Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Kd-tree and quad-tree decompositions for declustering of 2D range queries over uncertain space

Ahmet SAYAR, Süleyman EKEN[‡], Okan ÖZTÜRK
(*Department of Computer Engineering, Kocaeli University, Kocaeli 41380, Turkey*)
E-mail: ahmet.sayar@kocaeli.edu.tr; suleyman.eken@kocaeli.edu.tr; ozz.okn@gmail.com

Received May 5, 2014; Revision accepted Oct. 11, 2014; Crosschecked Jan. 5, 2015

Abstract: We present a study to show the possibility of using two well-known space partitioning and indexing techniques, kd trees and quad trees, in declustering applications to increase input/output (I/O) parallelization and reduce spatial data processing times. This parallelization enables time-consuming computational geometry algorithms to be applied efficiently to big spatial data rendering and querying. The key challenge is how to balance the spatial processing load across a large number of worker nodes, given significant performance heterogeneity in nodes and processing skews in the workload.

Key words: Kd tree, Quad tree, Space partitioning, Spatial indexing, Range queries, Query optimization
doi:10.1631/FITEE.1400165 **Document code:** A **CLC number:** TP391

1 Introduction

Data analysis is a crucial step in understanding and solving problems in many different applications and scientific areas. Analyses involve extracting the data of interest from all available data sources, and can occur at several stages along a data processing pipeline ranging from raw data to advanced data products. However, processes of data analysis can be hindered by huge data sets. Increasing amounts of data inhibit efficient access to the data and the management of potentially heterogeneous system resources for data processing. For this reason, subsetting and aggregation are widespread techniques used in intensive large-scale scientific data applications (Furht and Escalante, 2011). First, subsets are calculated and distributed to replica nodes and then the results are aggregated to create the response to the main query set. For good data distribution, we should consider evenly distributing the workload over processors to maximize parallelism and minimize communication cost. Yet, due to the stringent characteristics and dynamic nature of data, performing efficient load balancing and parallel processing over an unpredictable workload is not easy (Chou and Abraham, 1982). The work is divided into independent smaller pieces, even if they have the same range size, and the size of the pieces is highly variable. Creating subsets for uniformly distributed data, such as raster images, is straightforward. In contrast, creating subsets for non-uniformly distributed data is cumbersome, for instance, for vector data defined with (x, y) coordinates on a 2D plane. In such cases, sub-setting most probably results in data and execution skews.

The aim of our study was to evaluate the feasibility and efficiency of well-known space partitioning approaches (kd trees and quad trees) (Samet, 2006) in declustering (Moon and Saltz, 1998) applications. Declustering is a process used in distributed computing or clustered computing environments to reduce processing and query execution times by applying load balancing and workload sharing

‡ Corresponding author
ORCID: Ahmet SAYAR, http://orcid.org/0000-0002-6335-459X; Süleyman EKEN, http://orcid.org/0000-0001-9488-908X

approaches. The workload is decomposed into smaller pieces and shared among multiple processors or computation nodes. However, usually parallelization and workload sharing do not give the expected performance gain because the workload is inefficiently and evenly shared among the worker nodes. For the spatial domain, the problem is presented in more detail in Section 3. In this paper, we evaluate some approaches for solving this problem by using space partitioning techniques, and we test their efficiencies using 2D range queries. Both kd trees and quad trees promise some advantages over naive (straightforward) space partitioning such as smaller response times, especially in systems with a high degree of parallelism, and efficient usage of clusters through load balancing. However, it is not easy to say which is better and more efficient in this context. The most efficient approach is expected to eliminate data and execution skews for efficient input/output (I/O) parallelization.

To address data and execution skew problems in uncertain space (Chilès and Delfiner, 2009), we have used some space partitioning techniques and evaluated their efficiencies through performance tests on 2D range queries applied to point data. Query efficiency depends on how well data are distributed (balanced) across the storage nodes and how compatible the index structures are with the data characteristics and query attributes. The evaluation technique is based on creating a spatial index table for the uncertain data sets, and then applying random range queries in the space. This technique takes dense and sparse data regions into consideration. The aims are to create efficient indexing over uncertain space, and divide the workload for a range query into sub-ranges carrying equal sizes of query payload.

2 Related work

The representation and analysis of multidimensional data are very important in various fields, including database management systems (e.g., spatial databases, multimedia databases), computer graphics, game programming, computer vision, geographic information systems (GIS), and pattern recognition. The key idea in choosing an appropriate representation is to facilitate operations such as search (query). This means that the representation involves sorting the data in some manner to make it more accessible. Indexing is a technique built on data structures and algorithms (Samet, 2006).

Indexing is a well-known and widely used approach to the optimization of query execution time (DeWitt and Gray, 1992) for non-uniformly distributed data. Indexing keeps metadata for data and enables efficient searching and extraction for a given query. There are some related projects using index structures in declustering applications. Beynon *et al.* (2002) introduced a technique for both sub-setting and aggregation of results to subsets in the domain of parallel queries. They focused on optimizing query response times by ordinary striping of uniformly distributed data, i.e., continuous data, such as images. However, their technique might not give the expected performance gains when using non-uniformly distributed data, such as earth related vector data (census data, earthquake seismic data, etc.). The Seti (Chakka *et al.*, 2003) and TrajStore (Cudre-Mauroux *et al.*, 2010) projects are another two examples of related works. They concentrate on trajectory data, take dense and sparse regions into consideration, and use a two-level index structure. Zhang *et al.* (2010) introduced an index structure for efficient 2D range queries of uncertain data. They proposed an inverted index based on R-tree data structure, named UI-tree. Ray *et al.* (2013) proposed a declustering technique called Niharika that creates balanced spatial partitions. They exploited multiple cores in modern processors to improve spatial join performance. Zhong *et al.* (2012) proposed a two-tiered index structure for distributed spatial data analysis. They used the concept of geographic proximity. One tier is called a global index and is based on quad trees, and the other tier is called a local index and is based on Hilbert ordering. Data blocks are found by using the global index, and spatial objects by using the local index.

