Efficiently Managing Large-Scale Raster Species Distribution Data in PostgreSQL

Jianting Zhang
Department of Computer Science
City College of New York
New York, NY 10031
jzhang@cs.ccny.cuny.edu

Michael Gertz
Institute of Computer Science
University of Heidelberg
Heidelberg, Germany
gertz@informatik.uni-heidelberg.de

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT

Species distribution data play an important role in biodiversity related research, especially in exploring relationships with the environment. In the recent years, both the number of species being explored and the spatial resolution of species distribution data are increasing fast. It is thus imperative to develop database systems that allow users to efficiently query such large-scale data based on spatial and non-spatial (e.g., taxonomic and phylogenetics) criteria.

In this paper, we present our approach to building such a system by integrating several components, including a quadtree representation of binary raster data, tree path indexing and query processing in PostgreSQL, and window decomposition techniques for spatial queries. Our unique contribution is in associating species identifiers with intermediate quadtree nodes and query optimization for multiple independent queries after window query decomposition. Our system enables PostgreSQL to support binary raster data without requiring any changes to the database backend and is suitable for managing large-scale species distribution data.

Our experiments using 4000+ bird species distribution data related to the Western hemisphere show that the proposed approach in associating species identifiers with quadtree nodes reduces the number of database tuples by more than 1/3 and the average identifiers to be associated with each tuple from 110.6 to 4.8, a significant improvement compared to classic quadtree-based approaches. With respect to query optimization, optimized queries are 6-9.5 times faster than the baseline queries for average query response times and 5.5-8.3 times faster than the baseline queries for maximum query response times for four query window sizes ranging from 0.1 to 5.0 degrees. Our query optimization techniques thus make the system suitable for many interactive applications for querying and exploring species distribution data.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial Databases and GIS; H.2.4 [**Systems**]: Query Processing

General Terms

Design, Experimentation

Keywords

Species distribution data, quadtrees, window query decomposition, query optimization, PostgreSQL

1. INTRODUCTION

Recent years have witnessed significant developments in Internet-based information systems in support of biodiversity studies. In parallel, species distribution data available to the community is increasing at a staggering rate. Community standardization efforts, such as the Darwin Core standard developed by the Biodiversity Information Standards [7] and the adoption of distributed access protocols, such as Distributed Generic Information Retrieval [6], allow users to exchange, share and integrate disparate museum records and research data repositories.

Examples of taxon-based Web-Portals are the Global Biodiversity Information Facility [13], the Ocean Biographic Information System (OBIS) [24], the Mammal Networked Information System (MaNIS) [20], the ORNithological Information System (ORNIS) [25], and HerpNET [15], to name only a few. While point locations of species occurrences themselves are useful to model species distributions at the individual species level, range maps or distribution datasets compiled from museum records or scientific literature play an important role in global and regional biodiversity studies, especially when correlating biodiversity with environmental variables (see, e.g., [14, 36, 37]). For example, the USGS Digital Representation of Tree Species Range Maps from the "Atlas of United States Trees" by Elbert L. Little, Jr. [35] consists of 679 tree species and has played important roles in USGS's vegetation-climate modeling.

More recently, NatureServe has published Digital Distribution Maps of the Birds of the Western Hemisphere, which covers 4,273 bird species. Similar range maps for 1737 mammal species of the Western hemisphere and more than 6000 of the World's amphibians are also available from NatureServe's Website [23]. We envision that the availability of species range maps will significantly increase over the next couple of years, something that will be important not only to environmental modeling but also to phylogeography, phylogenetics and other branches of bioinformatics. It is thus imperative to develop suitable database management infrastructures and techniques to effectively support global biodiversity research and subsequent species conservation practices.

An important functionality in querying polygonal species distribution data is to efficiently answer window queries. One viable approach for this is to use traditional indexing approaches based on the R-Tree. However, for polygonal data, this approach has several shortcomings. First, polygons representing distributions of different species are highly overlapping, a fact that makes the selectivity of space partitioning trees less effective in query processing. Second, such polygons can be very complex and thus computing intersections between the polygons and query windows is computationally expensive. Consequently, this makes it difficult to use spatial databases for online or interactive applications. Finally, species distributions are fuzzy in nature and querying distribution data often can tolerate errors to a certain degree. As such, the majority of existing biodiversity research using species distribution data employs raster tessellations of distribution data.

Compared to vector data, raster data is less well supported in spatial databases. Although the region quadtree has been studied extensively for representing and indexing binary raster data (see, e.g., [12, 16, 21, 30]), surprisingly only a very few database systems support geospatial binary raster data on a native basis. While research prototypes such as QUILT [33] and SAND [9, 32] have been developed based on quadtrees in the context of GIS applications, they do not offer Structured Query Language (SQL) support. In many application domains, however, SQL-based query formulation is preferred, especially when integration with tabular and other non-spatial data is necessary and a client-server architecture is adopted in a distributed computing environment. Motivated by the availability of a module called LTREE for tree path indexing in the PostgreSQL system [26] and the similarity between the defined LTREE data type and linear quadtree paths, we propose to utilize the data storage, indexing and query processing functionality implemented in the LTREE PostgreSQL module for geospatial binary raster data representing species distributions. To support spatial window queries that are not natively supported by the LTREE module, we implemented a window decomposition algorithm to transform a query window into multiple query cells, which then can be used to query LTREE tree path data. Respective query results are then combined to answer the original spatial window query.

The contributions of our approach are the following.

- First, we tackle the problem of managing (including storing, indexing and querying) binary raster data in an object-relational database environment (PostgreSQL in particular), which has many practical applications, yet is poorly addressed in current spatial database practices. The approach we propose does not require any modifications to the database backend.
- Second, large-scale species distribution data are highly overlapping and complex, which makes traditional vector based representation and indexing inadequate for spatial window queries that require fast response times. As an extension to classic quadtree data structures designed for individual datasets, we associate species identifiers with both leaf and non-leaf quadtree nodes. As we will demonstrate, this makes cross-dataset querying much more efficient. Techniques for efficiently storing species distribution datasets and processing spatial window queries are tailored to such a data representation framework.
- Third, we present a prototype system that realizes the above functionality to efficiently manage and query 4000+ species distribution datasets. Our experimental evaluation demonstrates the effectiveness of the proposed approach.

The remainder of the paper is structured as follows. Section 2 introduces some background on large-scale species distribution data management, which is our key application domain, and it also provides an overview of related work. Section 3 presents our approach for storing binary raster species distribution data in a PostgreSQL database. A baseline and an improved approach to query window decomposition and query result combination are discussed as well in that section. Section 4 presents our experiments on 4000+ real species distribution datasets. Finally, Section 5 concludes the paper and outlines future research directions.

