Geospatial Processing in BaseX

Master Thesis in fulfillment of the requirements
for the degree of Master of Science (M.Sc.)

submitted from

Masoumeh Seydi Gheranghiyeh

in the

Universität
Konstanz

Department of Computer and Information Science
Chair of Database and Information Systems Group

1.Supervisor: Prof. Dr. Marc H. Scholl

2.Supervisor: Prof. Dr. M. Grossniklaus

Konstanz, 2014

Abstract

Geospatial information has been growing rapidly and has been used in variety of applications such as Geographic Information Systems, bioinformatics, and decision support systems. Variety of applications have demanded efficient approaches of data manipulation to cover their requirements. The XML-based data representation and manipulation in this field is one of these approaches that also has been considered and applied by databases, especially native XML databases.

This work tries to investigate the challenges of geospatial data processing in BaseX [26], an open-source native XML database, by discussing the issues in geospatial querying, geospatial functionality, and query efficiency. At first, this thesis starts with introducing a number of indexing approaches as various ways of efficient geospatial data manipulation. Next, general geospatial functionality in some similar databases are investigated. Then, the geospatial features for Geography Markup Language (GML) [23] data representation are added to BaseX, using the Java Topology Suite (JTS) library [66] and based on the EXPath specification.

To improve the performance of spatial queries, an indexing structure is applied as an external library. At the end, we have tried another querying approach by connecting to MongoDB, a powerful document-based database system, through BaseX as an effort to find other efficient ways. Evaluations in various stages and provided advantages and disadvantages of different approaches can be considered in deciding on future developments in BaseX.

Acknowledgments

The completion of this master thesis would not have been possible without the support of many people. I would like to express my sincerest thanks towards Dr. Christian Grün for his continuous support and patience during my research. I should confess that I cannot imagine a better person to guide and advise me. Besides, I would like to extend my gratitude to Prof. Dr. Marc H. Scholl to give me the chance to be involved in this group and to Prof. Dr. Michael Grossniklaus for his helpful guidance and comments.

My special appreciation also goes to Dr. Rolf A. de By from the University of Twente for answering completely and accurately the questions that I would ask frequently and providing me a sample data set to test. I would like to offer my heartful gratitude my parents and family for encouraging me throughout my whole study. At the end, I am deeply appreciative of my beloved, M. Ali Rostami, without whom it would have been impossible to continue.

Contents

1 Introduction

Geospatial data supports a wide range of ever-expanding applications in various scientific, governmental, business, and industrial areas such as geoscience, energy exploration, natural resource management, disaster management, transportation engineering, and urban planning. Numerous researches and efforts have been dedicated to make the most efficient use of this data, either in representation or manipulation. XML-based structures are part of the widely-used standards to represent the complex structure of geospatial data. Geographic Information Systems (GIS) [31], traditional databases, and other systems have been developed or adopted to handle this type of data format. Native XML database is also another approach introduced in recent years to fulfill the geospatial requirements. This requires adding geospatial functionality as a starting point.

Our contribution is to add geospatial functionality to BaseX with the support of GML as a geospatial data format. To do the spatial operations on GML, existing open-source libraries are applied. The big challenge of implementation is the effectiveness of operations since geospatial data is often huge in size and complex in structure. Hence, performance improvement of data retrieval from such a data structure needs special efforts.

Spatial indexing structures are common solutions to improve query performance according to specific constraints and properties of spatial data. As various spatial index algorithms consider different requirements, choosing a proper index should be based on our individual system properties. We have followed this purpose first through a short study over a huge set of indexing structures. Since the number of indexing structures are large, we have selected the structures among the most widely accepted ones in the literature. The goal of this study is to add a new source of discussion for these structures that helps to understand them more effectively for further developments in BaseX. However there are other efforts discussing the index structures, here we provide a comprehensible explanation and different perspectives, which are collected from various sources. To give a quick and efficient overview over these structures, we add a short summary explaining the indexing structures, advantages, and disadvantages. This summary helps to decide the optimum structure for our approach.

Then, an overview of the geospatial features in similar database systems is provided. Since the database systems with geospatial functionality are practical samples, this overview can help us to gain an insight into the different ways of implementation. In this part, we explain how the geospatial functionality is provided in each database. Furthermore, the indexing structures together with different abilities supplied by each of these databases are discussed briefly.

In the next steps, we develop the *Geo Module* together with an indexing structure in BaseX. Evaluation of this module using the real-world data comes hereafter to demonstrate the efficiency of implementation. As an experimental way towards more effective geospatial features, we provide a driver to connect to the well-known document-based database MongoDB. Through this connection, the queries that are written in BaseX, are executed in MongoDB and then the results are accessible via BaseX. This approach can benefit from the spatial index structure and other spatial functionality of MongoDB. We compare the query time and indexing via this connection with the direct query on BaseX. These results and evaluations are the basis for further investigations and developments in BaseX.

This thesis is structured as follows. We first start with the definitions of concepts needed later in the discussions in Section 2. In addition, a number of spatial indexing structures are introduced in this section. Then, Section 3 explains related works in various areas in geospatial processing. The implementation of the Geo Module in BaseX is described in Section 4. Towards more efficient ways, an approach is experienced by connecting to MongoDB which is evaluated in Section 5. At the end, ideas for further works are summarized in Section 6.

2 Geospatial Data Processing

According to *Collins* dictionary, the word *geospatial* is an adjective relating to a position of things on the earth's surface and *geospatial data* is the data representing any thing on the earth's surface. Regarding the importance of geospatial data, many standards and systems have been developed during the years to manipulate, represent and make use of them. Countries and organizations are using different systems and standards to represent geospatial data and consequently there are numerous systems to fulfill a range of requirements. A an example, the World Geodesic System (WGS) is one of these systems that is commonly used all around the world. WGS84 is the latest version of this standard, which was introduced in 1984 and is widely used for expressing the locations on the earth in wide range of services, including satellite services, notably Global Positioning System (GPS), and Google Maps which employ this system as a reference.

In this section, we mainly explain some existing geospatial indexing structures as the most critical part of geospatial data processing in database systems. After some basic definitions in Section 2.1, geospatial data processing in XML [14] databases is shortly discussed in Section 2.2. Finally, a set of geospatial indexing algorithms are introduced in Section 2.3.

2.1 Terms and Definitions

Geospatial data is the location's information describing *features* and locations as well as their characteristics which have a spatial component to connect them to a place on the earth, such as countries, cities, buildings, roads, and rivers. Here, a *feature* represents a physical entity, e.g., a structure, a lake, a tree, or a shape. In fact, it is an abstraction of real-world phenomenons, such as geometries, attributes, and relationships. Hence, real-world entities could be expressed as a set of features. Geospatial data also applies to three-dimensional data, like above and below the earth's surface [55]. Quite often, this information is described in a multi-layer hierarchy. Geospatial objects and entities are formulated through different encodings and formats, which are developed in regards to the specific requirements. An ordered set of numbers, called coordinates, identifies the position of objects defined in a specific *coordinate system*.

A *coordinate system* as a basic concept is a method to uniquely determine the position of an object in a space of a given dimension. Using particular definitions, components, and properties, a coordinate system determines the coordinates which specify the linear and angular position of entities. Coordinate systems are defined by various organization, countries, or governments based on some parameters, which we discuss them in the following.

A country may have more than one coordinate system for its national-wide geographic data. As an instance, all three systems *DHDN*[1], *ETRS89 UTM zone 32N*, and *PD/83 Zone 3* are examples of Germany's coordinate systems. These systems are considered as local or regional ones, while there are global coordinate systems to give features of the world unique coordinates and specifications, like the WGS system. The coordinates of a feature or an object may differ from one system to another. As an example, the coordinates of Berlin in the WGS84 coordinate system with longitude and latitude is $(52.520007, 13.404954)$ while in Germany's *ETRS89 UTM zone 32N* coordinate system, it is $(32798809.64, 5828000.49)$. Accordingly, it

[1] Deutsches Hauptdreiecksnetz

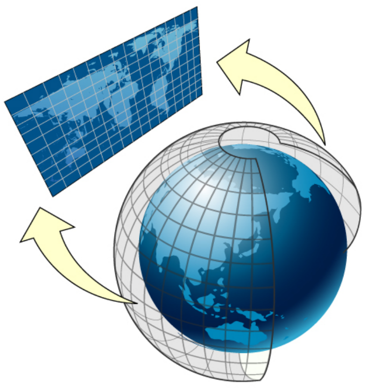


Figure 1: Projection of a geographic coordinate system to a 2D flat map (source [32])

is necessary to know, how the coordinate systems are related in concepts and definitions to make the conversion from one system to another possible. The conversion is required when we need to move data between coordinate systems.

In geographic information systems (GIS), coordinate systems are categorized in two major and common types:

- **Global or spherical coordinate system**

  Also referred as *geographic coordinate system*, represents the data on the earth in a 3D spherical surface by longitudes and latitudes. Points are referenced by their longitude and latitude values which are angles measured often in degrees (or in grads) from the earth's center to a point on the earth's surface. Sometimes it is useful to associate longitude and latitude values with x and y axis respectively. WGS84, NAD[2] 1927, and NAD 1983 are examples of geographic coordinate systems.

- **Projected coordinate system**

  This coordinate system represents the earth's round surface on a 2D flat surface, based on a map projection as shown in Figure 1. In fact, a projected coordinate system is always based on a geographic coordinate system on the earth and the word "projected" refers to the projection of data on spherical the earth to a flat space, called *map projection*. Points in such system are located by x and y values on a grid. In some references, this coordinate system is also called the raw coordinate system.

To define a coordinate system, some parameters should be considered, such as the measurement framework (geographic or planimetric), the unit of measurement (e.g., meter for the projected coordinate systems and decimal degrees for geographic coordinate systems), and the projection definition for the projected coordinate system.

In order to express the geospatial data by coordinate systems, various encodings have been developed by Open Geospatial Consortium (OGC) [16], following different goals. OGC as an international consortium is the main reference to develop interface standards for geospatial content, services, data exchange, data communication, and data processing. Here, we introduce only a short list of common data representation standards which are related to our study.

[2]North American Datum

- Geography Markup Language (GML): The XML-based standard encoding defined by OGC that serves as a modeling language to express geographical features in both spatial and non-spatial characteristic. It is used as an interchange format in Internet data storage, exchange, transport, and transaction. The key to the effectiveness of GML is its capability to integrate all forms of geographic information. The previous versions of GML, namely before version 3.0, encode a set of 2D geometries, i.e., Point, LineString, and Polygon. GML 3.0 introduces Curve and Surface as new geometries, plus the support of geometries in 3D.

- Keyhole Markup Language (KML) [51]: An XML-based grammar to encode and represent the geographic data in an 3D earth viewer such as Google Earth, also on a 2D map like Google Maps. KML has been adopted as an international standard by OGC. The 3D geographic coordinates for each place are represented with longitude, latitude, and if available altitude. Coordinates are provided using the plain unprojected WGS84 longitude and latitude. The altitude could be omitted and assumed as 0. KML mainly focuses on visualizations and expresses what and how to show the data. Code 1 shows a sample data in KML.