In an earlier work (Sayar *et al.*, 2014), we proposed a distributed framework for adaptive range query optimization aimed at increasing I/O parallelization over unpredictable workloads. Query size and distribution characteristics of data (data dense/ sparse regions) in varying ranges are not known a priori, and performing efficient load balancing and parallel processing over the unpredictable workload is achieved by using kd trees based space partitioning tree indexing. In another project, we developed a fine grained federation of geographic information web services to increase the performance of spatial data in querying and rendering (Sayar, 2013). The related projects mentioned above use either kd-tree, quad-tree, or R-tree spatial indexing techniques (Wei, 2010) or combinations of them. However, to our knowledge no previous study has analyzed the efficiencies of those approaches in declustering applications. In this study, we present a technique and scenarios to analyze their efficiencies in 2D space and when applied to non-uniformly distributed point data sets.

3 Problem definition

The classical approach for dividing the workload into smaller pieces and assigning them to worker processors does not always help to improve performance in uncertain spaces. Managing and manipulating uncertainty in spatial databases are important problems for various practical applications of geographic information systems (Li *et al.*, 2007; Wang *et al.*, 2013; Reich *et al.*, 2014). Occurrences of data and their features, such as locations, cannot be estimated or modeled. Depending on the time and other factors, data can change both their geographic (locations) and non-geographic features. The work presented in this study handles uncertainty in geographic locations of data. Uncertainty hinders the creation of an efficient indexing structure and also, pertinent to this study, causes data and execution skews.

The aim is to eliminate data and execution skews for efficient I/O parallelization in uncertain spaces. Data sets are defined in a 2D plane as points, and are queried by 2D range queries. Range queries are also called window queries and defined with rectangles. They are used mostly for regional selections. A range query is a general process to analyze and display spatial data defined by their coordinates in space, and is used in many science and application domains including GIS, astronomy, CAD/CAM, and computer graphics.

Fig. 1 illustrates the problem regarding efficient partitioning of ranges for non-uniformly distributed data. The spatial data are defined/queried with 2D Cartesian coordinates (x, y). The set of (x, y) coordinate values is accessed with range queries. Ranges are called minimum bounding rectangles (MBRs) or bounding boxes, which are the same and formulated as R=[($x_{\min}$, $y_{\min}$), ($x_{\max}$, $y_{\max}$)]. Term ($x_{\min}$, $y_{\min}$) refers to the lower left corner, and ($x_{\max}$, $y_{\max}$) the upper right corner. Terms $x_{\min}$, $y_{\min}$, $x_{\max}$, and $y_{\max}$ are assumed to be integers or rational numbers.

Consider some data as a subset of 2D rational space, Q^2. There are unlimited numbers of rectangles

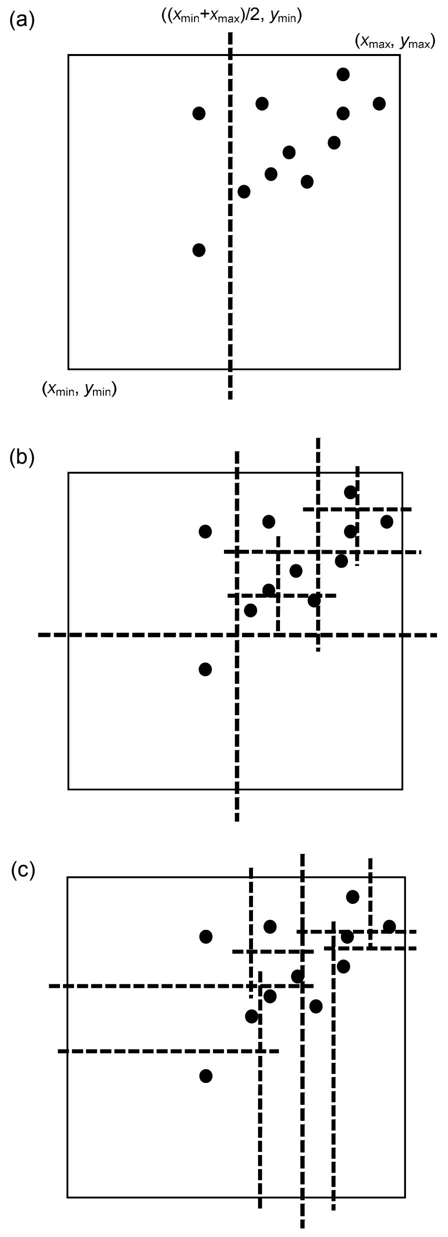


Fig. 1 An illustration of the problem with (a) naive partitioning, (b) quad-tree partitioning, (c) kd-tree partitioning

(i.e., ranges) that can be defined over the data space R. Let data space R be partitioned into non-overlapping rectangular regions $\{r_1, r_2, \ldots, r_n\}$ where $r_i=[(x_i^1, y_i^1), (x_i^2, y_i^2)]$ $(1\leq i\leq n)$. The number n changes depending on the average size of the r_i set. As the average size of r_i increases, n decreases, and vice versa.