2. BACKGROUND AND RELATED WORK

2.1 Species Distribution Data

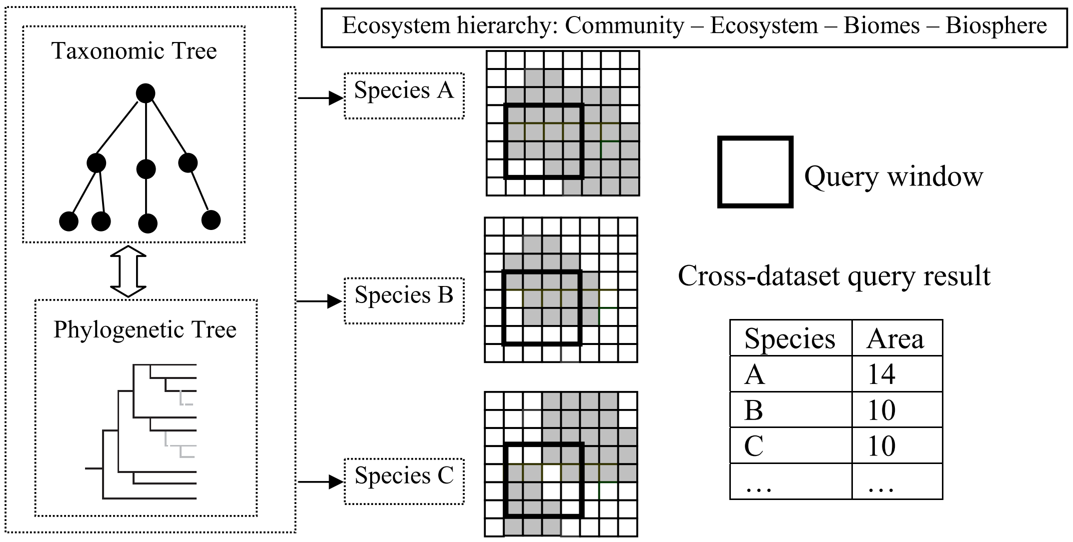
Our research is motivated by the various needs in biodiversity studies to query the numbers of species and their corresponding areas. Each species may have multiple polygons representing its geographic distribution. These polygons may be neighboring or disjoint and may vary significantly in terms of shape and area being covered. Some generalist species may have large distribution ranges while some specialist species may have very restrictive distribution ranges. We assume that species distribution datasets are available in raster format and that a quadtree for each individual species distribution dataset has been built.

Species distribution data are not only geo-referenced, but they are also *species-referenced*. Species may be grouped according to different taxonomic, phylogenetic or functional characteristics. A database system that has the capability to perform spatial- and species-based queries on large-scale species distribution data is essential for interactive exploration and rigorous statistical analysis of relationships between biodiversity and the environment. The most typical scenario for querying species distribution data is illustrated in Fig. 1. In the illustration, each species is associated with a quadtree representing its distribution grid cells using a common raster tessellation. Each species may also be associated with one or multiple taxonomic, phylogenetic or ecosystem hierarchies. The hierarchies can be represented as tree paths or sets of relational attributes, depending on the underlying database implementation. As our focus in this paper is on spatial window queries, we assume a list of species is available after non-spatial restrictions have been satisfied before processing spatial queries. For a spatial window query, we need to return the distribution grid cells of species in the candidate species list that intersect with the given query window. The species identifiers and their areas (and sizes) within the query window will be returned to the user.

The Social-Economic Data and Application Center (SEDAC) at Columbia University has rasterized the original NatureServe's animal species distribution data in ESRI shapefile format into GeoTIFF images at a resolution of 30 arc-seconds. The GeoTIFF images for individual species are available for download through a Web-interface [5]. They have also derived a family richness grid dataset by counting the number of species for all the species families. However, the system does not allow users to define regions of interest or to group species of interest, which motivates our work in managing large-scale species distribution datasets in a spatial database environment that allows users to query such data based on spatial and species criteria.

2.2 Quadtree Indexing and Query Processing

Numerous spatial query indexing approaches have been proposed in the last three decades. The quadtree family is among the oldest and most extensively studied one; for details, we refer the reader to the survey by Gaede and Günther [11] and in particular the book by Samet [31]. On the commercial side, Oracle Spatial and Microsoft SQL Server Spatial support quadtree based indexing of polygonal data. However, their techniques are designed as filtering mechanisms to facilitate querying spatial relationships (disjoint, touch, contain etc.) at the polygon level ([10, 17]) and are not directly accessible to application developers. To the best of our knowledge, quadtree indexing for binary raster data is not available in


Ecosystem hierarchy: Community – Ecosystem – Biomes – Biosphere
Taxonomic Tree
Phylogenetic Tree
Species A
Species B
Species C
Query window
Cross-dataset query result

Species	Area
A	14
B	10
C	10
…	…

Figure 1: Illustration of a spatial window query over three species distribution datasets.

open source databases (and PostgreSQL in particular). This is one motivation for our work to seek for alternative solutions. Existing systems related to raster data management in a database environment, such as Oracle Spatial GeoRaster [18] or Rasdaman [28], are designed to support storing and querying dense multi-dimensional real-valued arrays based on tiling (chunking) techniques and they are not suitable for our application. When large-scale species distribution data are forced to be represented as multi-dimensional array data with binary values, the array would be extremely large and sparse, something our approach tries to avoid. For example, the size of the array for 10,000 species at the global 1 minute resolution would be 10,000 * 10,800 * 21,600, which is approximately 2.5 petabytes, assuming each array element takes 1 byte.

A few research prototypes, such as QUILT [33] and SAND [9, 32], have been developed to manage geographical binary raster data. However, they are more suitable to be categorized as GIS rather than spatial databases. Many database features that are important to application developers are not provided in these prototype systems. Aref and colleagues have published a few papers on implementing and experimenting with space partitioning tree indexing approaches based on the Generalized Search Tree (GIST) framework in PostgreSQL, known as SP-GIST [2, 8]. A variety of quadtree indexing methods, including MX-quadtree, PR-Quadtree, MX-CIF-quadtree and PMR-quadtree, have been developed for PostgreSQL. However, these quadtree indexing methods are designed for line segments rather than polygons. While they are suitable for filtering polygons stored as collections of line segments in processing spatial window queries, they do not support region-based queries directly, which is important in our application to querying species distribution data.