Code 1: A simple KML example representing a Point

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
<Placemark>
  <name>New York City</name>
  <description>New York City</description>
  <Point>
    <coordinates>-74.006393,40.714172,0</coordinates>
  </Point>
</Placemark>
</Document>
</kml>
```

- Well-known Text (WKT): A text-based markup language to represent the vector geometries which supports a big range of objects like Geometry, Point, LineString, Curve, Polygon, Surface, curvePolygon, and multi-part geometries such as MultiPoint, MultiCurve, MultiSurface. Below are examples of Point, LineString, and Polygon represented in WKT:
  - Point: POINT (1 2)
  - LineString: LINESTRING (3 1, 1 3, 4 4)
  - Polygon: POLYGON ((3 1, 4 4, 5 4, 1 2, 3 1), (2 3, 3 3, 3 2, 2 3))

- Well-known Binary (WKB): The binary representation of WKT data to be used in a database system.

- GeoJSON [17]: A geospatial data encoding based on the JavaScript Object Notation (JSON). An object is represented by a set of name/value pairs. The whole data is encoded as text and no advanced numeric data types are employed. It is mainly used as a lightweight text format for data exchange. The default coordinate system in this standard is WGS84, however other systems could be defined. A sample point in GeoJSON is provided in Code 2:

Code 2: A simple GeoJSON example representing a Point