If the area of an input rectangle (r_i) is denoted by Area(r_i), then

$$\text{Area}(R)=\sum_{i=1}^{n}\text{Area}(r_i). \tag{1}$$

The area of each partition (rectangle (r_i)) can be calculated as

$$\text{Area}(r_i)=(x_i^2-x_i^1)(y_i^2-y_i^1). \tag{2}$$

The ordinary binary partitioning (Fig. 1a) gives the best results for maximizing the performance in parallel queries if the data are uniformly distributed. In a uniformly distributed data set, query payloads can be calculated from their range sizes, i.e., Area(r_i). In such cases, the sizes of partitions (Area(r_i)) are proportional to their corresponding payloads, and cutting the spaces into equally sized partitions would result in the most efficient declustering. However, in the spatial domain, optimal partitioning of such data is difficult to achieve because polygons, line strings, points, etc. are neither uniformly distributed nor of similar size.

The research problem can be illustrated by a sample scenario. Consider two replicated data servers serving replicated data whose population in 2D space is illustrated in Fig. 1. When we do a naive partitioning (Fig. 1a), we assign the partitions to the replica servers in a round-robin fashion. In this scenario, the first replica server receives two points of load and the second receives nine points of load. This is not a fair sharing of the workload. On the other hand, optimal partitioning is expected to end up with a workload sharing where each replica server receives an equal number of points to serve, i.e., five or six points. Even for such a small data set, performance gain would be almost twofold.

It seems that workload sharing in both cases (Figs. 1b and 1c) is the same and optimal. However, the cost of each approach changes depending on the distribution characteristics of the point data, the size of the range query, etc. In this paper, we evaluate the efficiencies of kd-tree and quad-tree decompostions in declustering of point data in a distributed environment in terms of increasing I/O parallelism, by performing efficient workload sharing among the replicated servers. The efficiencies of kd-tree and quad-tree approaches are evaluated based on their costs.

4 Analysis of kd-tree and quad-tree decompositions in declustering of uncertain space

4.1 Some information about quad-tree and kd-tree partitioning

The problem mentioned in Section 3 can be solved by using space partitioning techniques (Fig. 1b). Space partitioning is a well-known technique to divide a space into smaller convex subspaces by hyper planes. The subdivisions and their resulting sub-regions are represented by tree structures. In this process, sub-regions are recursively partitioned into smaller sub-regions, and this goes on till all sub-regions satisfy one or more predefined requirements. This subdivision gives rise to a representation of objects within the space by means of a tree data structure. Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. There are two general classes of space partitioning techniques, which are also used in spatial data indexing. These are kd-tree and quad-tree approaches. In this section, we will analyze and evaluate their feasibilities and efficiencies while solving the problem mentioned here. In other words, we are evaluating these two indexing approaches in terms of their efficiencies in I/O parallelization and load balancing in declustering applications.

A quad tree is an algorithmic solution approach for space partitioning problems. It recursively divides a space into 2^2 subspaces, which are also called quadrants. Quad trees are of three types: region quad trees, point quad trees, and point-region (PR) quad trees (Wei, 2010). The region quad tree was the first to be developed. It is based on a recursive regular decomposition of a space into four equal sized subspaces. Recursive calls stop when the area of the sub-regions becomes small enough. In the case of PR quad trees, again the spaces are recursively decomposed into 2^2 quadrants. The regions are assumed to be filled with point data. Therefore, recursive calls stop when the number of points in each quadrant reaches a limit known as node capacity. Finally, in the case of a point quad tree, again space is decomposed into 2^2 sub-regions recursively. Recursive calls keep running until each sub-region contains at most one point. In this study, we preferred to use the PR quad-tree approach for indexing and declustering point data (Sinha *et al.*, 2010). The pseudo code for the algorithm to create a point-region quad tree is given in Algorithm 1.

Algorithm 1 Building a point-region quad tree

```
Procedure ConstructRegionQuadTree($x_{\min}, y_{\min}, x_{\max}, y_{\max}$)
Input: A number of points and parameters that indicate the
  starting ($x_{\min}, y_{\min}$) and ending ($x_{\max}, y_{\max}$) coordinates of
  region
begin
  while some quadrant contains more than 1 point do
    if the top left quadrant contains more than 1 point then
      ConstructRegionQuadTree($x_{\min}$, $(y_{\min}+y_{\max})/2$,
        $(x_{\min}+x_{\max})/2$, $y_{\max}$)
    else if the top right quadrant contains more than 1 point
      then
      ConstructRegionQuadTree($(x_{\min}+x_{\max})/2$,
        $(y_{\min}+y_{\max})/2$, $x_{\max}$, $y_{\max}$)
    else if the bottom left quadrant contains more than 1
      point then
      ConstructRegionQuadTree($x_{\min}$, $y_{\min}$,
        $(x_{\min}+x_{\max})/2$, $(y_{\min}+y_{\max})/2$)
    else if the bottom right quadrant contains more than 1
      point then
      ConstructRegionQuadTree($(x_{\min}+x_{\max})/2$, $y_{\min}$,
        $x_{\max}$, $(y_{\min}+y_{\max})/2$)
    end if
  end while
end
```

A kd tree is a data structure for indexing *k*-dimensional point data distributed in a *k*-dimensional space. A kd tree can be considered as a *k*-dimensional binary search tree (Bentley, 1975). Kd trees can also be considered as algorithmic solutions to the space partitioning problem. Internal nodes of kd trees provide two types of information: coordinates of point data and rectangular representations of corresponding sub-regions. The latter are derived from the former using the properties of kd trees. The root node represents the whole region of interest. A kd tree is created by recursively dividing regions into two smaller sub-regions along the *x*-axis and *y*-axis alternately. The division is done at the median points along the axes. The pseudo code for the algorithm to create a kd tree is given in Algorithm 2.