The work by Aboulnaga and Aref [1] describes an algorithm and a cost model for processing window queries in linear quadtrees. The algorithm recursively decomposes the data space into quadtree blocks and uses the blocks overlapping the query window to search the B-Tree used to index the quadtree blocks. While the technique can be implemented in an extensible spatial database environment in a way similar to the authors' later work on SP-GIST, the window query processing technique requires modifying the database backend, and a significant effort may be needed for coding, debugging and testing, which often prevents application developers from adopting such techniques.

The recently released Microsoft SQL Server Spatial is based on an adaptive quadtree-like multi-level grid [10]. It decomposes both polygons and query windows into grid cells according to a common raster tessellation and matches their paths for query processing. Similar to the work described in [1], the B-Tree indexing infrastructure is used to speed up query processing by limiting search ranges. Different from the work in [1], the key values used in Microsoft SQL Server Spatial do not need to be sorted based on Morton code [22]; instead, the parent-child relationships are explicitly maintained based on cell ids (in the form of tree paths). A path matching is further decomposed into a sequence of B-Tree queries. Similar to SP-GIST, Microsoft SQL Server Spatial is based on an extensible indexing framework. However, as discussed above, the shear amount of work required for database backend development makes it less attractive for application developers. A question to ask is "Can we utilize existing database backend functionality to support new types of data without changing the backend?".

In addressing this question, it is interesting to observe that PostgreSQL includes a module called LTREE [26]. The LTREE module essentially implements a generic trie-based indexing method based on the GIST framework. The module has been well developed and is included in recent PostgreSQL releases. Naturally, quadtree paths can be stored as LTREE data objects in a PostgreSQL database that supports path-based queries, such as exact matches and querying ancestors and descendants of a tree path. Unfortunately, LTREE does not support spatial range queries directly, an aspect our framework addresses, as described in the following.

2.3 Overview of the Proposed Approach

Motivated by the work reported in [1] and Microsoft SQL Server Spatial [10] on decomposition based window queries over spatial data represented as quadtree, in the context of managing large-scale species distribution data, we propose the following approach to support spatial window queries in PostgreSQL without requiring any changes to the database backend. Compared to previous work that extends a spatial database backend directly to support new data types and queries, the proposed approach can be categorized as a middleware approach that involves three components: database preparation, query transformation, and result combination.

For database preparation, we build a quadtree for each of the individual species distribution datasets. Second, we union these quadtrees and build a combined quadtree, which associates its nodes (leaf and non-leaf) with sets of species identifiers. Third, we store both the paths of the tree nodes and the corresponding identifiers in a PostgreSQL database after the combination. Finally, we create an index for the tree paths using the LTREE indexing module. For query transformation, we implemented a window decomposition approach where a query window is decomposed into cells using the same raster tessellation as used for the input datasets. The cell identifiers are then used to query the PostgreSQL database, and the individual query results are combined to obtain the final result, which is then delivered to applications.

While the details of the approach are deferred to the next section, we would like to point out that the proposed approach is built on top of a few existing techniques. The most relevant ones are quadtree representation of geo-referenced data and linear quadtrees [9, 12, 16, 30, 32, 33], efficient decomposition of window queries [1, 3, 4, 27, 34] and tree path indexing [26]. Although we do not claim making novel contributions to individual components, the proposed middleware approach seamlessly integrates these components in a novel way. The end-to-end system we developed is able to manage large-scale species distribution data effectively and to process queries efficiently, as we will show in our experiments in Sect. 4.

3. THE PROPOSED SOLUTION

As outlined in the previous section, the basis for our approach consists of three concepts: database preparation, query transformation and result combination. The system components realizing these concepts are illustrated in Fig. 2 and will be detailed in the following subsections. For the rest of the paper, we adopt the common quadtree terminology when referring to classic quadtrees. In classic quadtrees, leaf nodes can be either black or white, representing the presence or absence of data in the respective block (or quadrant). Non-leaf nodes are gray, representing a mixture of white and black descendant nodes. A quadtree node can be represented by the path from the root to that node by concatenating the quadrant numbers (0-3) along the path. Figure 3 shows how quadrants in a quadtree are numbered.

3.1 Database Preparation

When a large number of species distribution datasets – each represents an individual species distribution with a unique identifier – is imported into a database, it is typical that the polygons representing the distributions vary in size and shape. More importantly, the distributions among species may overlap frequently. While one could follow an approach using Oracle Spatial or Microsoft SQL Server Spatial to create quadtrees for the polygons, many of the quadtree nodes will be associated with multiple species, and the quadtree paths will be duplicated multiple times, potentially up to the number of datasets. From a data storage perspective, it is thus beneficial to associate a quadtree node with a set of species identifiers instead of just a single identifier as done in the classic quadtree approach. This is especially true when the number of species is large, as in our application.

A straightforward approach to create such combined quadtree leaf nodes with their associated identifier sets is to visit the gray nodes of multiple quadtrees in a synchronized manner, break down black nodes and combine the relevant identifiers until the largest depths of all quadtree nodes in respective quadrants has been reached. Consider the example shown in Fig. 3 where three species distribution datasets A, B, and C are involved. The quadtrees corresponding to the three datasets are shown in the top of the figure. The combined quadtree and its corresponding leaf nodes are shown in the bottom part of the figure. To illustrate the classic combination process, node 3 in the combined quadtree is derived from node 3 of species A quadtree and so is node 3.0 of species B quadtree, and nodes 3.0.0 and 3.0.2 of the species C quadtree. When the quadtrees of species A and B are combined, node 3 of species A is broken down into four quadrants and each quadrant is associated with species A. Thus, the first quadrant corresponding to node 3.0 is associated with species A and B. Similarly, when the species C quadtree is combined, the quadrant needs to be broken down and the quadrants corresponding to node 3.0.0 and node 3.0.2 will be associated with species A, B and C. To comply with the notation of classic quadtrees, we set a leaf node in the combined quadtree to black if there is at least one species associated with the node.

While the standard quadtree combination (union) process based on the classic quadtree definition (herein referred to as the classic combination) could in principle be implemented, we observed that this is not necessary in the context of processing spatial window queries on quadtree paths. Instead of breaking down the upper level black nodes, combining lower level nodes (black or white) and associating sets of identifiers only with leaf nodes, our new combination approach allows a non-leaf quadtree node to be associated with a set of species identifiers. The proposed combination approach is a straightforward union of identifiers at each combined quadtree node.

