCT data structure is small and robust to many insertions and deletions of CQ regions. Insertions and deletions take $O(\lg n)$ time as the query region is potentially added to the endpoint trees in some constant dimension and A. These trees are simple to maintain dynamically in $O(n)$ space.

3.2 Query cost versus the number of boundaries crossed

The DCT is designed for trending data, which can be measured by the number of region boundary crossings from one window query to the next. The structure works best when the number of region boundaries crossed by successive input windows is not large. When no boundaries are crossed, then no internal lists are modified, and reporting CQ regions runs in $O(k)$ time. When a boundary is

crossed in movement of the window, then each region with a crossed boundary needs to be inserted into or deleted from subsequent endpoint trees. This is true for at least one set of endpoint trees or A, even for CQ regions whose domains in other dimensions do not overlap the new window and thus do not contribute to the final result. The cost of a window query in this case can be as high as $O(n \lg n + k)$, since all CQ regions could potentially be inserted into dimensional trees during one window query.

The DCT data structure indexes somewhat lazily in the sense that for insertions of new regions that do not overlap the current window, only the first dimension is indexed, and not the other dimensions. The indexing costs on window queries can be thought of as finally incurring those indexing costs. However, the problem with the DCT is that these costs can occur many times in the motion of the input stream data. Rather than indexing region values once, the DCT re-indexes a subset of points multiple times as boundaries are crossed. The hope is that many successive window queries will be in the same region with the same result, and that the low cost for those queries will make up for the extra cost of maintaining a dynamic index.

3.3 Trajectory of the trending windows

Another aspect affecting performance is the movement trajectory of the input windows. For example, consider a DCT in two dimensions as shown in Fig. 2 with trajectories that are monotonic in the x and y dimensions. In this case, CQ regions are inserted into the y endpoint trees and A at most one time, and once more potentially for deletion. The total time for maintaining the DCT then is at most $O(n \lg n)$, fixing a bound on the dynamic maintenance costs. For m window queries over that trajectory, the cost of all queries would be $O(n \lg n + mk)$ where k is the average number of results per window query. Similar savings exist for trajectories that are only generally monotonic, such as most RSI data streams.

On the other hand, more erratic window trajectories can result in poor performance. Consider a window movement that repeatedly crosses all n region boundaries. Each window query iteration would require $O(n \lg n)$ time, as the y dimensional trees are repeatedly made up and torn down.

Also, the DCT as described in the above figures, which indexes on x and then y, favors windows that trend in the y direction over windows that trend in the x direction. This is because boundary crossings in the x dimension have to modify more trees, and the trees tend to be bigger, so they take more time. In our examples, movements in the y dimension only modify A, the smallest tree in the DCT. Also, movements in the x direction result in insertions into Y_- and Y_+ that can be somewhat wasted in the sense that these CQ regions may never contribute to a final result set, whereas boundary crossings in the y direction will always affect the result set. Although, the time it takes to update the DCT due to a boundary crossing is $O(\lg n)$, for either x or y dimension, y modifications will always be faster.

This shows that order in the cascade is very important, and dimensions that see more boundary crossings for successive window queries into the DCT should

be pushed deeper into the structure. Boundary crossings are of course dependant on the trajectory of incoming windows and the organization of the regions in the *DCT*. Section 4 tests and quantifies some of these performance features.

4 Experiments

To test the performance of the *DCT*, two experimental setups were used. The first tests are on randomly moving stream data (windows) and random CQ regions, with a number of variations on specific input parameters. The second experiment was developed to more closely replicate the queries made on Dsms with GOES RSI data as input, and more realistic region parameters.

Comparisons were made to an existing in memory R*-tree implementation from the *Spatial Index Library* [9]. For better comparison, the *DCT* implementation included some components from this same library. All trees in this implementation of the *DCT* were simple set objects from the Standard Template Library (STL) [22]. The total size of this experimental *DCT* implementation was about 600 lines of C++ code. All experiments were run on a single 1.6 GHz Pentium M CPU with a 1MB L2 cache and 512MB of main memory.