```
{
    "type": "Point",
    "coordinates": [30, 10]
}
```

2.2 Geospatial Data in Native XML Databases

As a powerful platform-independent XML grammar introduced by the GIS community (OGC), GML plays an important role in spatial data modeling, integration, sharing, transmission, and exchange because of its flexibility in application schema, self-descriptive format, and rich data expression. Considering the XML-based structure of GML, it could be embedded and queried anywhere in the document, and be mixed with any other type of XML, images, and etc.

In addition to the general functions and features on the structure of GML data, geospatial concepts and requirements could be included in the native XML databases. To start with, the GML elements should be read and treated as geometries. Therefore, the string information is not enough to satisfy the geospatial features. The positions, locations, and relations between an arbitrary set of geometries raise some requirements regarding to the topics and definitions in related scientific fields like Geometry and Geoscience. For instance,"What is the distance between two geometries?" or "Which geometries intersect a particular geometry?" are examples of such requirements. Furthermore, there are features related to single geometries, such as the length and area of a geometry, the number of interior rings of a polygon, and the starting point of a line. These features can be provided as XQuery functions by a database system or XQuery processor and be mixed with standard XQuery, full-text, and other functions. To this date, numerous ideas have been introduced for geospatial functionality as well as indexing. Here, we discuss some geospatial index structures and then we will have a look at some sample implementations in Section 3.3.

2.3 Geospatial Indexing

According to the survey of Chin *et al.* [19], indexing has dramatic influences on data manipulation and storage. Conventional index types are not suitable for supporting geospatial data, since it is very large and complex in structure and relations. Besides, the spatial queries differ from non-spatial queries in several important ways. For example, spatial queries include geometry data types and consider the spatial relationships, e.g. containment, between the geometries. Additionally, the operators used for geospatial data retrieval are complicated and spatial orderings would be hard to define. An important reason which makes the traditional indices not appropriate is that indexing strategies will consider the spatial objects in one dimension and do not preserve the spatial proximity.

In order to fulfill the geospatial specialty and improve the data retrieval, a large number of spatial indexes has been introduced. The strength or weakness of an indexing approach mostly depends on the requirements, query types, and applications

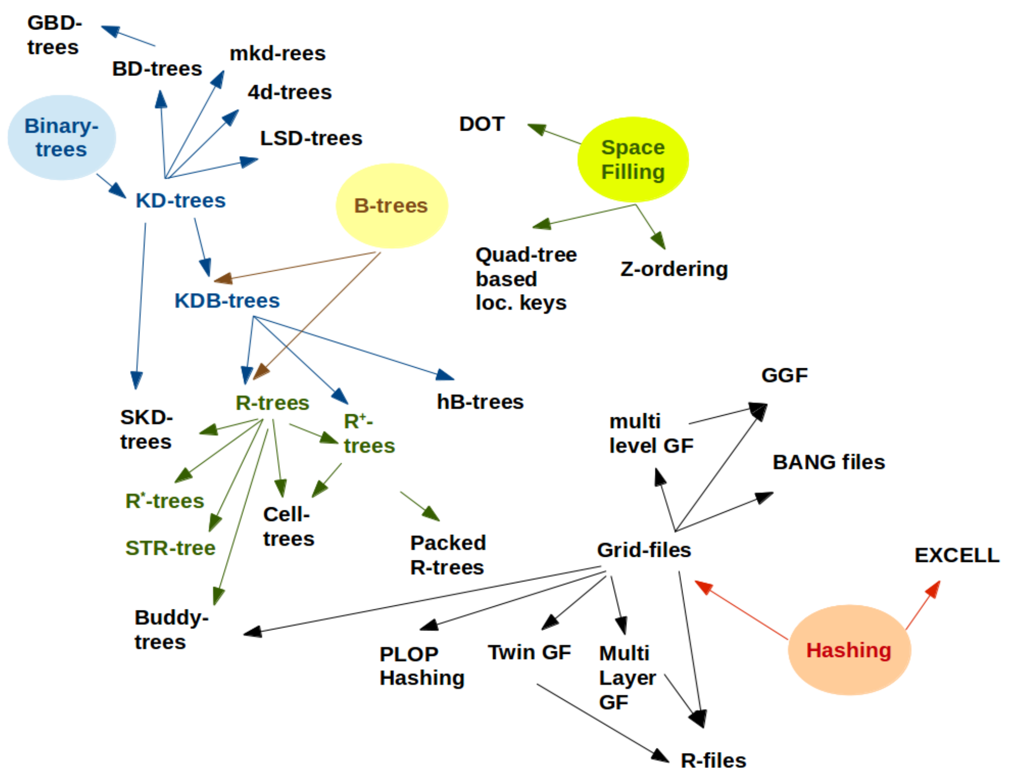


Figure 2: An overview of some important indexing structures developed over the years (created based on the ideas mainly from [19])

for which it is developed. The high percentage of these structures is based on the divide and conquer algorithm [1]. In general, Chin *et al.* [19] have categorized all approaches to three different abstract classes:

- Object Mapping: Given a k-dimensional polygon with n nodes, these approaches map this shape to a kn-dimensional or try to look at the shape of polygon as a single object in the original k-dimensional polygon.
- Object Duplication: These approaches keep different instances of an object in different spaces with the similar representations to take advantage of the properties of all these spaces.
- Object Bounding: Similar to the idea of Quad-tree [25] and Marching cube [45] (an approach in Computer Graphics), these approaches enclose the polygon as accurate as possible by dividing the space in different ways.

On the other hand, the index structures are mostly based on well-known tree structures, like Binary-tree [38], B-tree [6], and Quad-tree. To see how different approaches are developed based on these data structures, here we cover just a selective list of them. Figure 2 visualizes the relation of some indexing structures, which were developed over the years. An arrow from structure A to B means that B is an extension of A. The colored circles also show the main categories of these trees. In the upcoming sections, these structures are briefly explained: Binary-tree based in Section 2.3.1, B-tree based in Section 2.3.2, and Quad-tree in Section 2.3.3 from the space filling index structures.

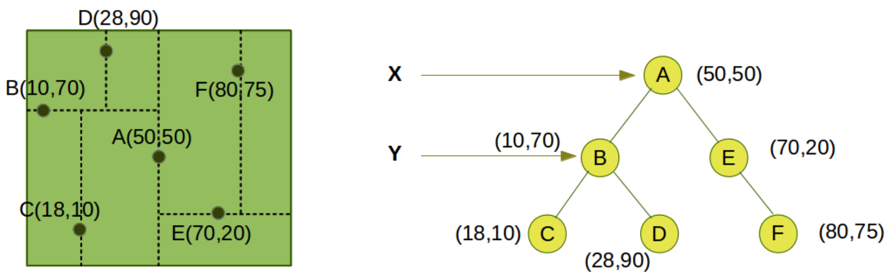


Figure 3: A representation of the KD-tree

2.3.1 Binary-Tree Based

The indexes in this group are conceptually derived from the binary search tree by adopting and generalizing the idea of space partitioning. The KD-tree was the very first indexing structure of this kind introduced by Bentley [11]. It is a generalization of the binary tree structure for organizing k-dimensional points. The basic idea for a 2D example is to alternatively split the area by x- and y-coordinate, such that at each level the points are split half in left, half in right and half below, half above, respectively. Generally, for every non-leaf nodes in the same level, there is a k-th dimension discriminator that defines the dimension along which the areas must be split to form the left and right subtrees. The split has to result in two subspaces, such that the points in the left or below subspace have smaller value in the dimension which split happens, e.g., X, than the parent node. In the same dimension, all nodes in the right or above subspaces have greater values. More clearly, if the discriminator for a node is associated to the i-th dimension, after splitting a space along the i-dimension all i-th attribute of the left subtrees nodes have smaller value than the i-th attribute of this node and all the right subtree nodes have the greater i-th attribute.

Figure 3 shows the representation of some sample data and the corresponding KD-tree. As can be seen, according to the discriminator x in the first level, nodes B, C, D in the left subtree of A have a smaller x value and nodes E, F in the right subtree have a greater x value. In the next level that the discriminator is y, the same consequently happens for y values. For example, node C, which is in the subspace half below B, has a smaller y value.

The KD-tree as an important search structure, has been widely used and studied [61]. With a simple implementation, it works effectively for range queries since they are formed by splitting the space by planes orthogonal to axis. The balanced KD-tree is also effective in searching the Nearest Neighbour in 2D which plays critical roles in database retrievals, classification problems, and clustering problems together with other range queries. However, some variants have been introduced to allow for better performance in clustering, searching, storage efficiency, and balancing, especially in higher dimensions. We discuss some of these variants in the following.

Non-homogeneous KD-tree. A complication arises when an item is deleted from the KD-tree and a node from the subtrees must be replaced. This complication comes from the discriminator in that level. After the deletion, either the node with the smallest value in the right subtree should be replaced or the node with the biggest value in the left subtree.

The Non-homogeneous KD-tree [10] was proposed to make the process of deletion cheaper. The major alteration to build this structure is that the partitioning is not

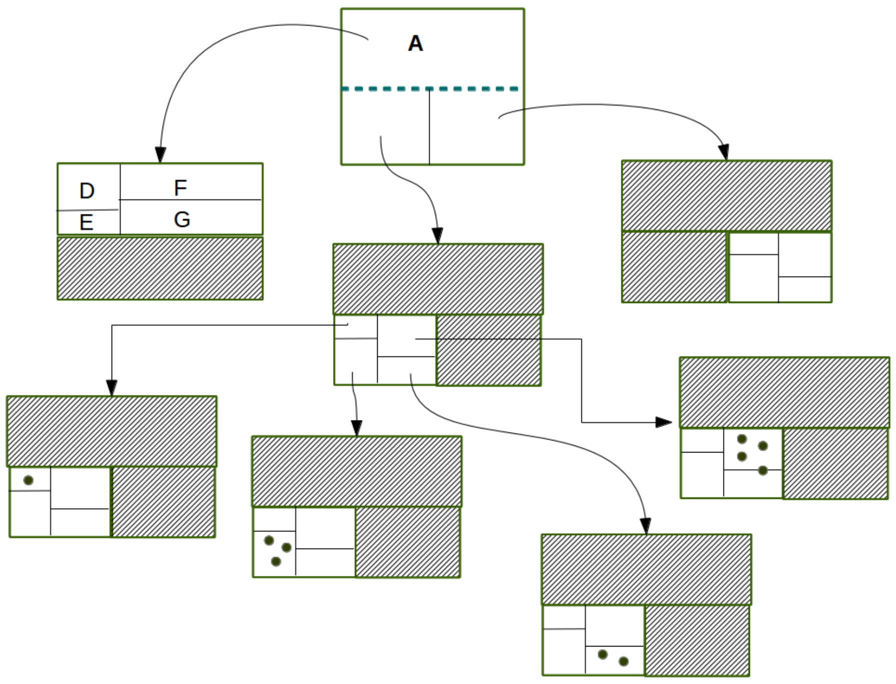


Figure 4: A representation of the KDB-tree (created based on [56])

always on different dimensions cyclically. Instead, the dimension including more range queries is selected to be the discriminator in each level. For example, if the discriminator at the first level is x, at the second level it can be x again instead of y in a two-dimensional KD-tree. The partitioning in this tree can happen in an arbitrary hyperplane like a line in 2D in contrast to KD-tree that a data point is always chosen for partitioning. In this partitioning, almost-balanced subplanes are desired.

KDB-tree. Another issue to be discussed here is storage efficiency of indexing. Since the size of indexing can grow fast enough to fill a huge chunk of main memory, most of the index structure must be stored necessarily on disk while being used. To make efficient use of the secondary memory, the KDB-tree [56] is proposed to provide the search efficiency of a balanced KD-tree together with the optimized storage access taken from the B-tree. It partitions the areas similar to the KD-tree and stores each node as a page similar to the B-tree. Pages exist in two types, region and point pages. A region page as an internal node contains a description of the bounding region enclosing its children and reference to the children. A point page as a leaf node carries the object identifiers. Therefore, different pages are kept in different parts of memory instead of a huge chunk.

As a property of the KDB-tree structure, the regions in every region page are disjoint and their union is the parent region. Similarly, all points in a point page as a child of a region are placed necessarily in that region. As shown in Figure 4, D, E, F, and G subspaces are disjoint and shape together the region A as the parent node. Here, the black small points in the leaf nodes are representing the points belonging to a page. Also, the hatched areas in each node are the regions that are not included in the page of that node. As Robinson [56] states, the KDB-tree represents a more efficient search structure for large multi-dimensional dynamic structures with optimized storage utilization. It should be mentioned that there is a trade-off between being height-balanced and storage efficiency. It means that the storage efficiency can be even poorer in some cases.

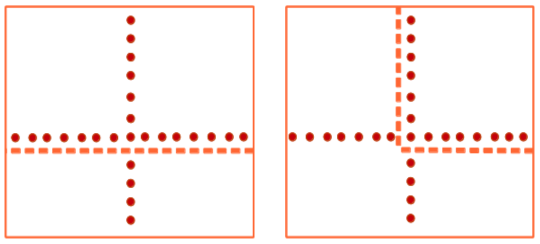

(a) Single-dimensional split of data points (b) Two-dimensional split of data points

Figure 5: Partitioning of data space in the HB-tree (based on [15])

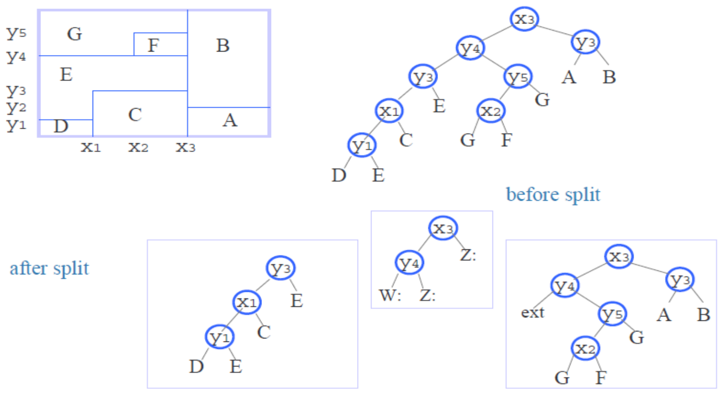


Figure 6: A representation of the HB-tree (source [44])

HB-tree. Splitting the region nodes is inevitable for keeping the balance of KDB-tree during the modification processes. This splitting is done by intersecting the data space with a plane. This intersection does not always cross the actual borders of current subregions. Then, new subspaces would be produced. This means that that splitting can propagate to the children, resulting in more and more sparse nodes in lower levels. The HB-tree (holey brick B-tree) [44] as a multi-attribute index structure is proposed to solve this problem. This structure allows data spaces to be divided along more than one dimension such that rectangular regions inside that space can be shaped, called holes. This gives more flexibility in partitioning and serves as a solution for the cases in which partitioning a region along one entire dimension forces splitting of the children subregion.

The HB-tree also helps to distribute the objects in subspaces more evenly when it is not possible by cutting across one dimension. Figure 5 illustrates an example at which partitioning along one dimension in 5(a) results in uneven division of data, while two-dimensional split in 5(b) divides the objects more evenly.

To represent the holey regions, the HB-tree uses the KD-tree within internal nodes to organize the information about the lower levels. It means that a more complex data structure is required to represent a node. In addition, it might happen that a region is referenced by more than one leaf of the KD-tree in internal nodes of the HB-tree, such as node G in Figure 6. As it can be seen in this figure, splitting along y on the third level results in dividing into two subtrees, one having 1/3 and the other 2/3 of the nodes. To achieve this, the subtree with node x_2 shows the split along x in the next level and therefore duplicates G in both right and left subtrees. The subtrees with root nodes x_1, x_2, y_3, and y_5 are extracted to show F and C regions.

As Lomet and Salzberg [44] state, removing the sparse nodes of the KDB-tree gives better search and insert performance and decent space utilization in the HB-tree. Nevertheless, expensive deletion and splitting as a result of complex structure, and multiple references to a data node that may lead to more than one traversal of a path are the trade-off.

Matsuyama's KD-tree. Since KD-trees are designed as a point access method, the variant structure introduced by Matsuyama *et al.* [47] is suitable for non-point objects via adding an extensive duplication strategy. In order to achieve this, a data page is associated to each leaf node containing pointer to the objects. The pointers refer to objects either totally or partially included in the data space. Those objects that overlap several unpartitioned spaces will be referenced in all corresponding pages. As a disadvantage, it should be mentioned that this structure is not appropriate for data with large objects due to duplication.

4D-tree. Another extension of the KD-tree to make it useful for two-dimensional rectangular objects is the 4D-tree [4]. This data structure maps objects into four-dimensional points which are indexed in the KD-tree. It means that a rectangle (x_1, y_1), (x_2, y_2) is considered as the point (x_1, y_1, x_2, y_2). Similar to the KD-tree, the discriminator in each level is chosen repeatedly from the set (x_1, y_1, x_2, y_2). For each node, a discriminator, a discriminator value, and pointers to two children are stored. For a region search (qx_1, qx_2, qy_1, qy_2), depending on the discriminator, the result of one of the following operators,

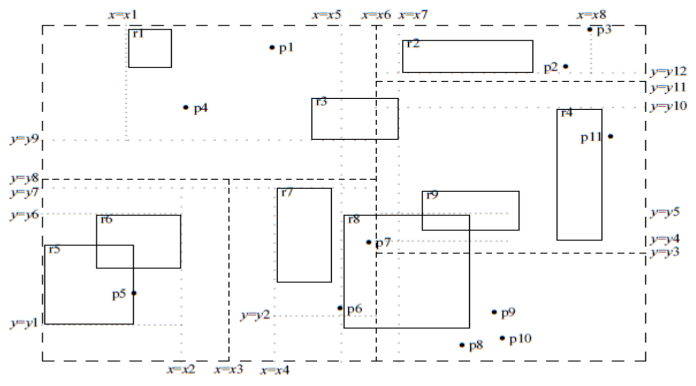
$$x_1 \leq qx_2, \quad x_2 \geq qx_1, \quad y_1 \leq qy_2, \quad y_2 \geq qy_1,$$

determines which subtree (or both) has to be searched. The major problem of this approach is the high cost of its intersection search when the query needs propagating in both subtrees.

SKD-tree. The Spatial KD-Tree (SKD-tree) [50] alters the KD-tree, such that the object duplication and object mapping is avoided. Similar to the 4D-tree, it was proposed to handle non-point spatial objects since they may extend in more than one subspace in the KD-tree. To prevent those objects to be divided and referenced multiple times, the SKD-tree defines a virtual subspace for each subspace produced in the KD-tree. Each virtual subspace determines the bounding area of all objects with the centroid inside the original subspaces. It means that even though an object is not totally contained in the original subspace, it can be contained in the corresponding virtual subspace. More clearly, the objects will be placed in subspaces based on their centroid. Therefore, each node has the following parts [49],

- pointers to two children,
- the level discriminator,
- the level discriminator value, and
- the maximum and minimum values of the left and right subtree objects respectively in the dimension that the level discriminator belongs to.

Figure 7 shows an example of a data set and the space partitioning together with the corresponding SKD-tree. Figure 7(a) illustrates the way that minimum and maximum boundaries of each space are determined to shape a virtual subspace. In this example, the discriminator for the root node is x with the value of x_6. The maximum boundary of the rectangle r_3 is x_7, which defines the maximum boundary of the left virtual subspace. Correspondingly, the minimum boundary of the right


(a) Space partitioning in the SKD-tree

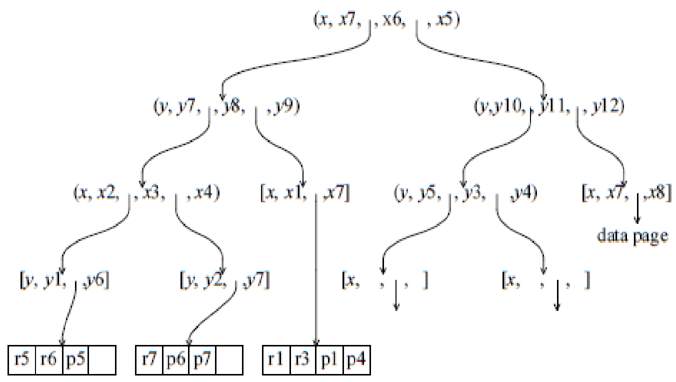

(b) A 2D directory of the SKD-tree

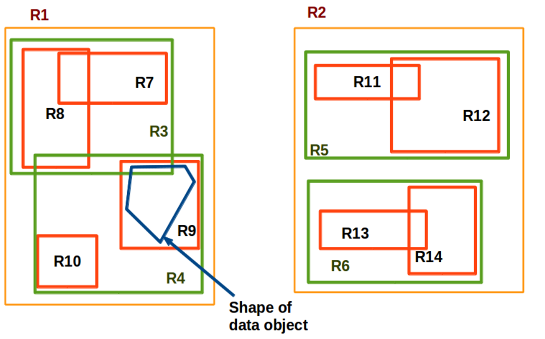
Figure 7: An example of the SKD-tree structure (source [19])

subspace is x_5, which is the right boundary of rectangle r_8. Figure 7(b) shows the tree directory of this sample data. The SKD-tree is at the advantage when the virtual subspace may bound the objects tighter than the partitioning line. In such cases, the intersection search cuts the search space efficiently. The containment search is directly supported in this structure and is useful to operators, like *within* and *contain*, since all objects, which are not contained totally in the query region, are ignored. As the SKD-tree is memory-based, not disk-based, it is not suitable for very large databases.

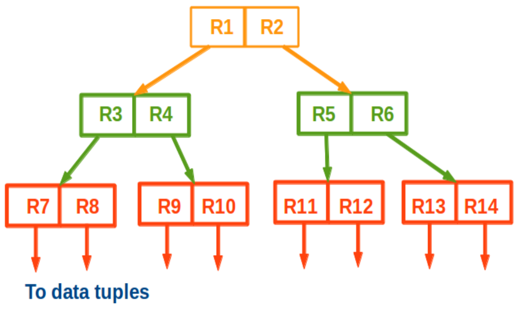
2.3.2 B-Tree Based Indexing Technique

The B-tree is used extensively in databases due to its flexibility and ability to handle a large amount of data. However, it is not designed for data in multi-dimensional spaces. Therefore, it is not suitable for spatial searches like other classical one-dimensional index structures. Frequent queries in geospatial applications, such as finding the all places within $20km$ of a point, make it important to find the objects according to their spatial location in a space. Regarding the utility of B-tree, a variety of index structures have been developed based on its structure and focusing to meet the spatial requirements. In the following sections, we discuss some of these structures.

R-tree. The R-tree [30], as one of the spatial indexing structures, extends the B-tree by adding support for multi-dimensional data. It also inherits the height balance from the B-tree. This structure is designed based on the idea of grouping the neighbouring objects and representing them with their minimum bounding rect-



(a) Minimum bounding rectangles of an object set to build the R-tree



(b) R-tree structure

Figure 8: A sample of the R-tree creation for a set of spatial objects in 2D

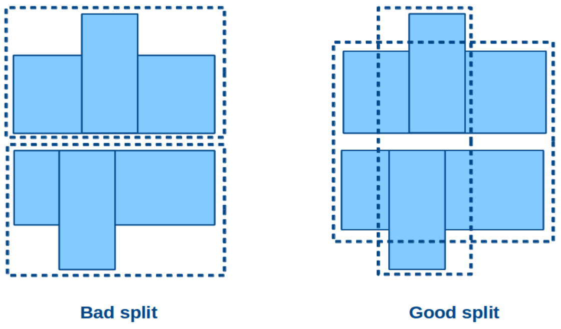


Figure 9: Two different split strategies in the R-tree (created based on [30])

angle (MBR). The idea minimizes the visiting area in spatial queries. Shapes and objects are described in leaf nodes by rectangles containing a single object. Each non-leaf node in higher levels is the smallest bounding rectangle spatially containing its children, either the rectangles or objects. The root node with at least two children is the biggest minimum bounding rectangle, which spatially contains the whole objects. Figure 8 depicts how an R-tree is built for a spatial data set. Figure 8(a) illustrates the minimum bounding rectangles calculated for different levels and the corresponding R-tree is shown in Figure 8(b). As this structure is designed for disk storage, the leaf nodes in Figure 8(b) point to the data tuples on disk.

The operations on the R-tree are basically the same as in the B-tree. The only difference is that the whole operations have to be done on bounding rectangles. Also, the order used in the element placement in B-tree is the set inclusion order on bounding rectangles and shapes in leaf nodes.

Partitioning the area with MBRs is the challenging point of the R-tree. Each split algorithm, generates different partitions such that the R-tree structure may differ for each of them. Since the rectangles are areas to be investigated in spatial calculations, minimizing both their coverage area and overlapping is crucial to its performance. Therefore, splittings should result in the minimum total area of rectangles and minimum overlapping of each two rectangles. Achieving these goals largely depends on the splitting algorithm chosen to define the bounding rectangles. Figure 9 illustrates this point by two different splits. Total coverage area of the "Bad split" case in this sample is larger than the "Good split" and this will lead to unnecessary investigation of empty spaces.

As mentioned before, the idea of bounding rectangles helps to decide which parts of the tree have to be visited in queries, such as intersection, overlapping, and

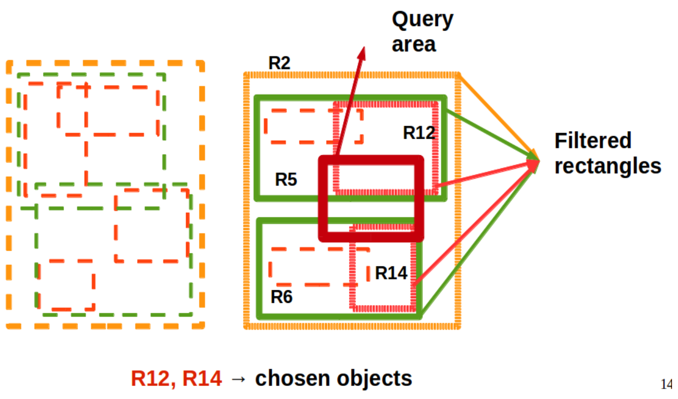


Figure 10: Querying the R-tree

containment. In fact, the following steps are taken to run such queries:

1. Filtering: This step selects those rectangles which overlap the query area.

2. Spatial Operation: The requested spatial operation is applied on the selected rectangles from Step 1.

3. Step 2 continues until it reaches those rectangles containing single objects.

4. The selected objects are checked against the query operation.

Figure 10 depicts how an intersection query is done on the R-tree and how the rectangles are chosen to be included in calculations. In this example, the rectangles R_2, R_5, R_6, R_{12}, and R_{14} are selected to be checked while the other areas remain untouched. At the end R_{12} and R_{14} are selected as rectangles containing the final results.