From an implementation perspective, such type of combination simply requires a map data structure that is readily available in many packages, including C++ STL. An entry in the map is a pair of $\langle$`path`, `identifiers`$\rangle$, where `path` can be implemented as a string and `identifiers` can be implemented as a set of identifiers. For each given quadtree node represented by its path from the root to the node, we simply look up the path in the map data structure. If there is a match, we add the identifier associated with the quadtree node to the set. If not, we create a new entry with the path as key and a set with the identifier as the value and add the entry to the map. It is trivial to enumerate the key-value pairs in the map data structure and to store the data in a PostgreSQL database. The approach works for both linear quadtrees with secondary storage and pointer quadtrees in main memory. Figure 4 shows the results of the proposed quadtree combination approach. The combined quadtree has 25 leaf nodes and 8 non-leaf nodes that need to be stored in the database, which is significantly less than storing 42 leaf nodes derived from the classic combination. As each quadtree node corresponds to a database tuple, fewer database tuples usually lead to improved indexing and query response time.

Using an example, we next show that query results based on the proposed combination are the same as those based on the classic combination. Assuming path 3 is one of the paths representing a query window cell after query window decomposition (see next subsection for details) and thus we need to retrieve all the species identifiers associated with cells of quadtree nodes that are descendants of path 3 (c.f. Fig.1). Based on Fig. 3 it is obvious that we need to retrieve all the black nodes and count the number of data cells at the finest level for species A, B, and C, respectively. Assume `bk_id` and `sp_ids` are the columns for the quadtree paths and species identifiers, respectively, in a table called `TB`. An SQL query selecting relevant tuples (quadtree node paths) in Post-

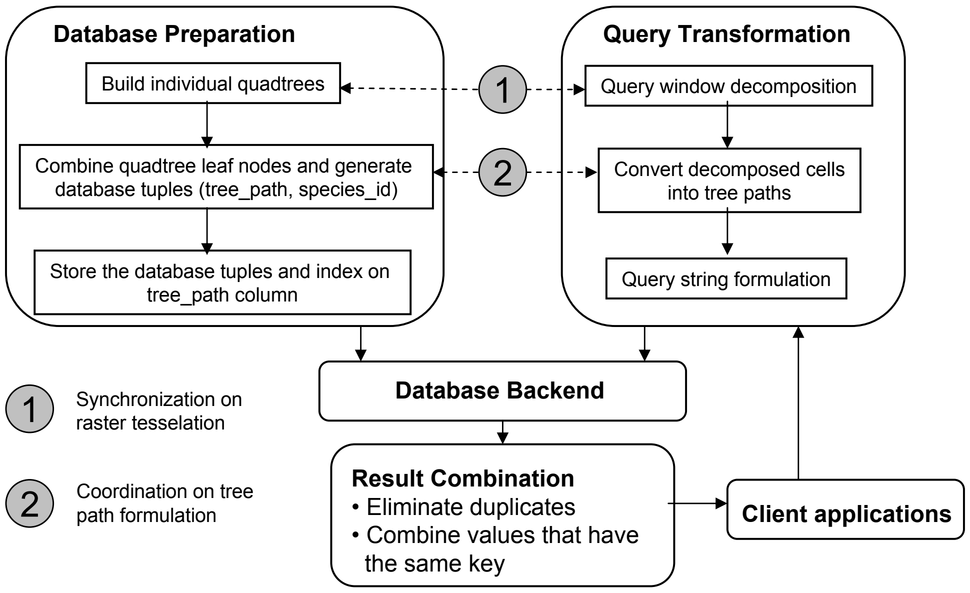

Database Preparation
Build individual quadtrees
Combine quadtree leaf nodes and generate database tuples (tree_path, species_id)
Store the database tuples and index on tree_path column
Query Transformation
Query window decomposition
Convert decomposed cells into tree paths
Query string formulation
Database Backend
Result Combination
• Eliminate duplicates
• Combine values that have the same key
Client applications
1 Synchronization on raster tesselation
2 Coordination on tree path formulation

Figure 2: Components of the proposed system.

greSQL before summarizing the results looks as follows:

```
select bk_id, sp_ids from TB where bk_id <@ '3'
```

`<@` is an operator defined by the LTREE module to predict whether the LTREE object on the left is a descendant of the LTREE object on the right. By definition, the operator is inclusive, i.e., an LTREE object is a descendant of itself. The selection criterion `bk_id <@ '3'` means selecting all the tuples whose `bk_id` are descendants of tree path `'3'`. Since only leaf nodes of the quadtree are stored in the database based on the classic combination (see lower part of Fig. 3), 10 database tuples representing leaf nodes will be returned as answer to the query. It is not difficult to determine the result based on Fig. 3, which are 16, 4, and 4 data cells at the finest level, respectively.

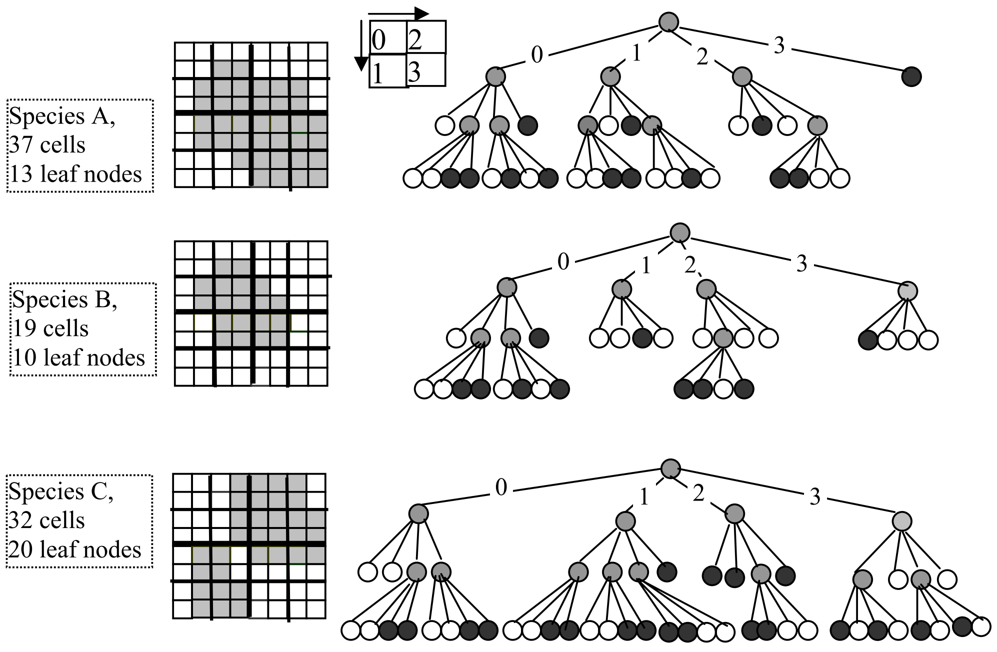
When comparing the tables in Fig. 3 and Fig. 4 (the right-most ones), we can see that the table in Fig. 4 is much smaller. Only six identifiers are associated with the quadtree paths (one for each) and the number is significantly less than the 18 identifiers in the same table in Fig. 3. The example clearly demonstrates that the proposed combination approach requires less storage and computation for producing query results. Associating less species identifiers with quadtree nodes (and hence database tuples) not only reduces storage overhead, but, more importantly, reduces memory consumption per database tuple when swapping pages between disk and main memory. For a limited main memory buffer for a database instance, this means that more database tuples can be brought into main memory, which then speeds up query processing.