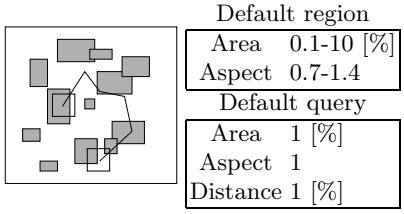
4.1 Random continuous queries with a random data stream

The more extensive tests were run using a set of CQ regions that were randomly located throughout a two dimensional space, with some variation of parameters as shown in Fig. 4(a). For most tests, the input stream window moved randomly through the region, with the default parameters shown in Fig. 4(a). The CQ regions were distributed uniformly throughout a square region. Some region parameters are modified for certain tests, these tables show the default values. Aspect is the ratio of lengths $\frac{x}{y}$ of the spatial extent of the input window. For most tests, input windows moved in a random direction from one window to the next. Sometimes, the extent of the input windows would trend off the region of interest. When this occurred, a new starting point within the region was chosen to reset the window location.

These experiments do not correspond too closely to any real world application, but are instructive in testing the performance of the *DCT* with respect to various parameters. All experiments plot the average response time for a window query as a function of a single parameter. All experiments were run using 4K and 16K CQ regions.

Figure 4(b) shows the average window query time while varying the average distance moved from on window to another. The direction of the move was randomly determined. This is one of the most important parameters to consider when using the *DCT*. The windows must come close to one another for the index to work effectively. As expected, while the R*-tree is insensitive to this parameter, the *DCT* performance is very dependant on the distance moved for a window. As the distance between windows increases, the amount of information that can be shared in the trees of the *DCT* becomes less and less. For data in the input stream at a size of 1% of the total area, a move over a distance of

2% virtually eliminates all sharing of information between window queries. At what point the distance between queries becomes limiting is dependant on other application parameters. For example, as more regions are added into the DCT, more boundaries are likely to be crossed over the same distance moved.



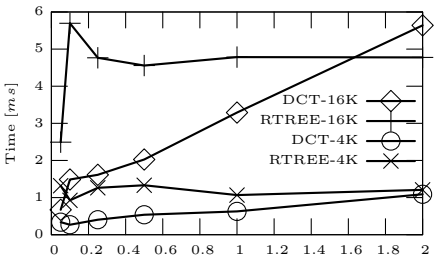
Default region

Area	0.1-10 [%]
Aspect	0.7-1.4

Default query

Area	1 [%]
Aspect	1
Distance	1 [%]

(a) Parameters



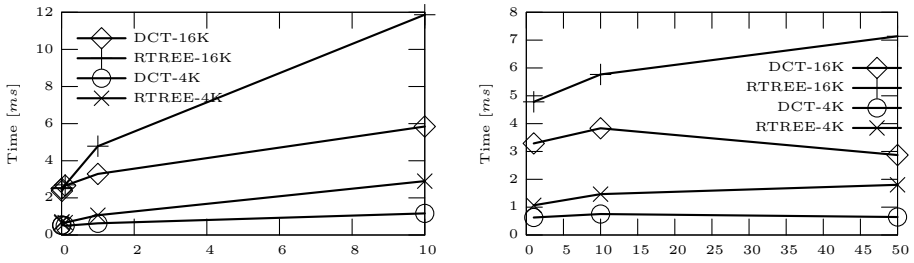
(b) Distance Moved [%]

Fig. 4. (a) Random test parameters for stream input windows and query regions; (b) average query time as function of the distance between subsequent windows and window queries.

The size of the stream input window spatial extent also affects the performance of the DCT. In general, windows with a larger spatial extent would be expected to share more results from query to query and therefore improve the performance of the DCT. Figure 5(a) compares the DCT to the R*-tree for different incoming input window extent sizes. The average response time increases for both implementations, but less so for the DCT. For this experiment, the result set for the window queries increases with window query size as well, so it is expected that the response time increases with stream data extent. The DCT results are somewhat artificially high for another reason. As was described, the query window is relocated to a new random location when the window trends off the experiment's region. Since this is more likely to occur with a larger extent, more of these relocations occur in those experiments, and since each relocation corresponds to larger query distance movements in the DCT, this increases the average distance between data in the stream.

As described in the introduction, the intent of the DCT is for use in streaming RSI, and these usually entail incoming data packets of a row, or small number of rows of data. Since these correspond to the extents of the individual data in the incoming geospatial data stream, DCT queries can have a high aspect ratios, that is, $\frac{x}{y} \gg 1$. Figure 5(b) shows the change in response time as a function of aspect ratio. The R*-tree performance is slightly degraded as the aspect ratio increases, while the DCT performance remains insensitive to this parameter.