The R-tree benefits from the highly balanced structure and organizes the data in pages. This structure is designed for secondary storage and most of implementations often keep the inner nodes in main memory while the leaves are loaded from disk. It yields good performance in low-dimensional spaces. Generally, this structure is a common indexing technique for efficient execution of multi-dimensional queries on both point and extended spatial data. However, there are two main disadvantages according to Manolopoulos *et al.* [46], which motivate to revise the structure towards more efficient variations. First, point-location queries may lead to multiple path navigation in the tree leading to declined performance, especially when overlapping areas of MBRs are considerable. Second, investigating empty spaces of large rectangles which have significant overlaps, deteriorates the performance. Here, we briefly explain some of the most important variations of the R-tree to see how they alter the algorithm to circumvent the disadvantages. The improvements for R-trees can be classified into two groups with distinct objectives, *dynamic* and *static*. The dynamic variants effectively handle insertions and deletions on an existing tree, while the data in the static ones must be known in advance.

***R^+*-tree.** The R^+-tree [64] is introduced as a compromise between the R-tree and KDB-tree. In simple words, the R^+-tree avoids overlapping of internal nodes (rectangles) at the same level of tree. To obtain this, objects will be inserted in more than one leaf if necessary. Figure 11 shows an example of an R^+-tree together with its corresponding planar representation. As can be seen, the node e, which is partially contained in both rectangles C and B, is referenced twice in the leaf nodes.

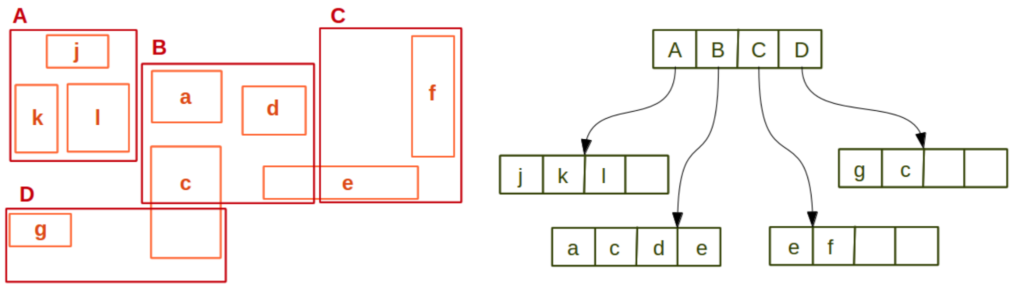


Figure 11: An R^+-tree example

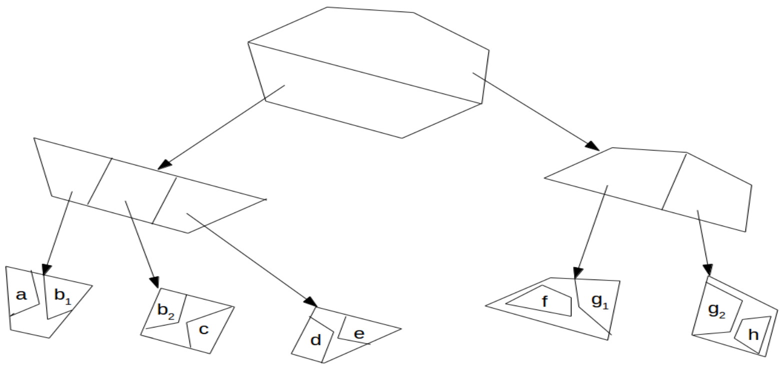


Figure 12: An Example of a cell tree (source [19])

With the changes made in the structure, the R^+-tree has some advantages. Performance for points queries is better due to the absence of overlapping. Instead of several paths, a single path is traversed. Consequently, fewer nodes are visited in point queries. On the other hand, there are some disadvantages. The R^+-tree might be bigger than the R-tree as a result of duplicating rectangles. Additionally, construction and maintenance are more complex in the R^+-tree. Furthermore, the increased complexity of the deletion algorithm is another drawback of the approach of avoiding overlaps. According to Günther [27], there is another problem in the R^+-tree algorithm, which could happen in some insertions. It happens when the covering rectangles do not allow each other to expand in order to include the inserted object. As a result, the so-called *dead spaces* are produced in the current structure that cannot be covered. It means that if a new object places in those regions, it cannot be fully covered. To cover the uncovered areas, one or more rectangles have to be split and this leads to splitting of some children as well. This case degrades the storage efficiency. However, the overall performance of this structure is improved as a result of absence of overlap between MBRs in internal nodes [53].

The Cell Tree. To avoid the overlapping problem of R-tree and *dead spaces* in the R^+-tree, the cell tree [28] was introduced. Partitions in this tree may be convex polyhedra as bounding polygons, instead of rectangles, such that the partitions do not overlap. Since some spatial objects are shaped irregularly, this method gives a better approximation of them. For example, the internal nodes in Figure 12 are polygons.

Similar to the R^+-tree, objects may be stored in more than one leaf, if necessary. This could lead to a division of new objects in order to place them in bounding polygons that do not overlap. As an instance, the object b in Figure 12 is contained in two different leaf nodes as b_1 and b_2. This will be more serious in populated databases.

***R^**-tree.** In order to achieve better performance, the R^*-tree [7] goes beyond the MBR area minimization by trying the following criteria, as Manolopoulos *et al.* [46] explains:

- Minimizing the covered area by MBRs to minimize the dead spaces, which leads to fewer paths needed to be traversed in queries.
- Minimizing the MBRs overlapping to reduce the paths, which must be followed by queries.
- Minimizing the MBR margins to shape squarish rectangles, which improve the performance of queries with large quadratic shapes.
- Maximizing of storage utilization.

In fact, unlike the R-tree, which determines the MBRs according to their area, the R^*-tree determines them based on their area, margins of space, and overlaps with other MBRs. Additionally, *Forced Reinsertion* is a key concept in this algorithm which deletes and reinserts elements of a full node to prevent splits. To find the best combination of the criteria, it is necessary to implement an optimized approach. This is important since the criteria can conflict sometimes. The optimization is obtained with a revised node split algorithm and also forced reinsertion, which finds a better place for a node than its original place. This structure results in the following performance improvements:

- More rectangular pages produced by the revised split algorithm are better for many applications.
- Better performance is gained through the reinsertion method, however it increases the complication.

As expressed by Beckmann *et al.* [7], the efficiency of R^*-tree is generally better than R-tree and some other variants, although it has slightly higher implementation cost than the R-tree.

X-tree. The X-tree, proposed by Berchtold *et al.* [12], is another extension to support high-dimensional data processing. X-trees are in fact based on the R^*-tree and focus on the prevention of MBR overlappings in order to minimize the amount of paths that have to be traversed. Unlike R^+-tree, this approach does not always split a data region. Instead, it uses a more complex algorithm by analyzing the size of overlap occurred for each split. Then, it decides on choosing the minimum overlapping or creating a new kind of node, called *supernode*. *Supernode* refers to internal nodes containing more data values and pointers than the maximum capacity of normal nodes. In fact, supernodes are constructed to contain the extra entries when there is no good split, particularly in high-dimensional spaces which overlapping becomes a serious problem. Figure 13 shows the supernodes in gray which are extended to contain six node links instead of three in normal nodes. This will help to decrease the tree levels and consequently improves the efficiency of queries.

The X-tree outperforms the R^*-tree in high dimensions as a result of increase in the number of supernodes. However, the unlimited number of children in supernodes could effect the search time and sibling overlaps will grow with dimensionality as Wang *et al.* [71] state.

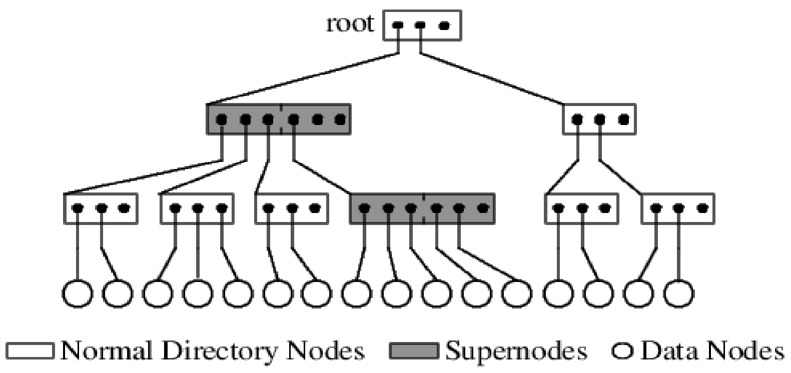


Figure 13: Structure of the X-tree (source: [12])

R-tree with packing algorithms. Suppose all the required data is available or at least does not change frequently when the tree is going to be built. For instance, in census and cartographic and environmental data insertions and deletions rarely or even never happens. For such applications, constructing an optimal structure should be focused with regards to some tree characteristics, such as maximizing the storage utilization and minimizing the storage overhead, coverage, and overlaps in the R-tree. The methods to build such an R-tree contain a preprocessing step and are known as *packing* or *bulk-loading*. The packed R-tree, proposed by Roussopoulos and Leifker [58, 59], is one of the statically constructed R-trees that has this preprocessing step. This early effort simply suggests to order the objects based on a criterion, like ascending X coordinate. Then, to build the tree, the object with the minimum value of in the ordered set is chosen to find its M *nearest objects*. Here, M is the maximum number of objects that are allowed in a page. These nearest objects form a node. The bounding boxes of these objects also shape the higher nodes. This step is repeated until the whole objects are assigned to a node and the tree is completed up to the root.

The Hilbert Packed R-tree is another method to construct a static R-tree using the packing algorithm by Kamel and Faloutsos [35]. This method, which has a full space utilization, proposes sorting the objects based on their Hilbert value of their center and then inserting them into the tree. The packed R-trees could be created with other packing algorithms. In the following, another packing algorithm is described, which we will use to create a static index structure.

Sort-Tile-Recursive (STR). Sort-Tile-Recursive (STR) [41] is the state-of-the-art packing algorithm to build a static R-tree. As we discussed before, a dynamic R-tree may not always give the best structure, in addition to the high load-time and suboptimal space utilization. Therefore, static creation in cases at which the data does not change frequently, is advantageous. The so-called STR-tree approach sorts the rectangles by their X coordinate and then places them into $\sqrt{r/n}$ vertical tiles, such that the number of rectangles in each tile is $\sqrt{r/n}$. Here, r is the number of total rectangles and n is the maximum number of rectangles that can be inserted in each node. Then, the rectangles are sorted by Y coordinates. At the end, the rectangles are inserted in tree nodes in sorted order. As declared by Leutenegger [41], this structure improves space utilization and query performance as well as the other packing structures. However, in case of data modification the whole structure must be rebuilt.

2.3.3 Quad-Tree

Another general category of indexing structures is the space filling structures that is studied by Jagadish [34]. Among this category, we present the Quad-tree [25], which is mostly used in geospatial indexing. Quad-trees are based on the principle of recursive decomposition. The goal of this tree is to design a structure in main memory, often for two-dimensional data.

In fact, Quad-tree is a class of tree structures in which the common structure, called *region Quad-tree*, is based on the subdivision into four equal-sized quadrants. It means that each internal node has four children. The spaces may have rectangular or any arbitrary shape. The leaf nodes represent the corresponding data contained in a region. The space is divided more and more in the regions in which the data is more dense. As a result, this structure is very efficient for sparse data. Several variants of Quad-tree have been introduced to obtain more optimized structures such as Quad-CIF-tree [36], MX Quad-tree [67], and a combination of both, MX-CIF Quad-tree. However, most of them satisfy the common properties, are easy to implement, and provide good performance for two-dimensional data. It should be mentioned that the performance is not always satisfying since the tree might not be balanced and main memory maintenance of the structure is tricky due to the poor I/O throughput. Also, as we discussed, this data structure is basically designed for two-dimensional data.

2.4 Summary

In this section, we mainly explained some important spatial indexing trees which are widely studied in the literature. Indeed, there is a huge number of indexing structures introduced in various sources over this area. Also, these structures are categorized differently in each source. However an extensive research on indexing trees was not our focus, we tried to cover a set of leading ones as an introduction in developing a proper index structure in BaseX. As mentioned before, choosing an index depends on the characteristics of system and also the functionality that the system provides. Of course the spatial characteristic of data should be considered as well. Regarding the discussion in this section, there is no optimum indexing structure for any type of data and requirements. Based on the application, each structure has own pros and cons. Together with the explanations, we provided a brief description of advantages and disadvantages for each index. Table 1 lists these structures and summarizes their advantages and disadvantages from different viewpoints. Based on the requirements, one can search this table for the suitable tree of that special case.

Among the presented indexes, the STR-tree is the one that we have selected for further implementation. Here, we explain different reasons for this selection. In general, the B-tree based structures are designed to search efficiently structures stored in disk files. Among those, the R-tree is a common and widely used technique in spatial databases according to Lee *et al.* [40]. Focusing on the packed R-trees, they have a preprocessing stage, which results in 100% space utilization and better query time (see [40]). The STR-tree as one of these packed R-trees is counted as an advanced one and is effective for datasets with few changes. Also, Papadopoulos [53] states that the STR-tree generally outperforms the previously proposed bulk-loading methods except in some minor cases. Besides, this structure is also included in the widely used JTS library, which we will use in the implementation of the Geo Module. Since the number of open-source libraries providing these tree structures is small and we do not aim to implement a structure from the beginning, the STR-tree package in the JTS API, which is compatible with BaseX Geo Module is a good option to choose.

Another tree structure in the JTS library that can be considered to implement is the Quad-tree. In contrast to the STR-tree, it is particularly optimized when the data set changes dynamically. However, this is not the focus of this study and is considered as future work.

Indexing Tree		Advantage	Disadvantage
Based on the Binary-tree	KD-tree	Simple implementation, Good performance for 2D if the tree is balanced	Not balanced, Storage inefficiency, Poor deletion
	Non-Homo KD-tree	Optimized deletion	Not balanced, Storage inefficiency
	KDB-tree	Balanced, Paging Technique, Storage Efficiency	Trade off between height-balanced and storage efficiency
	HB-tree	Handling sparsity, Better search performance, Better insert performance, Decent space utilization	Expensive deletion, Multiple reference to the same data
	Matsu.'s KD-tree	Non-point data indexing	Duplicate Storage
	4D-tree	Non-point object indexing, Avoiding object duplication	High cost of its intersection search
	SKD-tree	Non-point object indexing	Memory-based
Based on the B-tree	R-tree	Highly balanced point and non-point objects indexing	Large covered area and overlapping by MBRs, Multiple path traversal in point queries
	R^+-tree	No overlapping of MBRs, No multiple path traversal	Rectangle duplication, Bigger structure, Complicated structure, Complex deletion process, Dead spaces
	Cell-tree	No MBR overlapping, No dead spaces	Object duplication
	R^*-tree	Minimizing coverage area, overlapping, and margin of MBRs, storage utilization	Slightly more cost than R-tree
	Packed R-tree	Only one time-consuming preprocessing step	Poor performance in frequent data modification
	STR-tree	Only one time-consuming preprocessing step	Poor performance in frequent data modification
	X-tree	No overlapping of MBRs, Good performance	Increased search time in high dimensions
Space filling	Quad-tree	Efficient for sparse data	Poor performance in high-dimensional data

Table 1: Summary of the index structures studied in Section 2.3.

3 Related Work

Investigating the various concepts of geospatial processing in XML databases helps to gain an insight into further solutions and developments. Based on our investigations, we will mostly be discussing three main areas: storage, querying, and indexing. There is a large amount of researches and literature in these topics from which we summarize selected ones here. Besides, the geospatial processing issues in three famous databases are reviewed, which might be beneficial to design the properly fitting solutions in BaseX.

Since GML data type is our focus, we mainly concentrate on the topics related to this standard. Section 3.1 introduces the approaches regarding the storage of geospatial data in GML. These approaches are investigated with respect to the effectiveness of storage models in terms of query processing. The proposed querying ideas will be explained in Section 3.2. Finally, we will look at the databases in Section 3.3 and how they handle the geospatial data and provide the related features.

3.1 Storage

One of the basic approaches to store GML data is proposed in Li *et al.* [43] to store both spatial and non-spatial data in a spatial database, like Oracle Spatial and PostGIS/Postgres SQL. This approach first generates the schema tree and then maps it into the relational schema to store all spatial objects as the values of table fields in the spatial database. Geospatial queries can be submitted in an XQuery-like language with spatial functional extensions. These GML queries are translated into the equivalent SQL queries which are evaluated using the spatial database management system. A similar approach is discussed in Xu *et al.* [76] to store and query the GML data focusing on non-schema documents. After the generation of GML parse tree, the nodes are analyzed and schema mapping is generated to store the document in an object-relational database. Both spatial and non-spatial queries are also supported in this approach.

In contrast to the mentioned approaches, Zhang *et al.* [73] considers the characteristics of XML databases and stores the GML data in the original format without any conversion or mapping to a relational or object-oriented database. It also supports the self-descriptive and semi-structural characteristics of XML. This is only one of the many strategies, which propose a native XML database, considering the GML structure in storage and querying. In general, these kinds of strategies follow the below steps,

1. import and parse the GML schema,
2. map the feature object and document schema,
3. establish the corresponding collection in the database,
4. import the GML document according to the parsed schema,
5. partition and store the GML document into the corresponding collection, and
6. record the log file containing the GML partitioning information.

Zhang *et al.* [73] as an example of these approaches develops a prototype system based on *Java API for XML Processing* (JAXP) and JTS library to implement the above-mentioned steps. This prototype contains three building blocks: the schema mapping constructor, the document storage tool, and the spatial analyzer.

Another challenging issue in storage is the size of GML data that is commonly huge. We present one example of several research papers in this area to complete the discussion over the storage. Wei [72] discusses a query-friendly compression of GML file in SAX document parsing, which does not consider the full decompression for both the direct and the spatial path querying. Briefly, this approach generates three different structures from the SAX parsing event, which are an event hierarchy, an event dictionary, and a binary event tree. All these three structures together with the document contents are stored in a compact representation to be used in partial decompression for querying.

3.2 Geospatial Query Languages

Numerous contributions have been done toward introducing a GML query language. Here, we summarize just a short list of them trying to cover the various ideas we have met. Section 3.2.1 starts the discussion with the XQuery based languages which extends the XQuery to process the GML data. Then, we bring related work regarding the integration of geospatial and non-geospatial data from different sources in Section 3.2.3. This integration is a matter of interest as the data is mostly is a mixed of both types. The knowledge- or ontology-based approaches come next in Section 3.2.2.

3.2.1 XQuery-Based Languages

XQuery language, recommended by *W3C*, as a powerful query programming language for structured and non-structured data, is not applicable to spatial data, as discussed in [42, 18]. However, Li [42] concludes that XQuery is suitable to be expanded in order to develop a spatial query language. It is emphasized that the GML data should be treated different than XML data. Hence, Li [42] chooses XQuery as the base language to expand into a GML query language. The proposed language supports not only predefined GML elements but is also flexible to handle various GML data. It means that a query can be designed without knowing the concrete elements of the GML document. Code 3 is an example of this language and its flexibility, which retrieves all geometries within a region. As can be seen, the common vocabularies such as *feature* and *geometry* are used and no concrete tag names or data types are mentioned. In addition, a set of operators and functions is appended that covers the typical queries over the geospatial data. In comparison with similar contributions, this approach is applicable not only for predefined GML types, but to any type.

Code 3: A sample query of a flexible query language.