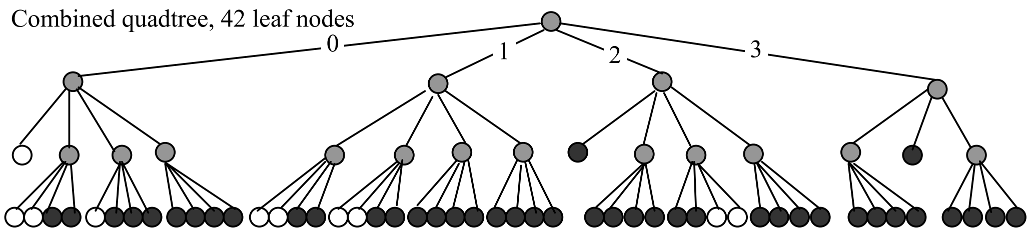
3.2 Query Window Decomposition

Because species distribution data is now stored as tree paths in the database, in the proposed approach, it is necessary to transform a spatial query window into tree paths to match database tuples. While several query window decomposition algorithms are available [3, 4, 27, 34], we have chosen the one reported by Tsai et al. in [34] because of its efficiency and ease of implementation. The output of a decomposed window is given in the form of (x, y, n) where x and y are the coordinates of the top-leftmost data cell at the finest resolution and n is the width/height of a decomposed query window cell (square). As the output is not suitable to be used for a path query against PostgreSQL, we have also converted the algorithm's output into tree paths by repetitively dividing x and y by 2 until the size (width/height) of the divided quadrant is the same as n. The remainders are kept during the division process and are used to construct the path of the query window cell.

While the detailed algorithm is omitted due to space limitations, Figure 5 shows an example of a query window decomposition using the algorithm presented in [34]. It uses the same query window as shown in Fig. 1. At the finest level ($k = 3$), four query window cells, 0.1.3, 1.0.2, 1.0.3 and 1.1.2, are stripped from the left. Subsequently, cells 0.3.1, 0.3.3 and 2.1.1 are stripped from the top, cells 3.0.0, 3.0.1 and 3.1.0 are striped from the right and cells 1.3.0 and 1.3.2 are stripped from the bottom. At the next level ($k = 2$), only one query window cell (1.2) is stripped before the query window becomes empty. The 4*4 query window is decomposed into one level 2 and 12 level 3 cells. Note that the query window cells are different from the data cells at the finest level. A query window cell corresponds to a leaf node in the query quadtree and may cover many data cells at the finest level. They agree only at the finest level of the same spatial configuration that the data quadtree and the query window quadtree share.

The level 2 node with tree path 1.2 has an exact match in the database storing the quadtree paths shown in the combined quadtree in Fig. 4. Among the 12 level 3 cells, some have exact matches in the database and some do not. As we can see from the example, the query window decomposition result does not always agree with the combined quadtree that allows exact matches. As such, for querying the decomposed cells, we not only have to search the database for exact matches of cell paths but also have to search for their ancestors. On the other hand, for the decomposed cells that are not leaf nodes in the corresponding quadtree, we also need to search for all its descendants. Thus, the condition to match a query window cell C with a database tuple r is as follows, assuming that a path is both an ancestor and a descendant of itself.





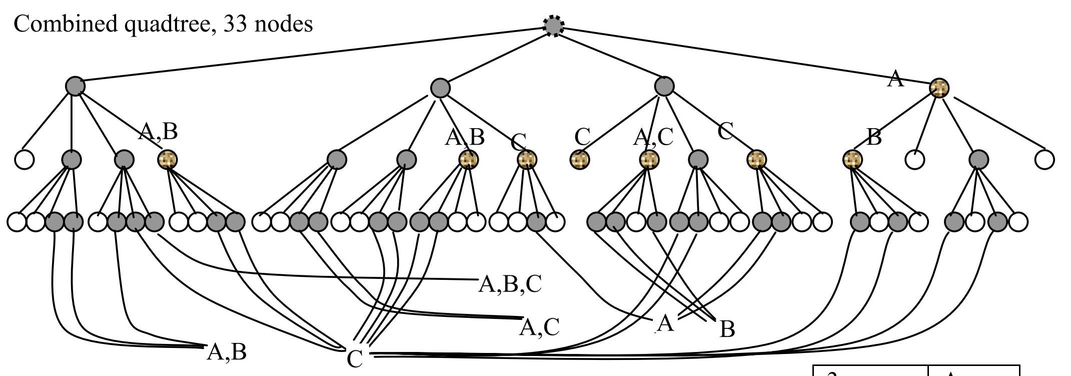
0.1.2	A,B
0.1.3	A,B
0.2.1	A,B
0.2.2	C
0.2.3	A,B,C
0.3.0	A,B
0.3.1	A,B
0.3.2	A,B,C
0.3.3	A,B,C

1.0.2	A,C
1.0.3	A,C
1.1.2	C
1.1.3	C
1.2.0	A,B,C
1.2.1	A,B,C
1.2.2	A,B
1.2.3	A,B
1.3.0	C
1.3.1	C
1.3.2	A,C
1.3.3	C

2.0	C
2.1.0	A,B,C
2.1.1	A,B,C
2.1.2	A,C
2.1.3	A,B,C
2.2.0	C
2.2.1	C
2.3.0	A,C
2.3.1	A,C
2.3.2	C
2.3.3	C

3.0.0	A,B,C
3.0.1	A,B
3.0.2	A,B,C
3.0.3	A,B
3.1	A
3.2.0	A,C
3.2.1	A
3.2.2	A,C
3.2.3	A
3.3	A

Figure 3: Individual quadtrees for species A, B, and C (top), and combined quadtree based on the classic combination approach (bottom), including species identifiers associated with nodes and node paths, respectively.