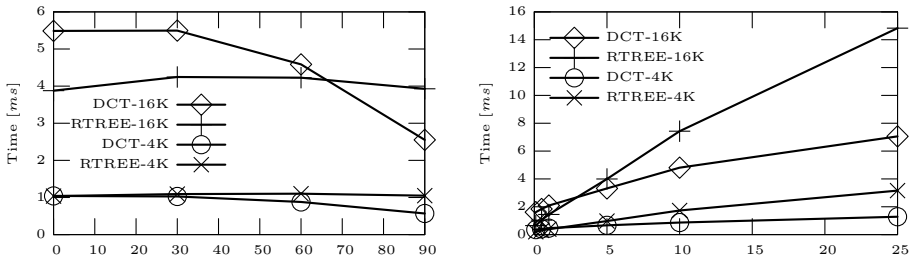
In Sect. 3, we described how the DCT can be affected by the average trajectory of the successive data (windows) from the stream. Performance is expected to improve as the window trajectory aligns with the dimensions farther down



(a) Window Query Area, [%] (b) Window Query Aspect $[\frac{x}{y}]$

Fig. 5. Average window query times as a function of area of the window (a) and aspect ratio of spatial extent of the stream windows (b).

in the *DCT* cascade. Figure 6(a) illustrates this gain. As expected, the R*-tree is insensitive to this parameter, whereas the *DCT*'s performance increases by a factor of two based on the trajectory of the window.



(a) Trajectory Direction [deg] (b) Input Area [%]

Fig. 6. Average query times as a function of trajectory of the input window (a), and size of CQ regions (b).

Figure 6(b) shows the affect of the average size of the CQ regions on the *DCT* performance. As the size of the regions increases, the average response time increases for both search structures, due in part to the fact that more regions overlap with each individual window. The R*-tree slows down more than the *DCT*, since more of the results from successive window queries can be shared from query to query. This is an important consideration for applications where the CQ regions are large.

In summary, the *DCT* performance is affected by the number of region boundaries crossed for successive window queries, and the trajectory of the win-

dow. Also, the size of the CQ regions and input data extent affect performance. Often the performance changes are different than seen in the more typical R*-tree. How the DCT performs under more realistic situations is described next.

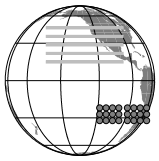
4.2 GOES Experiment

From the discussion of the DCT performance and the experimental results for random windows and regions, the parameters associated with higher performance of the DCT can be anticipated. These include: (1) relatively large extents of data in the stream, possibly with a high aspect ratio, (2) small distances from data to data in the stream, and (3) a regular trajectory of the data in the stream, with the DCT tuned to that direction. The DCT also works well with large CQ regions, having a relatively high selectivity. These are all aspects of a typical satellite RSI data stream.

For a more realistic scenario for the DCT, an experiment using queries for weather satellite data from the National Oceanic and Atmospheric Administration (NOAA) GOES satellite [7] was developed. Figure 7(a) gives an overview of the GOES sensors. The GOES satellite continuously scans a hemisphere of the Earth with two sensors, the Imager and the Sounder. Imager data arrives row-by-row, whereas Sounder data arrives more in a pixel-by-pixel manner. The experiment uses Imager data as input. GOES offers a continuous stream of data for regions ranging from the continental United States to a hemisphere centered near Hawaii. Each complete image at some time is termed a frame. Data from the GOES visible channel comes in blocks that contain 8 rows of data. Other channels from the GOES imager come in blocks of 1 or 2 rows. The number of columns in a row varies from frame to frame. An entire frame of data is reported 8 rows at a time from North to South. New frames start from their most northern extent. On average there are about 5125 rows per frame, which means that for every frame start that requires a long traverse through the DCT structure, there are about 640 small steps downward in the y direction. Query regions also tend to be approximately square regions covering relatively large regions.

GOES streaming RSI data is well suited to the DCT. In each frame, the data trends only in the downward direction and incrementally. Therefore, endpoints are only added into the X_-, X_+ structures one time, limiting the maintenance time to $O(n \lg n)$ for each frame. For normal GOES data, the starting column of each row does not change within a frame, and the Y_-, Y_+, and A structures are the only structures that change in determining which regions overlap any given set of incoming streaming rows of data.