```
for $feature in gfn:getFeatures(doc( sample.xml ))
where gfn:within(gfn:getGeometry($feature), gml:Envelop(...))
return $feature
```

Another contribution that supports the idea of integrating GML and XQuery is presented in Chen [18]. It introduces a new GML query language, named GXQuery. Adding spatial data types and operations to XQuery, based on the OGC Simple Features Specification for SQL, Chen [18] represents the architecture and implementation methods of GXQuery which supports spatial, non-spatial, and mixed spatial and non-spatial queries. The GXQuery engine is composed of XQuery engine, GML parser, spatial extension module and a module of GML index interface. A component based on the JTS API is implemented in this engine to be used as the base of spatial extension. A typical spatial and non-spatial query example of GXQuery is given in Code 4. This query checks if two geometries corresponding to the “Building” features from “Campus.gml” file are spatially equal. This file contains the data representing the facilities of a university campus.

Code 4: A typical example of combined spatial and non-spatial query in GXQuery.

```
for $var1 in doc( Campus.gml )//Building,
    $var2 in doc( Campus.gml )//Building
where
    $var1/gml:name/node()= Library and $var2/@fid = l0028
return geo:Equals($var1, $var2)
```

A final example of XQuery-based GML database is Fubao [75] which expands the data model, algebra, functions, operations, and formal semantics in XQuery to achieve a GML query processor. New spatial datatypes and operators are added to XQuery to support both spatial and mix (spatial and non-spatial) queries. Unlike the previously mentioned approach in Chen [18], supported data types are limited to Geometry, Coord, Coordinates, Point, LineString, LinearRing, Polygon, Box, GeometryCollection, MultiPoint, MultiLineString, and MultiPolygon. These data types are actually the predefined ones in this method. The spatial operations which are defined in this method are as follows,

- simple geometry operation, such as calculating the envelope, boundary, or dimension of a geometry,
- spatial relation operation, such as finding the overlapping, containing, and touching relations of geometries,
- spatial analysis operations, such as finding the union, difference, or intersection of two geometries, and
- geometry-specific operations, such as finding the start-point of a line or the interior ring of a polygon.

The spatial computations are handled again by the related JTS functions in background.

3.2.2 Knowledge- or Ontology-Based Approaches

In addition to the XQuery-based language developments, other concepts of spatial query languages have already been developed. Alemdros [2, 3] defines a semantic version of XPath which is not based on the tree-based (syntactic) structure of GML. Instead, a system is developed to implement a query language based on the semantic structure of GML. More clearly, the GML schema is used to define the queries. However, a semantic content can be structured in several syntaxes. The system stores GML by PostGIS database system and transforms the XPath-based GML queries into the SQL language, considering the GML schema in order to execute semantic based queries. The results of these queries are visualized by exporting into the KML format.

Gutièrrez *et al.* [29] also brings a new idea and presents a knowledge-based approach to query heterogeneous spatial databases by identifying not only the entity classes but also the similar instances. It means that the database uses a conceptual schema to do the queries. The queries based on the ontology and conceptual similarities are transformed into a formal specification of entity classes which are compared against the definitions in the database. This process is carried out by determining the conceptual similarity between entities in a user ontology and by comparing the entities in ontology with the entities in conceptual models of databases. In addition to the specification of the idea, the system architecture, the user ontology, a conceptual schema are provided in this effort.

Another important task that is handled by Ontology-based queries is to query the data of different format and from various sources, like legacy data stored in databases, shape files, or even the feature data in Web Feature Services (WFS). Since a huge amount of the geospatial data is increasingly provided distributed over sources, a sharing accesses strategy is extremely needed. Zhao *et al.* [74] presents a method to spatial data interoperability at semantic level by an interface in RDF ontology. There is no need to replicate legacy data stored in relational databases, shapefiles, or GML data accessible through WFS. Instead, the queries are written to WFS *getFeature* request and SQL statements in databases. This interface provides an ontology layer for geospatial data accessible through WFS services or databases. The queries are written in the common terms of the domain and application ontology. The interface uses the spatial query functions of WFS servers for feature rendering and relational databases as sources for non-spatial data, since this can improve the performance of non-spatial queries.

As a completion to the discussion from the previous paragraph, a work is presented here to discuss the whole idea of the geospatial processing from web. The web technology, which is widely used nowadays, was an area investigated by Córcoles [21] for the geospatial data. This effort proposes an approach to integrate geospatial data on the Web by focusing on the definition of a mapping between the ontology and the DTD/Schemas. Geospatial data in this work is stored in GML format and a query language is designed for querying each GML document in the same way. Moreover, ontologies solve the semantic heterogeneity of different GML documents by defining a catalog in RDF, such that this catalog establishes a correspondence.

3.2.3 Integration of Spatial and Non-Spatial Data

Toward integrating spatial and non-spatial data from different sources, it is necessary to develop an integration system. Córcoles [21] discusses a prototype of an integration mediator for querying the spatial XML resources. This approach mainly provides an interface or global schema for querying the resources with different schema. This global schema is a simple object-oriented structure in RDF described as an ontology. More clearly, this work provides the infrastructure for formulating structured spatial queries considering the conceptual representation of a specific domain, like the information of New York, in the form of an ontology. The mediator translates the queries in terms of RDFS to queries that refer to the schemes of GML resources.

Regarding the integration issues, Belussi *et al.*[9] explains a problem arised from the different representation of spatial data in various resources or even in integration scenarios or architectures. For example, one dataset represents roads and bridges as regions, another dataset represents roads as regions and bridges as lines, and a third dataset represents both as lines. In case the user wants to query this spatial data integrated with non-spatial data, there will be indeed some gaps in the stored data. Here, this gap might cause difficulties in querying as well as some inaccuracies in the results. This integration approach introduces a solution based on a query relaxation mechanism to return approximated results with possibility of specifying maximal error allowed in the execution by the user. In particular, some relaxed topological predicates, called *weak*, and the related application contexts in terms of application scenarios are presented to show their usability. Moreover, an existing XQuery-based GML query language is extended to support similarity-based queries through the proposed operators and to handle the *weak* topological predicates, from the syntax and implementation perspective. This language uses also the JTS API to provide spatial object model and fundamental topological functions and relations. Below, sample queries in this language are shown:

- Determine all roads overlapping some bridges.