0.1.2	A,B
0.1.3	A,B
0.2.1	A,B
0.2.2	C
0.2.3	A,B,C
0.3	A,B
0.3.2	C
0.3.3	C

1.0.2	A,C
1.0.3	A,C
1.1.2	C
1.1.3	C
1.2	A,B
1.2.0	C
1.2.1	C
1.3	C
1.3.2	A

2.0	C
2.1	A,C
2.1.0	B
2.1.1	B
2.1.3	B
2.2.0	C
2.2.1	C
2.3	C
2.3.0	A
2.3.1	A

3	A
3.0	B
3.0.0	C
3.0.2	C
3.2.0	C
3.2.2	C

Figure 4: Combined quadtree based on the improved combination approach.

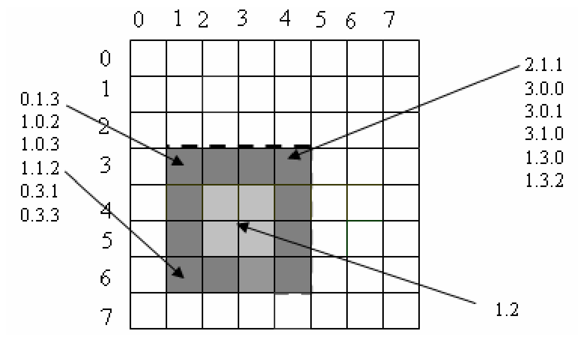


Figure 5: Example of query window decomposition.

(C.ID is an ancestor of r.path) or
(C.ID is a descendant of r.path)

The matching criteria are the same as in the filtering function in Microsoft SQL Server Spatial [10]. Different from Microsoft SQL Server Spatial, which only allows four levels in linear quadtree paths, our approach does not impose a restriction on the depth of tree paths. This is necessary to achieve the desired accuracy for both data and query windows, which is different from the objective in Microsoft SQL Server Spatial whose the primary purpose is object filtering.

For a complexity analysis of the window decomposition process, in addition to a time complexity of $O(m)$ for the decomposition algorithm, where m is the larger of the width and height of a query window given in [34], the complexity for converting the cells into tree paths has a complexity of $O(l * d)$, where l is the number of decomposed cells and d is the depth of the quadtree for a predefined raster tessellation. According to [34], l is proportional to one dimension of the query window, and the complexity can be reduced to $O(m * d)$, because m and l are comparable. Because d is a relatively small number, for example, $d = 14$ is already sufficient for global 1 minute resolution datasets, we conclude that the overall complexity is $O(m)$ for reasonable values of d. For a query window with a few degrees in width (longitude) and height (latitude), m is about 100-1000. As such, the complexity for the window decomposition component is insignificant, and experiments have shown that in general the computation time for window decomposition is negligible.

3.3 Query Optimization and Result Combination

While the condition to match a cell C with a database tuple r discussed in the previous section (which we denote as *baseline query*) guarantees to compute correct query results, it may not be the most efficient technique. For a large query window that does not align with quadrant divisions very well, the decomposed cells can be in the thousands and most of them will be small cells with the same ancestor nodes. In addition, as the queries are sent to the server independently, duplicate tuples may be returned and need to be removed when generating query results. Ideally, the number of duplicates should be minimized as much as possible. In the following, we discuss how to handle these two related issues.

We observe that while the decomposed cells are different, their tree paths can share the same sub-paths. For example, two decomposed query cells 0.3.2 and 0.3.3 share path 0.3. When the query condition "C.ID is a descendant of r.path" is applied when querying the two cells, tree path 0.3 will be returned twice from the database. Figure 6 shows the quadtree for the query window based on its decomposed cells (denoted as *query quadtree*). Assuming all the cells in the query quadtree have matches in the database, the number of times the database tuples corresponding to the query cells will be returned has been indicated by labels in Fig. 6. These numbers are the same as the numbers of leaf nodes under the respective intermediate query tree nodes. For example, in Fig. 6, the database tuples associated with node N1 (in the form of "x.y.z") will be returned 3 times, because it has three leaf nodes as its descendants. The total number of duplicates can be fairly large when the query quadtree has a large depth.

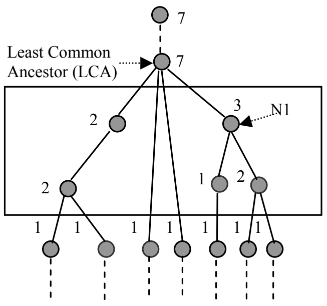


Figure 6: Dividing nodes of a query tree into three parts.

To solve this problem, we divide the cells into three parts. The first part consists of all cells below the rectangle. To answer a spatial window query correctly, we need to retrieve all the tuples whose quadtree paths are descendants (inclusive) of the nodes below the rectangle. The second part covers the nodes inside the rectangle. We need to retrieve all the tuples whose quadtree paths exactly match the cell identifiers. As such, we can combine the queries for individual nodes into one single query by using the query condition

```
where r.path in (path1, path2, ..., pathN).
```

The exact query can further utilize B-Tree indexing to improve query response time. The third part consists of only one node, which is the root of the query tree and it is the Least Common Ancestor of all the decomposed query cells. We need to retrieve all the tuples whose paths are ancestors of the path corresponding to the root of the query tree. It is not difficult to see that the combination of the query results using the three types of cells is the same as using the query criteria used previously without distinguishing the three types of nodes in the query quadtree. The new approach is better in the sense that no duplicate tuples are returned and thus, it is more efficient in query processing and result combination. For distinction, we term the approach as *optimized query*. We will compare it with the baseline query in the experiments in the next section.

3.4 Discussion

The proposed solution differs significantly from existing studies on managing large-scale geo-referenced data in an object-relational database environment. Rather than defining new data types, developing new indexing approaches, modifying query syntax and revising database query engines, our approach utilizes existing database storage and indexing functions. Correspondingly, user queries are transformed into a format that is supported by existing database backends, and the results are combined to answer users' queries effectively and efficiently. As mainstream database systems are becoming more and more sophisticated, even though some systems provide mechanisms to allow user-defined data types and indexing, the shear amount of development work might explain the fact that very few indexing and query processing algorithms have been implemented and are not available in mainstream database systems. We believe our solution has certain appealing features from an application perspective. For example, the underlying database system is left untouched, and one simply uses SQL query syntax instead of being forced to use language APIs in some prototype systems. The features are especially desirable when commercial database systems are used where the source code is not available.

As mentioned previously, our approach relies on efficient tree path indexing. In PostgreSQL, this functionality is provided by the LTREE module. We argue that many commercial database systems are now supporting XML data and thus are likely to support efficient path indexing and querying. As reported in [10], Microsoft SQL Server Spatial actually uses its XML indexing module for polygon indexing based on a variant of the quadtree representation. As such, our approach can be applied to other mainstream databases with minimal modifications.