For the experiment, 5000 regions were indexed. Rather than randomly locating these regions throughout the image domain, they were preferentially located around a small number of "hot spots" in the hemisphere. This more closely relates to real world scenarios, where specific parts of the RSI data are requested by a large numbers of users. Although the general area of the queries was fixed to a small number of locations, the individual region centers, total sizes and aspect ratios were varied for each region. This corresponds to many users requesting slightly different particulars for a general region of interest. A small fraction of



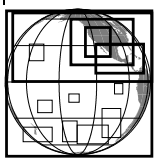
Imager
4 $[km^2]$ pixels
5 bands
1-8 rows at a time
Data arrives by row(s)
$[2,500 - 17,000]^2$ size

Sounder
19 Spectral bands
64 $[km^2]$ pixels

(a) GOES satellite scans

Contributions
30% North America
15% West Coast
15% Northern Hemi
15% Hemisphere
9% California
9% Mexico
2% Random (1K x 1K)
5% Random (6K x 4K)
from 5000 regions

Region Variations
Area 0.8-1.2
Aspect ±10%
Center ±10%



(b) Query region statistics

Fig. 7. GOES based *DCT* experiment

random regions was also included in the experimental setup. Figure 7(b) describes the parameters for the CQ regions. The table shows the various regions and their contributions to the total regions. The other table shows the variation of the individual regions. The figure graphically depicts the experimental setup.

10,000 window queries were performed on the CQ regions. The RSI data corresponding to the input data stream were taken from a sample of the GOES West Imager instrument, each query corresponding to 8 rows of data. Depending on some assumptions about other aspects of a Dsms indexing GOES data, this corresponds to somewhere between 7.5 to 60 minutes of streaming time.

Again, the *DCT* was compared to the in-memory R*-tree implementation. The R*-tree implementation required 48 seconds to perform all the queries, for an average query time of 4800 $[\mu s]$. The *DCT* performed the queries in 9 seconds, 900 $[\mu s]$ per query. It is also interesting to note that of those nine seconds for the *DCT*, only 0.88 seconds where used in the maintenance of the *DCT* and its structures. Nearly all the other 8.22 seconds involved copying the final result *A* structure into new lists for output. This was done to match the output of the R*-tree, but scenarios can be envisioned where applications access the *A* structure directly after a query, reducing this overhead cost.

5 Related Work

This work is an extension of a more simple Dynamic Cascade Tree (*DCT*), first proposed to efficiently restrict streaming spatial data *points* to specified regions of interest [10]. That structure defined the problem as solving many *stabbing point* queries, where the incoming data stream was a single moving point used for each query. The *DCT* proposed in this paper aims to allow many spatial regions to be evaluated simultaneously as the stream of input spatial data arrives in the Dsms. The obvious application of the *DCT* is in multi-query evaluation, where a single operator uses the *DCT* structure to perform the spatial restrictions for

a large number of queries, see Fig. 1(c). Similar multi-query optimizations have been discussed in streaming databases where they have been described as *group filters* [15, 16] and in spatio-temporal databases where this optimization has been described as *query indexing* [19].

Many data structures have been developed for one and two dimensional window queries including among others, interval trees, priority search trees, and segment trees [4, 20]. Space partitioning methods of answering window queries include quadtrees, hashes, and numerous variants of R-trees.

The common methods for solving window queries in two dimensions are multi-level segment trees [4], interval trees [20], and R-trees [8]. It is difficult to modify these methods to take advantage of trending data. 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 query times of $O(\lg 2n)$. Dynamic maintenance of such a structure is more complicated and requires larger storage costs [14]. It is difficult to modify the multi-level segment tree to improve results for trending data. If the input window moves a small distance, which does not change the query result, 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 result occurred.

Interval trees yield an optimal worst-case space and query time solution to the window query problem, $O(n)$ in space and $O(n \lg n + k)$ in query time, where k is the number of regions intersected with the given region. However, interval trees are static structures, because all the region intervals need to be known in advance to construct the trees. Also, like segment trees, it is difficult to modify interval trees to take advantage of data trendiness.

R-trees solve window queries by traversing through minimum bounding rectangles that include the extent of all regions in the sub-tree, generally with good performance. Since these rectangle regions generally overlap, there can be no savings from knowing the previous window, as there is no way to know if an entirely new path through the segment tree needs to be traversed. R^+-trees [21] may be modified for trending windows, since the minimum bounding rectangles are not allowed to overlap. Thus, maintaining the previous window can help to verify a window query that has not left a particular region. R^+-trees, however, have problems with redundant storage and dynamic updates [17].

Another approach similar to the *DCT* described in this paper is the Query Index [12, 19]. The Query Index builds a space partitioning index on a set of static query regions and at each time interval, allows a number of moving objects to probe the index to determine overlapping queries. Experiments with main memory implementations show that grid-based hashing of query regions generally outperform R-tree or quad-tree based methods. The query index is most effective for small regions and query points, and experiments show performance degradation with larger regions or high selectivity query windows [13].