  ```
  for $x in document(bridge.xml), $y in document(road.xml)
  where overlap($x/geometry, $y/geometry) = true
  return $x
  ```

- Determine all roads overlapping some bridge, up to a 22% error.

  ```
  for $x in document(bridge.xml), $y in document(road.xml)
  where overlap($x/geometry, $y/geometry,R,L,0.22) = true
  return $x
  ```

3.3 Geospatial Processing in Databases

Besides analyzing various general ideas of geospatial processing in databases, it would be advantageous to know the geospatial features and functionality in well-known database systems. In this section, we briefly introduce the geospatial issues in the widely used databases, MongoDB, eXist, MarkLogic, and MonetDB as all of them have geospatial functionality. We will see how the different ideas are provided and used in practice.

3.3.1 MongoDB

MongoDB [54, 5] is one of the leading NoSQL databases used especially in web applications. Here, we consider the 2.4 version series of this software. MongoDB supports both flat and spherical space in geospatial processing. Geospatial features in MongoDB are mainly executed by a few operators, which can be combined to produce more advanced queries based on the use cases. The operators are *$geoIntersects*, *$geoWithin*, *$near*, and *$nearSphere* which first indicate the query operation. Then, the geometry specifiers serve to form the query conditions. These specifiers, *$geometry*, *$maxDistance*, *$center*, *$box*, *$centerSphere*, *$polygon*, and *$uniqueDocs*, define the various geometries and conditions in GeoJSON or in legacy coordinate pairs to filter the results. The calculation units in the spherical space can be meter or radiant. The units in the flat space are determined by the given coordinate system.

Apart from the operators, some particular geospatial queries can be done through the geospatial commands, *geoNear*, *geoSearch*, and *geoWalk*. The command *geoNear* is an alternative to the *$near* operator which returns additional information, for example the distance of each returned item from the given point or some troubleshooting diagnostic information. The command *geoSearch* is an interface to use the Haystack index and *geoWalk* is an internal command.

The Geospatial Index. The geospatial index in MongoDB is based on the *geohash* [37] data structure. Regarding the importance of geohash, we shortly explain it before discussing the index in detail.

The geohash value is calculated by recursively dividing the space into buckets of grids and assigns a two-bit value to each part (Figure 14). This number, called geohash value, does not really point to the coordinates, rather the bounding of the coordinate is represented. As the division goes further, the geohash values are becoming longer and the areas are becoming smaller. At the end, we will have a hierarchical spatial structure. Here are some properties of geohashing:

- It provides a short, concise, expressive, and as accurate as necessary representation of locations that have not to be a precise point.
- It simply groups the points by giving the whole places of a quadrant the same value. Nearby places often share similar prefixes. Conversely, the longer a shared prefix is, the closer the two places are.
- Arbitrary precision and the possibility of gradually removing characters from the end of the code to reduce its size and precision is properties that geohash offers. Therefore, it allows to zoom into the areas to select the level of precision.
- It facilitates the unique identification for points and locations.
- It could represent databases as data points. All data points belonging to a grid square can be cached using the related geohash value.
- For finding the nearest neighbor, an index on geohash which sorts them alphanumerically, returns the result directly, as the closest string is the nearest point. However, here we can spot a problem with finding nearest neighbors which will be discussed in the following.

Using geohash as a structure for storing, the data structure in database has some advantages based on the above-mention properties. It saves the resources due to storing the two values of longitude and latitude in a single string value without any space and punctuation signs. In addition, having geohashes as strings is an important issue which some of the advantages are as follows:

- The length of the string correlates to its precision.
- Strings can be grouped by prefix.
- They are easier to store in databases.
- They are faster to query.

On the other hand, the indexed data with geohash are positioned in adjacent quadrants and consequently the queries will be faster when tracking them down with the single index than multiple-index on longitude/latitude values. This index can be used in quick proximity searches which do not have to be precise.

The Geohash algorithm has a shortcoming when it attempts to find the near points based on the common geohash prefix. When points are close but located on

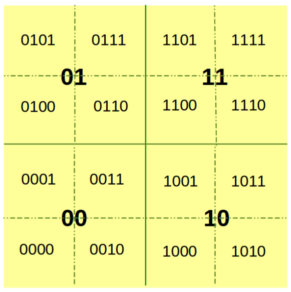


Figure 14: Representation of geohash values for eight quadrants

opposite sides of a common boundary of two quadrants, they have not only distinct prefixes but wildly distinct geohashes, such as points located on the other sides of the Equator or a meridian. This would cause inefficient proximity searches. To solve this problem, the geohash of the eight surrounding quadrants must be calculated to find the matching locations residing on both sides of boundaries. Another problem comes from the projection-based model of the geohashing. This means that a geohash of a specified prefix length will represent the areas with big difference in size near the pole than near the Equator. Rogers [57] discusses the limitations of geohashing and possible design aspects to improve the efficiency.

Using the geohash values, MongoDB provides two special indexes, *2d index* for planar geometries with legacy coordinate pairs and related calculations and *2dsphere index* for spherical calculations. The 2d index is required when we need an index structure for the flat data and 2dsphere index must be chosen for the spherical data expressed in GeoJSON, since the distance function differs respectively.

To define the 2d index, geohash values are generated for legacy coordinate pairs and the geohash values are indexed using B-Tree instead of the coordinate pairs. MongoDB also provide another 2d index optimized, called Haystack index, to return the results in small areas and is not supported by spherical query operations. To use the Haystack index in queries, *$geoSearch* command should be used. Actually, the ways which 2d and 2dsphere index could be applied, cannot access sometimes these small areas. MongoDB also uses B-Tree structure for other type of index, such as Single Field (e.g. indexing over names), Compound, and Text Index.

MongoDB supports the following general operations to be executed with geospatial indexes through the mentioned operator:

- Inclusion: for locations contained entirely within a specified polygon, which uses the *$geoWithin* function.
- Intersection: for locations that cross or intersects with a geometry using the *$geoIntersects* function.
- Proximity: for locations that are near to a specified point by the *$near* function.

The way that the index structure should be applied varies in different queries. For instance, the *$geoNear* command expects the collection to have at most only one 2d index and/or only one 2dsphere. The limitation has to be considered while using the operations or commands.

3.3.2 MarkLogic Server 7

Geospatial data in MarkLogic is encoded as XML elements and/or attributes in different representations. MarkLogic handles the geospatial data in GML, KML, GeoRss, WKT, and even a general format for geometric data that are not based on a specified format as explained in [22]. Similar to MongoDB, WGS84 and the raw coordinate systems are supported for geospatial data. WGS84 is used for data on the earth and the raw coordinate system is suitable for the data on the flat plane.

Regarding the geospatial queries, the following types are supported:

- Point query: Queries matching a single point.
- Box query: Queries searching for any point within a rectangular box.
- Radius query: Queries searching for any point within a specified distance around a point.
- Polygon query: Queries finding any point within a specified n-sided polygon.

To run the geospatial queries, numerous geospatial query constructors exist as well as geospatial operators as built-in functions to perform the operations. Here, some of these operators are listed:

- box-intersects: Returns true if the box intersects with the given region.
- circle-intersects: Returns true if the circle intersects with the given region.
- polygon-intersects: Returns true if the polygon intersects with the specified region.
- complex-polygon-intersects: Returns true if the complex-polygon intersects with the given region. A complex polygon is a polygon which is not convex or concave.
- polygon-contains: Returns true if the polygon contains the defined region.
- complex-polygon-contains: Returns true if the complex-polygon contains the given region.
- distance: Returns the distance (in miles) between two points.
- shortest-distance: Returns the great circle distance between a point and an region.
- destination: Returns the point at the given distance along the given direction from the starting point.

The region in these operations is a circle, box, polygon, linestring, or complex-polygon which is defined by a *region* constructor. The geospatial query constructors build the queries on the geospatial data. Depending on the use case, the relevant constructors and functions from the correspondent namespace should be applied. For instance, if we want to query data expressed in both GML and KML, the *geo* namespace functions are the best choice since they support the queries in all supported geospatial formats. Also for data in geoRSS, the functions in *georss* namespace could be used. This concept is shown in Code 5, which searches for the points inside the specified region.

Code 5: A query example in MarkLogic using geospatial functions

```
...
xdmp:document-insert("/points.xml",
<root xmlns:geo="http://marklogic.com/geospatial">
  <item><gml:Point><gml:pos>10.5 30.0</gml:pos></gml:Point></item>
  <item><georss:point>15.35 35.34</georss:point></item>
  <item><Dot Latitude="5.11" Longitude="40.55"/></item>
</root> );
 ...
cts:search(doc("/points.xml")//item,
 geo:geospatial-query(
   geo:box(
     <gml:Envelope>
        <gml:lowerCorner>10.0 35.0</gml:lowerCorner>
        <gml:upperCorner>20.0 40.0</gml:upperCorner>
     </gml:Envelope>)
 ))

 Result : <item><georss:point>15.35 35.34</georss:point></item>
```

Another example represented in Code 6 shows how MarkLogic returns the points in a given path which lie inside the defined region.

Code 6: A sample geospatial query in MarkLogic

```
cts:search(doc("/points.xml")//item,
  cts:path-geospatial-query("/root/item/point",
                            cts:box(10.0, 35.0, 20.0, 40.0)))
```

Primitive types are another fact in the geospatial queries which might be needed in query conditions as shown in the previous Codes 5 and 6 using the *box* constructor. These types as instances of the base type *region*, encompassing *box*, *circle*, *complex-polygon*, *linestring*, *point*, and *polygon*. For instance, the contents of the polygon element in the document “zip.xml” can be cast to a *cts:polygon* using the *cts:polygon* constructor as in the following,

```
cts:polygon(fn:data(fn:doc("zip.xml")//polygon[@id eq "712"]))
```

MarkLogic also provides functions to handle the WKT datatype. The function *parse-wkt* reads WKT into the *region* items as the following example returns a polygon.