4. EXPERIMENTS AND EVALUATION

As the research reported in this paper is application-driven, we opt to perform experiments on real data. Among the species distribution data published by NatureServe, we choose the birds datasets as they are more complex than the mammal and reptile datasets. Two groups of experiments will be reported. The first group of experiments is to compare the storage requirements of the classic and improved combination approaches to combine individual quadtrees and store the combined quadtree nodes in the form of (tree_path, species_ids) tuples in database preparation phase. The second group of experiments is to compare the query response times of the baseline and the optimized query approaches at the client side using different query window sizes.

4.1 Experiment Setup

We downloaded the bird species distribution maps from NatureServe's Website [23]. The geographical range of the bird species in the datasets are limited to the Western hemisphere, i.e., (-180,-90, 0, 90). Therefore, the number of cells along both latitude and longitude at 1 minute resolution is 180*60=10,800. We set the depth for the quadtree for the experiments to 14 as 2**14=16,384 is already greater than 10,800. Thus, the effective resolution at the finest resolution level is 180/2**14. We discard species that have only point data and species whose ranges are less than the size of the cell at the finest resolution. The final number of datasets for the bird species tested is 4,062. All experiments aere performed on a Dell Precision T5400 workstation with 8G main memory. As database we use PostgreSQL 8.3.5. The query window decomposition and result combination modules are written in Java. JDBC is used to connect to the PostgreSQL database. We run the database and the query client on the same machine to eliminate network communication cost. All reported query response times are measured in an end-to-end manner.

For the query processing experiments, we use four window sizes, 0.1, 0.5, 1 and 5 degrees for both width and height. For each experiment, we first randomly generate the center and then compute the spatial range of the query window. In the experiments, if any portion of the query window is outside of the study area (global in this case) or the query window does not result in at least one species, the cases will be discarded. For each query window, the experiments are repeated 100 times, and only the experiments with valid query results are recorded for further analysis. The number of valid queries for the four query window sizes are 44, 41, 50 and 45, respectively.

4.2 Experiments on Database Preparation

Combining all the rasterized bird species distributions generates 46,139,247 cells and 1,318,136,140 pairs of (cell, identifier) combinations at the finest resolution, i.e., about 28.7 species per cell.

Using the classic combination, 7,511,823 quadtree leaf nodes are generated while the improved combination approach results in 4,957,050 quadtree nodes, including both leaf nodes and intermediate nodes. Thus, the classic combination generates 51.5% more database tuples than the improved combination. The improved combination approach achieves a 1:9.3 compression ratio (#cells at the finest resolution/#quadtree nodes). The results support our claim that a quadtree representation is an effective way for managing large-scale species distribution data.

The classic combination associates a total of 831,903,250 identifiers with the 7,511,823 nodes, an average of 110.7; the proposed combination associates a total of 23,865,343 identifiers with the 4,957,050 nodes, an average of 4.8. The total number of identifiers to store in the classic combination is 34.9 times larger than that of the proposed combination. The average number of identifiers to associate with each database tuple for the classic combination is 23.0 times larger than that of the our combination approach. The results clearly demonstrate the superiority of our proposed combination approach. The performance improvements are due to the fact that storing species identifiers at the upper levels eliminates the need to store multiple copies of identifiers at the lower levels. The results also show that the linear quadtree storage principle for a single identifier (each identifier represents a layer or dataset), i.e., only storing leaf nodes to save storage, does not apply to cases where multiple identifiers are associated with quadtree nodes.

4.3 Experiments on Query Processing

In this group of experiments, we first examine query window decomposition times followed by query response times using both the baseline query and the optimized query for all queries. We then report the average and maximum response times for the two query approaches under the four query window sizes. Experiment results show that most of the window decomposition cost is less than 10 milliseconds for up to 2,500 query cells in the query, which is the maximum number among all our experiments. The cost is less than 3 milliseconds for query windows whose number of decomposed query cells is less than 500. The cost for query window decompositions is negligible in general.

Figure 7 shows the query response times (in milliseconds, y-axis) versus the number of returned species using both a baseline query (blue diamonds) and the corresponding optimized query (pink squares). As one can see, the optimized queries perform better across different query window sizes. This can be further demonstrated through Table 1. The optimized queries are 6-9.5 times faster than the corresponding baseline queries for average query response times and 5.5-8.3 times faster than the baseline queries for maximum query response times for the four query window sizes, respectively.

The average response times for the optimized query approach vary from 0.03 to 1.16 seconds for query window sizes from 0.1 to 5.0 degrees. The fast response times make optimized queries suitable for many interactive applications. We again want to bring to it to the reader's attention that the excellent performance is achieved without any modifications to the database backend, which makes our solution preferable in many practical applications.

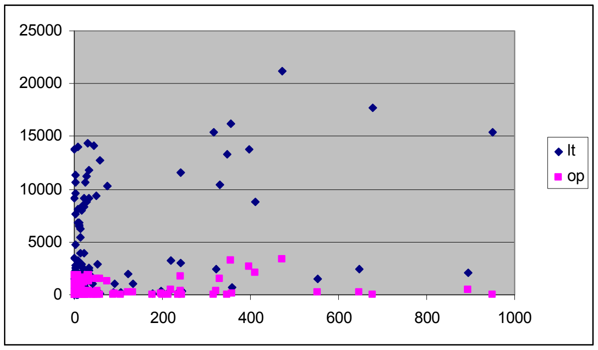


Figure 7: Plot for query result set complexities versus query response times (in milliseconds) for both the baseline (lt) and optimized (op) query approaches.

	Baseline Query		Optimized Query	
Window Size	avg	max	avg	max
0.1	0.14	0.36	0.03	0.06
0.5	0.93	2.39	0.12	0.25
1	1.63	3.93	0.20	0.50
5	9.68	21.13	1.16	3.38

Table 1: Average and maximum response times for four query windows for the three approaches (in seconds).

5. CONCLUSIONS AND ONGOING WORK

In this paper, we addressed the practically relevant problem of storing, indexing and querying large-scale species distribution data in the form of binary rasters in a database environment. We adopt a middleware approach by utilizing existing PostgreSQL database support for tree paths and to transform spatial window queries for tree path matching. Unlike previous studies, our approach does not require any modifications to the database backend, and it is applicable to database systems that support tree path matching. As most of the mainstream database systems do or will support XML data, the approach has the potential to provide spatial window query capabilities in such database systems.