Algorithm 2 Building a kd tree

```
Procedure ConstructKdTree($P$, Counter)
Input: A number of points $P$ and a counter set to zero (0)
begin
  if $P$ equals one (region contains only one point) then
    there is no need to split region any more
  else
    while some sub-region contains more than 1 point do
      if Counter is even then
        Cut $P$ into two pieces (subsets) at the median
          point along the $x$-coordinate with a vertical
          line. The cutting point is called $x$-median. Sets
          of points to the left and to the right of $x$-
          median are called $P_1$ and $P_2$, respectively
        Increase Counter by 1
      else
        Cut $P$ into two pieces (subsets) at the median
          point along the $y$-coordinate with a horizontal
          line. The cutting point is called $y$-median. Sets
          of points to the left and to the right of $y$-
          median are called $P_1$ and $P_2$, respectively
        Increase Counter by 1
      end if
      ConstructKdTree ($P_1$, Counter)
      ConstructKdTree ($P_2$, Counter)
    end while
  end if
end
```

Time and space complexities of quad-tree and kd-tree constructions are analyzed as follows:

If data points are uniformly distributed, then the whole tree structure looks like a uniform grid. The resulting tree will be balanced and the expected depth will be $\log_4 N$. Since the tree does not need to be updated, the best-case running time will be $O(N)$, where N is the number of data points. The worst case occurs when each point has to move up to the root and then down to the leaf nodes. The running time is dominated by the movement of data points. The depth of the tree will be a function of the side length of the simulation space and the closest distance between any two points. Thus, the worst-case running time will be $O((h+1)N)$, where h is the height of the quad tree. The worst-case complexity is also $O((h+1)N)$. The height of a quad tree is computed according to the height lemma as

$$h \le \lg(s/c)+3/2,$$

where s is the side length of the initial square containing all data points, and c is the shortest distance between any two points. Also, the average case equals $O(hN)$. The space complexity of the quad tree is $O(N)$.

As in the case of the point quad tree, the amount of work expended in building a kd tree is equal to the total path length of the tree, as it reflects the cost of searching for all of the elements. Bentley (1975) showed that the total path length of a kd tree built by inserting N points in random order into an initially empty tree is $O(N\log_2 N)$ and thus, the average cost of inserting a node is $O(\log_2 N)$. The extreme cases are worse since the shape of the kd tree depends on the order in which the nodes are inserted, thereby affecting the total path length. The space complexity of the kd tree again is $O(N)$.

Optimal partitioning is a process by which the computational load on each partition becomes roughly the same. The load is represented by the area of the corresponding region, represented as Area(r_i) throughout the paper. Load is mainly a cost measure to determine the sizes of partitions. The goal is to find out the most efficient number of partitions and their sizes for a given data space. The aim is to cut the data space R into smaller pieces (r_i) with an approximately equal number of points. However, the load to be assigned to the partitions for declustering applications is represented by the area of the sub-regions (r_i). So, the sum of the areas of the sub-regions (Eq. (4)) is supposed to be very close to the area of the range query. The difference is called the cost (Eq. (5)), and is the measurement used for defining the efficiencies of the partitioning approaches. Detailed cost analyses and comparisons are given in Section 4.2.

The efficiencies of kd-tree and quad-tree partitioning are measured using range queries over the point data non-uniformly distributed in a 2D plane. The main query range is positioned on the index table and overlapping sub-ranges are calculated. The output of this work is the set of rectangles overlapping with the main query range (Fig. 2). The algorithm explaining how to calculate sub-regions overlapping with query ranges is given in Algorithm 3. Partitioned sub-regions (r_i), which are all rectangles, are recorded as (x_1, x_2, y_1, y_2), i.e., ($x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$),

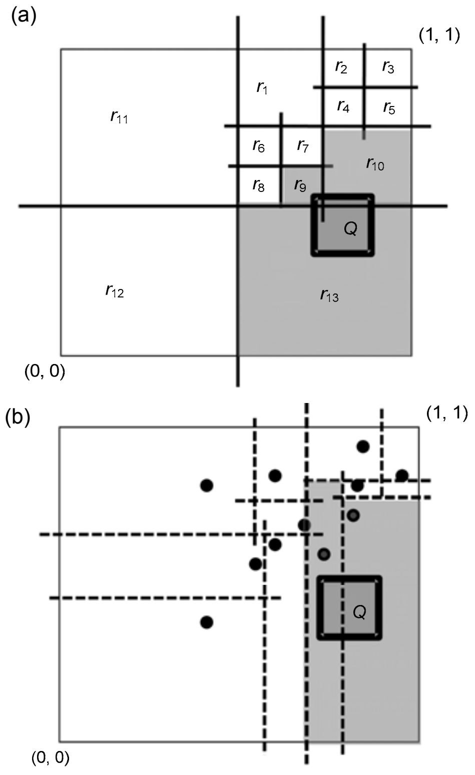


Fig. 2 Illustration of query decomposition with a sample scenario
(a) A sample range query on quad-tree indexed data; (b) A sample range query on kd-tree indexed data

Algorithm 3 Determining the overlap of the query rectangles and sub-regions

```
Procedure SEARCHINTERSECTEDREGIONS(Q, AllRectArray)
Input: Range query (Q) and sub-region array created after
        building trees (AllRectArray)
Output: Set of sub-region array (INTERSECTEDRECTARRAY),
         which is initially empty
begin
  for each rectangle (R) in all sub-region rectangles
    if !($Q.x_1>R.x_2$) and !($R.x_1>Q.x_2$) and !($Q.y_1>R.y_2$)
      and !($R.y_1>Q.y_2$) then
      Include rectangle R in INTERSECTEDRECTARRAY array
    end if
  end for
end
```

in an array. The main query (Q) is compared with each rectangle in the array to select those that are overlapping.