```
cts:parse-wkt("POINT(10␣10)")
```

This point can be used in queries as any other type. Conversely, the function *to-wkt* converts *region* type to WKT.

The Geospatial Index. Geospatial index in MarkLogic is based on neither the quad tree nor R-tree. It works similar to a range index with points as data values. In this range index, every value is a pair of latitude (lat) and longitude (long). Like an array of x,y values, sorted mainly based on latitude and then longitude values. The values in this array are also connected to the corresponding document. The points would be founded easily in a sorted structure. Boxes could be found first by finding the latitude range, then checking for the longitude range. For circles and

Data	**Path**
<a:geo> <a:location>37 −122</a:location> </a:geo>	/a:geo/a:location
<a:geo> <a:location **data**:"37␣−122/> </a:geo>	/a:geo/a:location/@data

Table 2: Data and the related path for indexing in MarkLogic

polygons as well as more complex types, the geometry's bounding box is used to find the region they belong to. Also, to check whether a point is inside the polygon, the number of intersections with the northward or southward arc of the point is counted. Based on this structure, Marklogic supports the following geospatial indexes:

- Geospatial Element Indexes: With this index, the data is represented by whitespace or punctuation separated element content. The lat-long or long-lat point formats could represent a point. Other entries such as z-coordinate or altitudes will be omitted.
- Geospatial Element Child Indexes: With this index, the data comes only from the elements that are a specific child of a specific element containing whitespace or punctuation separated content.
- Geospatial Element Pair Indexes: In this index, the data is considered from specific pair of elements that are a child of another specific element.
- Geospatial Attribute Pair Indexes: With this index, the data is involved in a pair of specific attributes of a particular element.
- Geospatial Path Range Indexes: With this index, the data is expressed in the same manner as a geospatial element index and the element or attribute index is defined by a path expression. Table 2 shows two examples of the data as an element and attribute and related paths for indexing each one. It should be mentioned that a defined path cannot be replaced with a new one unless it is removed.

To speed up the retrieval of geospatial values geospatial indexes enable geospatial lexicon lookups. Since these functions are implemented using geospatial indexes, the appropriate index must be created in order to use a geospatial lexicon. As an example, a geospatial element index is needed to use the *element-geospatial-boxes*. Here is a sample to show how these functions are used and returns values from the specified element geospatial value lexicon that match the specified wildcard pattern.

```
cts:element-geospatial-value-match(xs:QName("point"),
          cts:point(10,20))

Result  :  10,20
```

For querying on geospatial data in MarkLogic, after loading the data into the database and making the indexes, primitive types should be constructed to be used in query functions. Then, geospatial queries are constructed using these primitive types.

3.3.3 eXist-db

eXist-db [68] version 1.2[3] supports geospatial data in GML 2.1.2 format. The geospatial features are provided using the JTS API in geospatial computations. The general development idea of indexing in eXist-db follows a modular architecture. The geospatial index is developed based on this model and maintained as a pluggable extension. eXist-db connects to a relational database to index spatial data. The index does not store character data from the document. Instead, WKB index entries are stored through the JDBC in the HSQLDB or any other relational database namely PostGIS.

The index creation relies on the open-source libraries of the GeoTools [20] project. Geometries are stored both in the original coordinate system and WGS84 system in index structure to allow the operations on data originated from the different systems. In this way, transformations between coordinate systems are included using GeoTools library.

To query a spatial dataset, the spatial index and the spatial module must be enabled in the main configuration file (conf.xml). The following content in a configuration file of a collection in a database configures the spatial index on GML geometries.

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index>
    <gml flushAfter="200"/>
  </index>
</collection>
```

The attribute *flushAfter="200"* says that the index entries in memory will be flushed into the HSQLDB. Now, the dataset is ready to query and Code 7 is a sample query that returns the whole polygons intersecting with the second argument of the *spatial:intersects* function.

Code 7: A spatial sample query in eXist-db using the *spatial:intersects* function

```
declare namespace gml = "http://www.opengis.net/gml";
spatial:intersects(//gml:Polygon,
  <gml:Polygon srsName="osgb:BNG" xmlns:gml='http://www.opengis.net/gml
      '>
    <gml:outerBoundaryIs>
      <gml:LinearRing>
        <gml:coordinates>
          08.278200,187600 278400,187600 278400,188000 278200,188000
              278200,187600
        </gml:coordinates>
      </gml:LinearRing>
    </gml:outerBoundaryIs>
  </gml:Polygon>
)
```

3.3.4 MonetDB

MonetDB[4] basically is a relational column-store database, which has implemented a separate module to support the objects and functions in the OGC "Simple Features Specification for SQL" [8]. The current implementation of spatial support

[3]The spatial module in eXist-db is still in experimental status.

[4]The information that is presented here, is from January 2014 Bugfix.

is still simple. MonetDB stores XML in its relational architecture as Boncz *et al.* [13] describes. The geospatial component in MonetDB wraps the open-source GEOS library [48], which is the *C++* implementation of the JTS API. When *geom*, the geospatial module in MonetDB/SQL component, is enabled, then the created databases are spatially-enabled by default. MonetDB supports the OGC geospatial types. Besides, a non-OGC type, called *mbr*, is used for storing a 2D box for fast access. Numerous functions exist in MonetDB to provide SQL-based geospatial querying. These functions are categorized in geometry constructors, functions on specific geometry, and functions on bounding boxes. Function on boxes could be used for pre-filtering in some complicated queries. Below some geospatial functions that can be used in queries are shown:

- `ST_GeometryFromText(wkt string)`, which creates a geometry out of WKT string.
- `ST_AsText(geom Geometry)`, which returns the WKT representation of the Geometry.
- `ST_ConvexHull(geom Geometry)`, which returns the convex hull of the Geometry in the Geometry format.
- `ST_InteriorRingN(geom Polygon, ringNum integer)`, which returns the n-th interior ring of the Polygon as a linestring.
- `ST_IsRing(geom LineString)`, which returns a boolean value showing whether the LineString is both closed and simple or not.
- `ST_XMax(box mbr)/ST_XMax(geom Geometry)`, which returns the maximum X coordinate of the provided bounding box. In latter case, which a 2D geometry is provided, first its bounding box is computed.
- **mbr(geom Geometry)**, which returns the minimum bounding box created for the geometry.
- @ (mbrContained function), which returns a boolean value expressing whether box1 is contained by box2 in *box1:mbr @ box2:mbr*.
- && (mbrOverlaps function), which returns a boolean value showing whether two bounding boxes overlap in *box1:mbr && box2:mbr*.

The following example returns a geometry form the *spatial* table:

```
SELECT PointFromText('POINT(' || long || ' ' || lat || ')') FROM
    spatial LIMIT 1;
```

As Vermeij *et al.* [70] discusses, MonetDB benefits from the design principles for spatial query processing in three main areas. First, the column-based storage approach keeps the non-needed geometry out of the way since the traditional tuple-based storage model stores the whole tuple physically together on a disk block. Additionally, some efficient filtering stored in the same column and some approximate geometries, e.g., minimum bounding box, maximum enclosed circle, and maximum enclosed rectangle stored in other related tables would speed up the queries as the index structure does. Second, the MonetDB query optimizer effectively improves the geospatial query performance. Third, the spatial data types integration with XML types in MonetDB is considered as an efficient way of querying.

3.4 Other Related Topics

In addition to the aforementioned topics, some other areas have been discovered as well in this research line. We briefly discuss here two ideas that might be advantageous for more geospatial features and processing in a geospatial database. The first idea, presented in the first paragraph, is about the unification of the coordinate system when the data is coming from different sources. The second idea, presented in the second paragraph, is pointing to a completely different idea of finding a path between to points in these database.

Schwarz *et al.* [63] introduced a library for geospatial data management to enable handling the different coordinate systems. This library provides geospatial functions which are delegated to JTS library. As mentioned before, JTS library only supports the Cartesian coordinate system and the input geometries should be in a common system to be processed. It can happen that the geometries have different coordinate systems. For example, it might happen when the data is provided from different sources. In order to work with JTS library, a common coordinate system is chosen to which other systems of the current data is converted. This coordinate system is chosen considering the the transformation issues to have the minimum differences from the original data and to maximize the transformations precision.

Padmaja *et al.* [52] introduces an alternative way of finding the shortest path regarding existing cost. The k-shortest path and many other path finding algorithms are efficient as long as the effecting cost factors do not change dynamically. This provided method first finds the shortest paths without considering the cost factors. At the end, the factors are used to rank the paths. All paths between each two points could be stored in database in pre-processing stage to reduce the further combinations. To find the paths, each edge in the network of the connected points should be provided with the start and end points. The process starts from the starting point and finds the whole connected ones and continues till reaches the destination. The paths would be obtained by joining the edges detected in each step.

4 Spatial Querying in BaseX

Spatial query is a special type of query that requires the processing of geometries with two certain properties in general. First, it has geometries as input and output as well as other primitive types, like double, integer, etc. Second, it considers spatial relations between the geometries. As we discussed in Section 3.3, there are various viewpoints in providing geospatial features in a database to fulfill the expected requirements. Following the OGC Simple Feature (OGCSF) [39] data model, geospatial features in BaseX adapt the specification of EXPath geospatial API function interface. This specification defines commonly used functions from the OGCSF Common Access API [33]. Since the OGCSF data model is typically represented in GML, we concentrate on adding the support of GML format in BaseX. However, an implementation could support other encodings such as KML.

Throughout the following sections, we discuss the integration of geospatial data processing in BaseX. We introduce geospatial functions as a new module, called *Geo Module*, in Section 4.1. Query efficiency is improved later in Section 4.2 by implementing an index structure. Indexing and the related time complexities are discussed hereafter in Section 4.3 and Section 4.4, as the critical issue in this topic. Finally, the concluding points are explained in Section 4.5.

4.1 Geo Module

As mentioned above, geospatial features in BaseX are implemented based on the EXPath Geo Module Specification [60]. This specification contains the definition of functions for widely used geographic and geometric analysis operations, from OGCSF Common Access API version 1.2. These functions apply to geometries in different formats, such as GML, KML, GeoJSON, Well Known Text (WKT), and even Well Known Binary (WKB). Based on the specification, Geo Module in BaseX comprises a set of functions, like intersection, within, distance, boundary, centroid, difference, and union, added to the *basex-api* package. The geometries supported in this module are Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. This module is an individual package with a set of classes as briefly described below:

- *Geo*: This class is the main class, in which all geospatial functions are defined. The set of functions in this class is provided to be directly used in XQuery statements. The complete list of functions and their description are available in the online BaseX documentation [65].
- *GeoError*: This class defines error functions with related messages which are thrown when an error occurs.
- *GeoTest*: Test functions for the *Geo* class are implemented here.
- *GmlReader*: Functions required to parse GML geometries as XML elements are implemented in this class.
- *GeoIndex*: This class implements the functions related to the geospatial index.

The geospatial index structure is implemented in BaseX core. Spatial indexing and related implementation details will be described in Section 4.2. Here, we explain general functionality of this module.

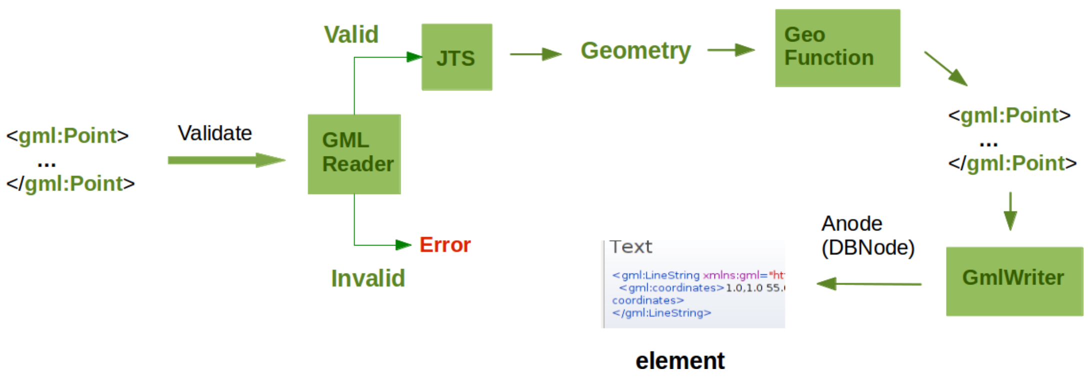


Figure 15: Converting GML elements to geometries

The geospatial functions are defined as public methods in the *Geo* class. They use some private methods to read geometries or write them out into the GUI. Besides, each function employs the corresponding function from the JTS library to do the required geometric operation and provide the appropriate result. Here, we shortly explain how the input geometries, either from a variable or a database node, are processed.

Suppose we are searching for all geometries within the specified polygon $p. The query should be written using XQuery via BaseX GUI as follows:

```
let $p :=<gml:Polygon>
            <gml:outerBoundaryIs> ... </gml:outerBoundaryIs>
         </gml:Polygon>
for $x in //gml:Polygon
  return if ( within($x, $p) ) then $x else ()
```

Each geospatial function has at least one geometry as input, as in the above function *within*. In this example, the variable $*p* is provided as a document node and the variable $*x* iterates all Polygon nodes in the current database. Various functions are involved in processing this query. In the following, this process, which is illustrated in Figure 15, is explained in detail.