In addition to building an efficient end-to-end system to manage large-scale species distribution datasets, our contributions also include an improved combination approach to union individual quadtrees and to prepare database tuples representing tree paths with species identifiers. We presented an optimized query method to avoid generating duplicate query result tuples and to reduce path matching overhead at the same time. Experiments using 4000+ real bird species distribution datasets have shown that the proposed combination approach reduces the number of database tuples by more than 1/3 and that it reduces the average number of species identifiers to be associated with each tuple from 110.6 to 4.8. Correspondingly, the total number of tree path and identifier combinations is reduced from 831,903,250 to 23,865,343, a 35 times saving. With respect to query optimization, the optimized queries are 6-9.5 times faster than the corresponding baseline queries for average query response times and 5.5-8.3 times faster than the baseline queries for maximum query response times for four query window sizes, respectively. The average response times for optimized queries vary from 0.03 to 1.16 seconds for query window sizes from 0.1 to 5.0 degrees, which makes the optimized query processing technique suitable for many interactive applications.

In our future work, the primary focus is to further extend our approach to manage even larger scale of species distribution datasets, possibly through utilizing support for additional data types, indexing and query optimization algorithms as well as developing new approaches at both the server and client side. Our ultimate goal is to support the management of all known species datasets at the million scale and to reduce average query response times to below one second for realistic query window sizes in order to support interactive user applications, such as visual exploration.

6. REFERENCES

[1] A. Aboulnaga and W.G. Aref: Window query processing in linear quadtrees. Distributed and Parallel Databases 10(2):111–126, 2001.

[2] W.G. Aref and I.F. Ilyas: SP-GiST: An extensible database index for supporting space partitioning trees. Journal of Intelligent Information Systems 17(2-3):215–240, 2001.

[3] W.G. Aref and H. Samet: Decomposing a Window into Maximal Quadtree Blocks. Acta Informatica 30(5):425–439, 1993.

[4] W.G. Aref and H. Samet: Efficient Window Block Retrieval in Quadtree-Based Spatial Databases. Geoinformatica 1(1):59–91, 1997.

[5] SEDAC Species Distribution Grid. `http://sedac.ciesin.columbia.edu/species/`.

[6] Distributed Generic Information Retrieval (DiGIR). `http://digir.sourceforge.net`.

[7] Biodiversity Information Standards (TDWG). `http://www.tdwg.org`.

[8] M.Y. Eltabakh, R. Eltarras, and W.G. Aref: Space-Partitioning Trees in PostgreSQL: Realization and Performance. In Proc. 22nd International Conference on Data Engineering, IEEE Computer Society, 2006.

[9] C. Esperanca, and H. Samet: Experience with SAND-Tcl: A scripting tool for spatial databases. Journal of Visual Languages and Computing 13(2):229–255, 2002.

[10] Y. Fang, M. Friedman, G. Nair, M. Rys, and A.F. Schmid: Spatial indexing in Microsoft SQL server 2008. In Proc. 2008 ACM SIGMOD International Conference on Management of Data, 1207–1216, 2008.

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

[12] I. Gargantini: An Effective Way to Represent Quadtrees. Communications of the ACM 25(2):905–910, 1982.

[13] Global Biodiversity Information Facility. `http://www.gbif.org`.

[14] K. He and J. Zhang: Testing the correlation between beta diversity and differences in productivity among global ecoregions, biomes, and biogeographical realms. Ecological Informatics 4(2):93–98, 2009.

[15] HerpNET: A new tool for the study and conservation of biodiversity. `http://www.herpnet.org`.

[16] G.M. Hunter and K. Steiglitz: Operations on Images Using Quad Trees. IEEE Transactions on Pattern Analysis and Machine Intelligence 1(2):145–153, 1979.

[17] R.K.V. Kothuri, S. Ravada, and D. Abugov: Quadtree and R-tree indexes in Oracle spatial: a comparison using GIS data. In Proc. 2002 ACM SIGMOD international conference on management of data, 546–557, 2002.

[18] R. Kothuri, A. Godfrind, and E. Beinat: Pro Oracle Spatial for Oracle Database 11g. Apress, 2007.

[19] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis: R-Trees: Theory and Applications. Springer, New York, 2005.

[20] Mammal Networked Information System)MaNIS. `http://manisnet.org/`.

[21] D.M. Mark and D.J. Abel: Linear Quadtrees from Vector Representations of Polygons. IEEE Transactions on Pattern Analysis and Machine Intelligence 7(3):344–349, 1985.

[22] G.M. Morton: A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, IBM, Ottawa ,Canada, 1966.

[23] NatureServe: A Network Connecting Science with Conservation. `http://www.natureserve.org/getData/animalData.jsp`.

[24] Ocean Biographic Information System (OBIS). `http://www.iobis.org/`.

[25] ORNithological Information System (ORNIS). `http://olla.berkeley.edu/ornisnet/`.

[26] LTREE Module for PostgreSQL `http://www.postgresql.org/docs/current/static/ltree.html`.

[27] G. Proietti: An optimal algorithm for decomposing a window into maximal quadtree blocks. Acta Informatica 36(4):257–266, 1999.

[28] rasdaman: The Intelligent RasterServer. `http://www.rasdaman.com`.

[29] H. Samet: The Quadtree and Related Hierarchical Data-Structures. ACM Computing Surveys 16(2):187–260, 1984.

[30] H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber: A Geographic Information-System Using Quadtrees. Pattern Recognition 17(6):647–656, 1984.

[31] H. Samet: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc., 2005.

[32] H. Samet and R.E. Webber: Extending the SAND spatial database system for the visualization of three-dimensional scientific data. Geographical Analysis 36, 87–101, 2006.

[33] C.A. Shaffer, H. Samet, and R.C. Nelson: QUILT: a geographic information system based on quadtrees. International Journal of Geographical Information Systems, Volume 4(2):103–31, 1990.

[34] Y.H. Tsai, K.L.N. Chung, and W.Y. Chen: A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. IEEE Transactions on Knowledge and Data Engineering 16(4):519–523, 2004.

[35] Digital Representations of Tree Species Range Maps from "Atlas of United States Trees" by Elbert L. Little, Jr. `http://esp.cr.usgs.gov/data/atlas/little/`.

[36] J. Zhang and L. Gruenwald: Embedding and extending GIS for exploratory analysis of large-scale species distribution data. In Proc. 16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, 28, 2008.

[37] J. Zhang, D.D. Pennington, and X. Liu: GBD-Explorer: Extending open source Java GIS for exploring ecoregion-based biodiversity data. Ecological Informatics 2(2):94–102, 2007.