4.2 Cost function for comparing the two approaches

Define three sets whose elements are rectangular regions. The range query Q is a set of one element. Set R is a set of all rectangles created by one of the two indexing techniques. Set S consists of all the rectangles meeting the requirements given in Eq. (3). When we position query Q on the index table (set R) we find all the intersection rectangles. These are called set S, as described below:

$$S=Q\cap R=\{r_i: (r_i\in Q)\wedge(r_i\in R)\}, \tag{3}$$

where $i\leq n$ and n is the number of rectangles in R.

The query is given in a rectangular format. So, it is a trivial calculation to compute the area of the query, Area(Q) (Eq. (2)). The sum of the area of rectangles intersecting with Q is calculated using

$$\text{Area}(S)=\sum_{r_i\in S}\text{Area}(r_i). \tag{4}$$

The ideal value for the sum of the areas of rectangles intersecting with query Q is expected to equal Area(Q). The difference is called the cost. The motivation behind selecting the cost metric given in Eq. (4) is explained as follows:

$$\text{cost}=\text{Area}(S)-\text{Area}(Q). \tag{5}$$

Let us illustrate this with a sample scenario (Fig. 2). Figs. 2a and 2b show index tables obtained after applying quad-tree and kd-tree algorithms, respectively. There are 13 sub-partitions (r_i, $1\leq i\leq 13$) in the quad-tree index table and 11 in the kd-tree index table. We also assume that the application does not treat partial and total overlapping ranges differently. Q is the selected region represented as a range query in the figure. When a user selects rectangular region Q, and positions it on the index table, the overlapping sub-partitions are colored in grey. The algorithm for calculating overlapping regions is given in Algorithm 3. The details for calculating cost values are given below for the quad-tree case (Fig. 2a). This also applies to the kd-tree case (Fig. 2b).

Here is a simple scenario to calculate the cost with real values:

$S=\{r_9, r_{10}, r_{13}\}$, which is a set of intersected partitioned sub-regions.

$$\text{Area}(S)=\sum_{r_i\in S}\text{Area}(r_i)$$
$$=\text{Area}(r_9)+\text{Area}(r_{10})+\text{Area}(r_{13})=0.328,$$
$$\text{Area}(Q)=1/16=0.063,$$
$$\text{cost}=\text{Area}(S)-\text{Area}(Q)=0.328-0.063=0.265.$$

Note that the whole area of the data space is one, which is a unit square.

For illustration purposes, the sample space is given as a unit square (Fig. 2). However, it can be converted to any other spatial reference system and metric value. Specifically, in this study, areas of rectangles were calculated based on pixel values. The number of pixels in a given area was used as a length metric for computing query areas (Area(Q), Area(S), and Area(r_9)). For example, if a query region had a height of 40 pixels and a width of 50 pixels, then the query area was calculated as $40\times50=2000$ pixels. This approach is valid for both kd-tree and quad-tree cases. The only difference between them was the creation of their index tables, because they have different approaches for indexing point data.

The actual load falling in the query area (Q) is very small compared to the sum of the partition sets (r_9, r_{10}, and r_{13}) returned by the index table. So, the sum of grey regions in Fig. 2, but not in Q, will be defined as costs. This is because the non-overlapping parts of r_9, r_{10}, and r_{13} are assigned to the workers, but the client does not actually need them. The aim is to keep the cost at a minimum. In other words, the difference between the area of the query region and the sum of the areas of overlapping regions should be as small as possible. As the cost becomes relatively small, the success of the index table for declustering applications becomes relatively high, in terms of even workload sharing among clusters.

4.3 Evaluation and test scenarios

The graphical user interface (GUI) used for testing and evaluating the system is shown in Figs. 3 and 4. A GUI enables the creation of random point data by a random function (Math.Random()). A GUI also enables the creation of users' custom data by mouse clicks. The GUI was divided into two sections, one (left-hand side) related to kd trees and the other (right-hand side) related to quad trees. They were closely related and synchronized; i.e., the data created for one indexing was also available for the other data instantly. These two indexing approaches were tested on the same data. The area of the whole region in which data were distributed was defined as 512×512 pixels (Figs. 3 and 4). This was to be used in calculations of the areas of the query and partitioned regions. We realized experimental tests on a machine with a 2.53 GHz Intel i5 CPU, 4 GB RAM, 500 GB hard disk, and Windows 7 ultimate operating system. The application for the tests and evaluations was developed in Java programming language, version 1.6.3. The GUI was developed using Java swing libraries.

Test scenarios were determined based on the data size (relatively large or small) and distribution (random or skewed) characteristics. Randomization could be applied both to the location and to the size of the queries. Skewed data were created according to the user's preferences by using interactive GUI tools. These were customized data for test purposes.

The test scenarios are listed below:

Case 1: Large range queries over random data space (Figs. 3a and 5).

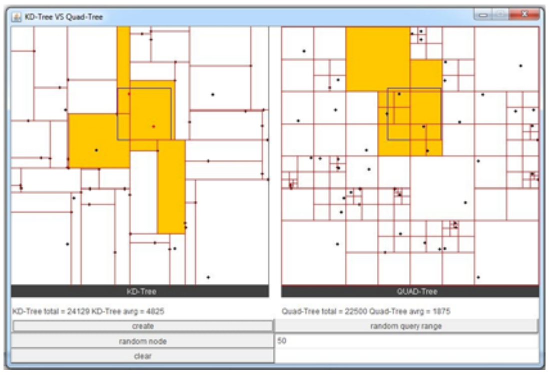
Case 2: Large range queries over skewed data space (Figs. 3b and 6).

Case 3: Small range queries over random data space (Figs. 4a and 7).

Case 4: Small range queries over skewed data space (Figs. 4b and 8).