The *DCT* is also an incremental approach to answering queries where updates are made to a current query result. SINA [18] describes an incremental method to solving the problem of intersecting moving objects, though the approach focuses on efficient integration of incremental query changes with disk-based static queries, and a complete main-memory implementation would be more similar to the query index approach.

In another sense, the *DCT* is a method of dynamically maintaining boundaries around a current window where the current result set is valid, and identifying where this result set will change. Another method of dynamically describing a neighborhood of validity for a query using R-trees was proposed in [23], which builds an explicit region of validity around a current query point. This method is meant to minimize transmission costs to a client, and the technique makes many additional queries to an R-tree style index to build these regions of validity.

6 Conclusions and Future Work

Remotely-sensed imagery clearly provides a great opportunity to study concepts and paradigms for the management and processing of streaming data, given the existence of various satellites that are used constantly for numerous data products in environmental sciences, disaster management, climatology etc.

In this paper, we have presented the Dynamic Cascade Tree (*DCT*), which is suitable for window queries that are most common to most of RSI data streams. This simple data structure is designed to work especially well for streaming data with spatial extents that are in close proximity and follow certain trendiness in their movement. These are characteristics of solving multi-query optimizations for streaming RSI data, for which the *DCT* was developed. Performance evaluation and experiments over random data identified some strengths and weaknesses of the *DCT* in more general situations, while an experiment using GOES weather imagery demonstrated its suitability for streaming RSI database applications.

The *DCT* is part of query processing architecture being developed to support complex continuous queries over streams of remotely-sensed geospatial image data and will be an important component to facilitate the optimization of multiple restriction queries against such a stream.

Acknowledgment This work was partially supported by the National Science Foundation under Grant No. IIS-0326517.

References

1. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. The VLDB Journal **12**(2) (2003) 120–139
2. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom:, J.: Models and issues in data stream systems. In: Proc. 21th ACM Symposium on Principles of Database Systems (PODS). (2002) 1–16
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. The VLDB Journal **13**(4) (2004) 335–353

4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Springer Verlag (2000)
5. Carney, D., Cetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: Proceedings of 29th VLDB Conference (2003) 838–849
6. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: First Biennial Conference on Innovative Data Systems Research (CIDR 2003) (2003)
7. GOES I-M DataBook. Space Systems-Loral `rsd.gsfc.nasa.gov/goes/text/goes.databook.html` (2001)
8. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD Int. Conf. on Management of Data (1984) 47–57
9. Hadjieleftheriou, M.: Spatial Index Library. Department of Computer Science and Engineering, University of California, Riverside. version 0.80b edn. (2004)
10. Hart, Q., Gertz, M.: Indexing query regions for streaming geospatial data. In: 2nd Workshop on Spatio-temporal Database Management (STDBM'04) (2004)
11. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive query processing: Technology in evolution. IEEE Data Eng. Bulletin **23** (2000) 7–18
12. Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E.: Main memory evaluation of monitoring queries over moving objects. Distributed and Parallel Databases **15**(2) (2004) 117–135
13. Kim, K., Cha, S.K., Kwon, K.: Optimizing multidimensional index trees for main memory access. In: Proc. of the ACM SIGMOD International Conference on Management of Data (2001) 139–150
14. van Kreveld, M.J., Overmars, M.K.: Concatenable segment trees. Technical report, Rijksuniversiteit Utrecht (1988)
15. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proc. of the ACM SIGMOD International Conference on Management of Data (2002) 49–60
16. Madden, S., Franklin, M.J.: Fjording the stream: An architecture for queries over streaming sensor data. In: Intern. Conf. on Data Engineering (2002) 555–566
17. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-trees have grown everywhere. Unpublished Technical Report, www.rtreeportal.org/pubs/MNPT03.pdf (2003)
18. Mokbel, M.F., Xiong, X., Aref, W.G.: SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In: Proc. of the ACM SIGMOD International Conference on Management of Data (2004) 623–634
19. Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W., Hambrusch, S.: Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. IEEE Trans. on Computers **51**(10) (2002) 1124–1140
20. Samet, H.: Hierarchical representations of collections of small rectangles. ACM Comput. Surv. **20**(4) (1988) 271–309
21. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R+-tree: A dynamic index for multi-dimensional objects. In Proceedings of 13th International Conference on Very Large Data Bases (1987) 507–518
22. SGI: Standard Template Library Programmer's Guide. (1999)
23. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: Proceedings of the ACM SIGMOD International Conference on on Management of Data (2003) 443–454