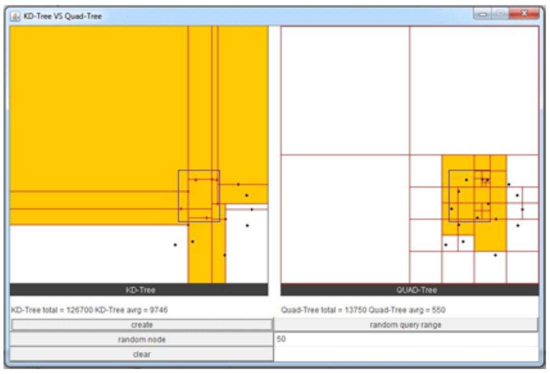
Screen shots from the GUI created for the performance tests and evaluations are shown in Figs. 3a, 3b, 4a, and 4b. The corresponding performance test results are given in the same order in Figs. 5–8.

Figs. 5 and 6 show the performance results for cases 1 and 2, respectively. Fig. 5 shows the cost values changing according to the query sizes in the



KD-Tree total = 24129 KD-Tree avrg = 4825
Quad-Tree total = 22500 Quad-Tree avrg = 1875
50

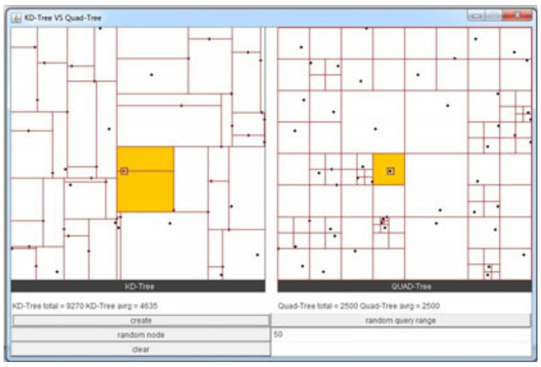
(a)



KD-Tree total = 126700 KD-Tree avrg = 9746
Quad-Tree total = 13750 Quad-Tree avrg = 550
50

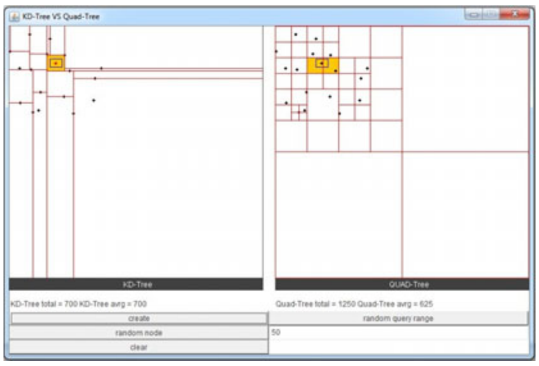
(b)

Fig. 3 Large range queries over (a) random data space, (b) skewed data space



KD-Tree total = 9270 KD-Tree avrg = 4635
Quad-Tree total = 2500 Quad-Tree avrg = 2500
50

(a)



KD-Tree total = 700 KD-Tree avrg = 700
Quad-Tree total = 1250 Quad-Tree avrg = 625
50

(b)

Fig. 4 Small range queries over (a) random data space, (b) skewed data space

case of applying relatively large sized queries to randomly distributed data sets. Fig. 6 shows the same thing for skewed data sets.

Figs. 7 and 8 show the performance results for cases 3 and 4, respectively. Fig. 7 shows the cost values changing according to the query sizes in the case of applying relatively small sized queries to randomly distributed data sets. Fig. 8 shows the same thing for skewed data sets.

The cost values tend to be smaller in the case of random data. However, they become higher in the test cases using random data. In addition, in all the test cases, point-region quad-tree indexing gives lower cost values than kd-tree indexing.

The same tests were performed on real data. We used Turkey's points of interest data (http://www.mapcruzin.com/free-turkey-arcgis-maps-shapefiles.htm) in a shapefile format. In the real world, vector data are stored mostly in Geographic Markup Language (GML) or shapefile formats for interoperability and openness. Shapefile is a popular geospatial vector data format for geographic information system software. Shapefiles spatially describe vector features such as points, lines, and polygons, representing, for example, water wells, rivers, and lakes, respectively. The shapefile used in this test shows points of interest such as monuments, attractions, ruins, and archaeological buildings. It has about 3272 data points. A plot of the data showing their distribution is shown in Fig. 9.

The screen shots from the GUI created for the performance tests and evaluations are shown in Figs. 10 and 11. They show large and small range queries, respectively, over the same data space. The

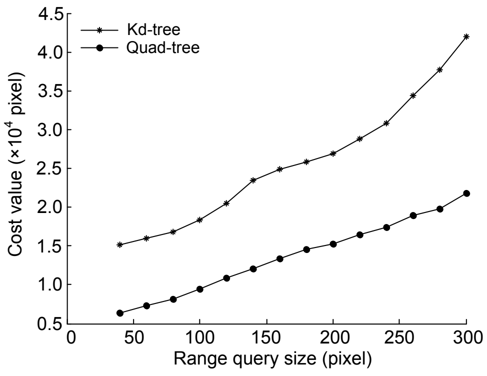


Fig. 5 Comparison of cost values for case 1

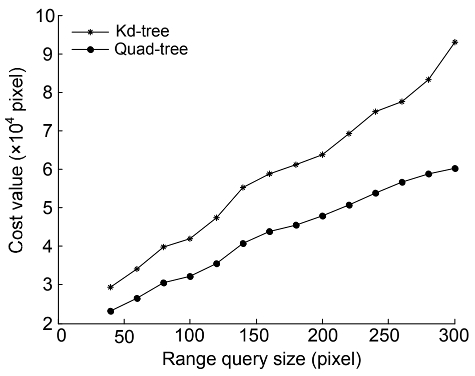


Fig. 6 Comparison of cost values for case 2

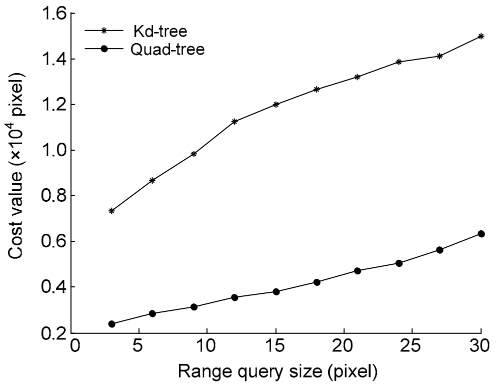


Fig. 7 Comparison of cost values for case 3

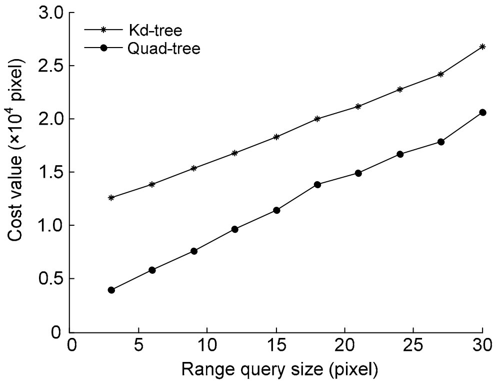


Fig. 8 Comparison of cost values for case 4

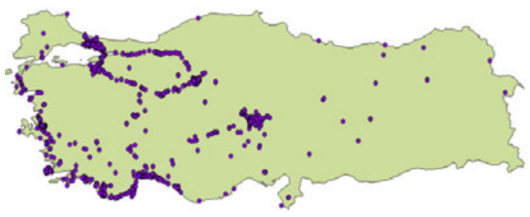


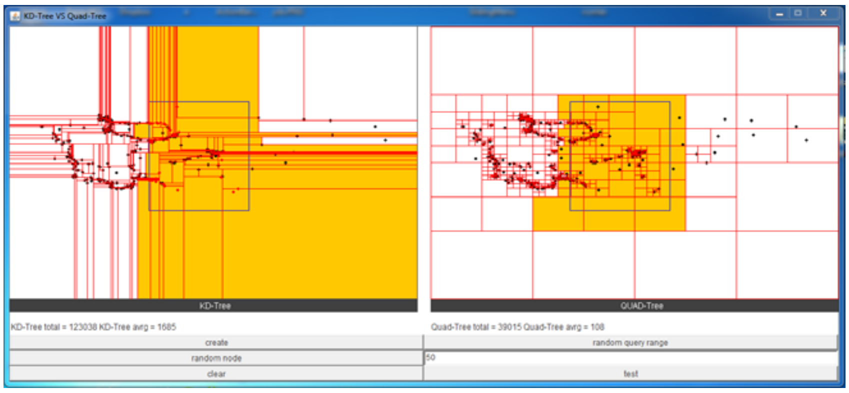
Fig. 9 Distribution of Turkey's points of interest

corresponding performance test results are given in the same order in Figs. 12 and 13.

Fig. 12 shows the change of the cost values according to the query size in the case of applying relatively large sized queries to Turkey's points of interest shapefile data sets. Fig. 13 shows the change of the cost values according to the query size in the case of applying relatively small sized queries to the data.

5 Conclusions

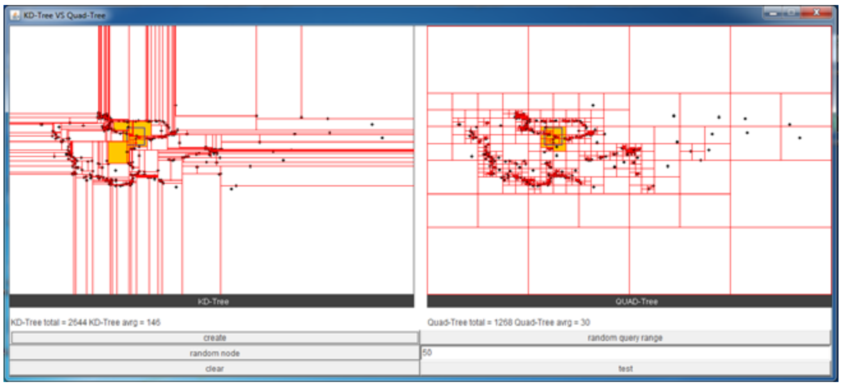
Spatial data are mostly distributed, and their distribution is uncertain and unexpected based on their locations in a given space. This causes data and execution skews and is not a trivial problem in terms of the declustering and parallelization of this type of data.


KD-Tree total = 123038 KD-Tree avrg = 1685

Quad-Tree total = 39015 Quad-Tree avrg = 108

50

Fig. 10 Large range queries over Turkey's points of interest


KD-Tree total = 2644 KD-Tree avrg = 146

Quad-Tree total = 1268 Quad-Tree avrg = 30

50

Fig. 11 Small range queries over Turkey's points of interest

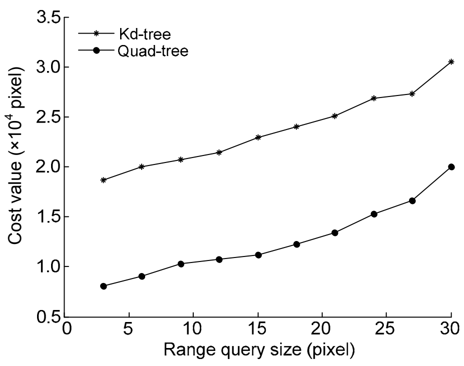


Fig. 12 Comparison of cost values for large range queries over data space

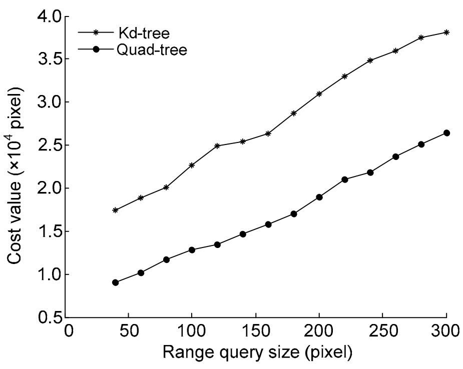


Fig. 13 Comparison of cost values for small range queries over data space

In this paper, we analyzed the feasibility and efficiency of using kd-tree and quad-tree space partitioning techniques for eliminating data and execution skews in distributed parallel querying and executions of spatial data. Spatial indexing techniques in uncertain spaces focus on query efficiency. However, in this study we evaluated their usage in declustering applications for efficient load balancing in 2D range queries applied to non-uniformly distributed point data. The proposed analysis helps with the selection of the best combination of partitions created by index tables that will minimize the response time of a given range query. The key challenge is how to balance the spatial processing load across a large number of worker nodes, given significant performance heterogeneity in the nodes and processing skew in the workload. Based on our initial experience in rendering distributed spatial data (Sayar *et al.*, 2014), and tests results in this paper, it appears that quad-tree indexing gives better parallelization. This is due mostly to the fact that quad trees reveal the spatial locations of data more clearly.

In the future, we will study the situation in which the I/O parallelization of query/rendering is increased on other more complex spatial data types such as line-strings and polygons. R-tree, R^+-tree, and R^*-tree will be evaluated for their efficiencies in declustering applications. As the area of overlapping regions in the R-tree index increases, the performance degrades because of the repeated objects returned by inefficient indexing. It would be good to know which index gives the least overlap and, therefore, the lowest repetition for efficient declustering.

References

Bentley, J.L., 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM*, **18**(9): 509-517. [doi:10.1145/361002.361007]

Beynon, M., Chang, C., Catalyurek, U., *et al.*, 2002. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parall. Comput.*, **28**(5):827-859. [doi:10.1016/S0167-8191(02)00097-2]

Chakka, V.P., Everspaugh, A.C., Patel, J.M., 2003. Indexing large trajectory data sets with SETI. Proc. 1st Biennial Conf. on Innovative Data Systems Research.

Chilès, J.P., Delfiner, P., 2009. Geostatistics: Modeling Spatial Uncertainty. John Wiley & Sons, New York, USA.

Chou, T.C.K., Abraham, J.A., 1982. Load balancing in distributed systems. *IEEE Trans. Softw. Eng.*, **SE-8**(4):401-412. [doi:10.1109/TSE.1982.235574]

Cudre-Mauroux, P., Wu, E., Madden, S., 2010. TrajStore: an adaptive storage system for very large trajectory data sets. Proc. IEEE 26th Int. Conf. on Data Engineering, p.109-120. [doi:10.1109/ICDE.2010.5447829]

DeWitt, D., Gray, J., 1992. Parallel database systems: the future of high performance database systems. *Commun. ACM*, **35**(6):85-98. [doi:10.1145/129888.129894]

Furht, B., Escalante, A., 2011. Handbook of Data Intensive Computing. Springer, New York, USA.

Li, R., Bhanu, B., Ravishankar, C., *et al.*, 2007. Uncertain spatial data handling: modeling, indexing and query. *Comput. Geosci.*, **33**(1):42-61. [doi:10.1016/j.cageo.2006.05.011]

Moon, B., Saltz, J.H., 1998. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Trans. Knowl. Data Eng.*, **10**(2):310-327. [doi:10.1109/69.683759]

Ray, S., Simion, B., Brown, A.D., *et al.*, 2013. A parallel spatial data analysis infrastructure for the cloud. Proc. 21st ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems, p.284-293. [doi:10.1145/2525314.2525347]

Reich, B.J., Chang, H.H., Strickland, M.J., 2014. Spatial health effects analysis with uncertain residential locations. *Stat. Methods Med. Res.*, **23**(2):156-168. [doi:10.1177/ 0962280212447151]

Samet, H., 2006. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco, USA.

Sayar, A., 2013. Fine-grained federation of geographic information services through metadata aggregation. *Sci. Res. Essays*, **8**(46):2242-2256.

Sayar, A., Marlon, P., Geoffrey, F.C., 2014. An adaptive range-query optimization technique with distributed replicas. *J. Cent. South Univ.*, **21**(1):190-198. [doi:10.1007/s11771-014-1930-7]

Sinha, R., Samaddar, S., Bhattacharyya, D., *et al.*, 2010. A tutorial on spatial data handling. *Int. J. Database Theory Appl.*, **3**(1):1-12.

Wang, L., Wu, P., Chen, H., 2013. Finding probabilistic prevalent colocations in spatially uncertain data sets. *IEEE Trans. Knowl. Data Eng.*, **25**(4):790-804. [doi:10.1109/TKDE.2011.256]

Wei, W., 2010. Analysis of spatial database index technology. Proc. 2nd Int. Conf. on Computer Engineering and Technology, p.29-32. [doi:10.1109/ICCET.2010.5486363]

Zhang, Y., Lin, X., Zhang, W., *et al.*, 2010. Effectively indexing the uncertain space. *IEEE Trans. Knowl. Data Eng.*, **22**(9):1247-1261. [doi:10.1109/TKDE.2010.77]

Zhong, Y., Han, J., Zhang, T., *et al.*, 2012. Towards parallel spatial query processing for big spatial data. Proc. IEEE 26th Int. Parallel and Distributed Processing Symp. Workshops & PhD Forum, p.2085-2094. [doi:10.1109/IPDPSW.2012.245]