To read the geometries as a document node, the private function *geo* makes sure that the node name is at least a valid geometry name. It means that if the element name is not contained in the set of geometry names, this function will throw an error. Then, the GmlReader class is used to read and parse the whole element and to check the validity based on GML 2.0 format. Regarding the JTS limitation, geometries have to be in GML 2.0 format to be validated and analyzed for further operations.

The aforementioned GmlReader class reads the elements differently based on their types, i.e., tag names. If the element is a valid GML geometry, the function creates the corresponding geometry, using JTS constructors. Otherwise, the matching error message will be shown. For instance, if a polygon does not have any outer ring or if the coordinates of a ring do not shape a closed ring, an error will be thrown.

Now, the geometries are ready to be processed in geometric operations, like:

- checking two geometries whether they intersect each other,
- finding the symmetric distance of two geometries, and
- getting the number of inner rings of a polygon.

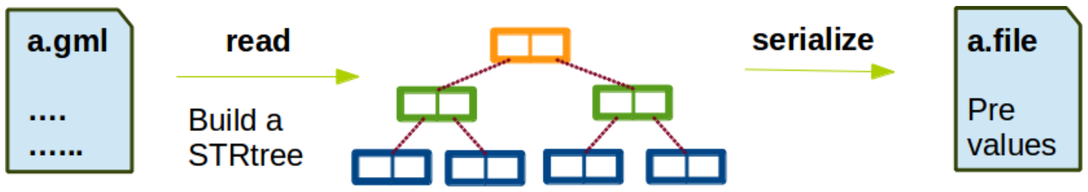


Figure 16: Geospatial index creation in BaseX

These operations are done using the JTS functions. The output could be either in any XML Schema (XSD) data types such as integer, boolean, string, and URI or GML Geometry. The XSD type are equivalent to the BaseX defined data types. For example, the primitive data types boolean, integer, and string are represented in BaseX as the classes *Bln*, *Str*, and *Int*, respectively. In addition, the Geometry type which is the output of functions like intersection, union, and difference of two geometries, must be returned as an element in the abstract node type (*ANode*) to be in compliance with the specification. Having the output geometries in GML format, GMLWriter function converts them first to a string, using JTS GmlWriter and builds a database node (*DBNode*) hereafter. Considering the point that DBNode extends ANode, the output node is sent to the GUI and is shown as a GML element, expressing the geometric result.

Now, it is time to discuss the query performance that seems not satisfying. Having commonly huge geospatial data, we encounter serious performance problems. The common solution is an indexing algorithm designed to improve efficiency. Since the geospatial data is a written form of geometries positioned in the space, an indexing structure should be designed considering the positions. In the following sections, we discuss an indexing approach and its influence on performance compared to querying without using any geospatial index.

4.2 Geospatial Index in BaseX

To enhance the geospatial query time in BaseX, we discuss an index structure. This index avoids processing the whole database when partial checking of the file would be enough. For this purpose, a bounding box is computed for each geometry by which an efficient filtering can be applied. For example, if we want to find the objects within a specific geometry, examining just the area near to this geometry is adequate. If two geometries have intersecting bounding boxes, they might intersect each other and have to be checked whether they fulfill the query condition. Otherwise, they will not have any contact and there is no need to check them. This approach, to which we will refer as two-step filtering in the following, decreases the number of scanned geometries and consequently gives the better run-time. The two step filtering is clarified as follows:

1. finding the geometries which their bounding box intersect with the bounding box of query area, and

2. checking the selected geometries against the query condition.

Among the spatial index structures discussed in Section 2.3, JTS supports the STR-tree and Quad-tree. Davis [24] discussed that STR-tree in contrast with Quad-tree cannot be updated after the generation. However, we choose the STR-tree since it fits our approach and application as we discussed in Section 2.4. As mentioned before, STR-tree has the basic structure of R-Tree with the improved performance.

The index tree, holding the bounding boxes in inner nodes and geometries in leaves, is made once when the spatial index for the database is requested. Then, this structure is stored in a file on disk. This process is illustrated in Figure 16. Each time the index is requested, the file is read into main memory and will be kept there for future requests. At the end, the two step filtering is executed. It should be mentioned that only the following queries would benefit from this index. The definitions of these functions are provided in EXPath Geo Module specification.

- intersects (geometry1, geometry2)
- within (geometry1, geometry2)
- contains (geometry1, geometry2)
- overlaps (geometry1, geometry2)
- crosses (geometry1, geometry2)
- touches (geometry1, geometry2)

The geospatial index structure is involved both in BaseX core and the *basex-api* package. The main structure of index is added as a package in core, called *index.spatial.* This package contains the index builder classes based on the core index structure of BaseX. The class *SpatialBuilder* which extends *IndexBuilder*, builds the index tree using pre-values instead of tag names to address the database elements. Then, the JTS serializer writes this tree into a file, called *STR-treeIndex.*

The other *GeoIndex* class in the *basex-api* package extends *QueryModule* and implements the method defined by the JTS STR-tree class for reading the index file from hard disk into memory. Additionally, this class implements the *filter* method to do the first step of two step filtering. The details of the index implementation is described in the following section.

4.3 Index Functions Implementation

To add the geospatial index, we thought of two approaches. The first approach implements new signature for the aforementioned functions in a new class. It means that the index filter function is encoded directly in the existing geo functions. Then, queries can be done with the new spatial function as in Code 8. In this query, the function *geo-index:intersects* in the new namespace and signature applies the index functionality inside the function. Therefore, we have two *intersects* functions.

Code 8: The geo function containing the index functions

```
import module namespace geo-index = "http://expath.org/ns/GeoIndex";
let $a:= <gml:Polygon> ... </gml:Polygon>
return geo-index:intersects("DB", $a)
```

Since having two functions with the same name and functionality is redundant, we introduce a new approach in which the *filter* function can be used in XQuery to benefit from the index structure. A sample query of this approach in Code 9, filters the geometries by the *geo-index:filter* function. Next, the *geo:intersects* function of Geo Module is applied to the filtered geometries. To summarize, the former

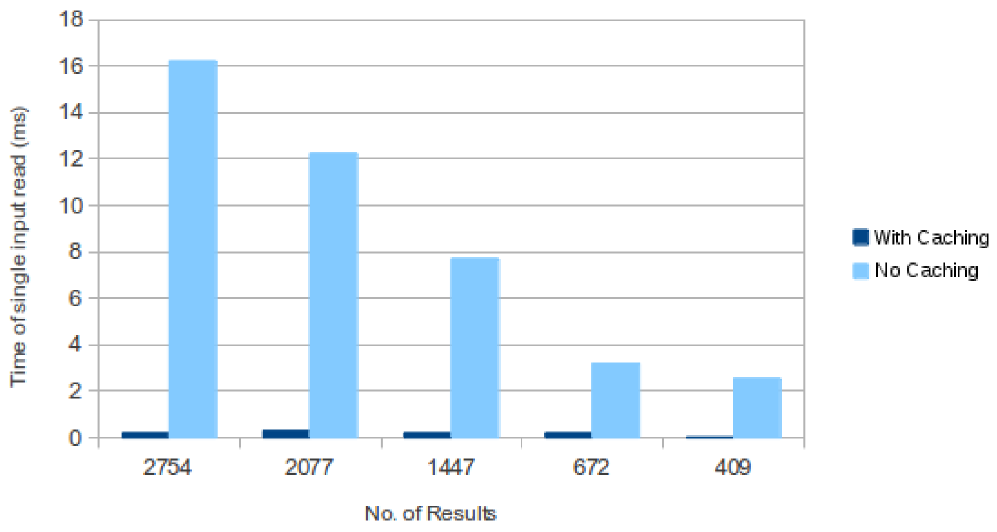


Figure 17: Using a Map to cache the input geometry: performance effect

approach does the whole process including the first step of filtering through the *geo-index:intersects* function, while the latter uses the original *geo:intersects* function together with *geo-index:filter* which does the first step of filtering.

Code 9: The geo function used together with the index function

```
import module namespace geo-index = "http://expath.org/ns/GeoIndex";
import module namespace geo = "http://expath.org/ns/Geo";
declare namespace gml="http://www.opengis.net/gml";
let $a:= <gml:Polygon>
                <gml:outerBoundaryIs>
                  <gml:LinearRing>
                    <gml:coordinates>
                      3.9,50.6 6,52.8 4.5,52.8 3.9,50.6
                    </gml:coordinates>
                  </gml:LinearRing>
                </gml:outerBoundaryIs>
             </gml:Polygon>
return ( geo-index:filter("DB", $a)[geo:intersects( $a, .)])
```

The negative point about the second approach is that the input polygon, like polygon *$a* in Code 9, is read, parsed, and created every time that the spatial function, here *intersects*, is called in the *for* loop, even though it is the same fixed object. Since, this causes redundancy and consumes a considerable amount of time, we use a hash map to cache the fixed input and prevent the further redundant readings. This will dramatically reduce the query time, as shown in Figure 17.

Figure 17 demonstrates the time consumed by different queries to read the single input geometry with or without caching. It could be seen that when no map is used, as the number of results goes higher more time is taken. In contrast, by caching the single input reading time remains constant, since the input geometry is read once. In other words, without caching the input geometry will be read each time the function *geo:intersects* is called, as it is one of the input arguments. In the next section, we assess the performance issues related to the geospatial index.

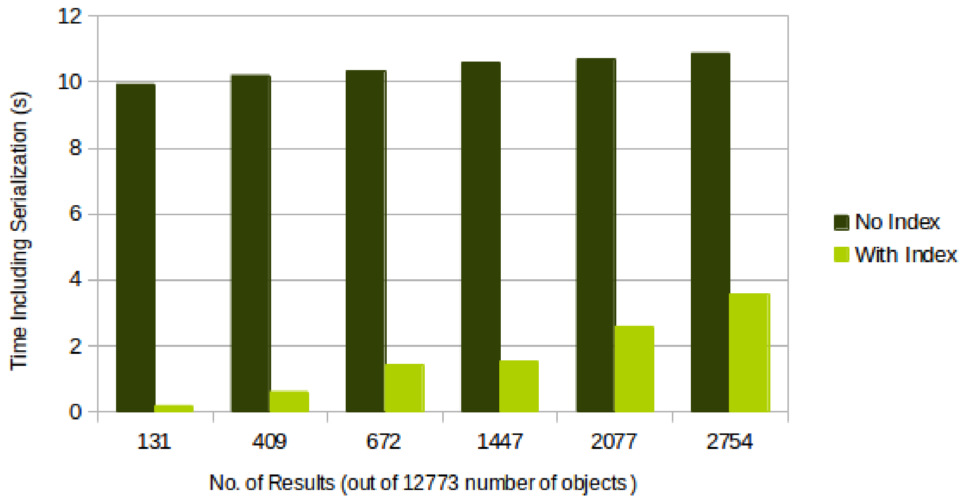


Figure 18: Query performance improvement by using geospatial index

4.4 Evaluation and Improvement

There is no need to emphasize the importance of role that indexing plays in the query time and performance improvement. Here, we use real-world data to observe the effect of currently implemented geospatial index, followed by in-depth looking at the implementation from other perspectives. The aim is to find ways in order to improve performance. This data is provided by University of Twente, Department of Geoinformation Processing and holds some real information on the earth in GML 2.0 format. The original file is based on one of the Netherlands coordinate system (RD/NAP Amersfoort RD New) and is around 133.3 MB including 12773 polygons inside 11886 multipolygons. We run different queries on this data with and without geospatial index to see time consumption dependency on the index structure.

Since queries have various number of results, we can see the trend changes in regards to the number of outputs. To start with, we take a look at the effect on index utilization in queries in comparison with queries using no geospatial index. Figure 18 represents the effectiveness of index utilization. As shown in Figure 18, when geospatial index is not used, the query time remains constant for every query regardless of the number of results. As we discussed in Section 4.2, this is due to the fact that the whole file is scanned and analyzed for each query. In contrast, queries using geospatial index relatively take more time as the number of output objects goes higher. It confirms that the filtering approach is going in the right direction, but the performance still is not satisfying. Thus, we need to investigate more to improve it.

By monitoring the times consumed by different parts of a query, we discovered that JTS GMLReader functions take considerable amount of time according to the Figure 19. This figure depicts that the Reader consumes nearly 30% of the whole query time. JTS GMLReader operates this part that reads the geometries from GML and converts them to JTS geometries. Regarding the geometry reading process by JTS, shown in Figure 20(a), it seems that the three steps serializing the XML, converting it to string, and constructing geometries by JTS can be eliminated and directly parsing approach might decline the query time. Thus, we implement a custom GML reader class to immediately parse the GML elements into the JTS geometries (see Figure 20(b)). To assess the time efficiency, a set of queries are tested and the results are presented in Figure 21. As it was supposed, reading time

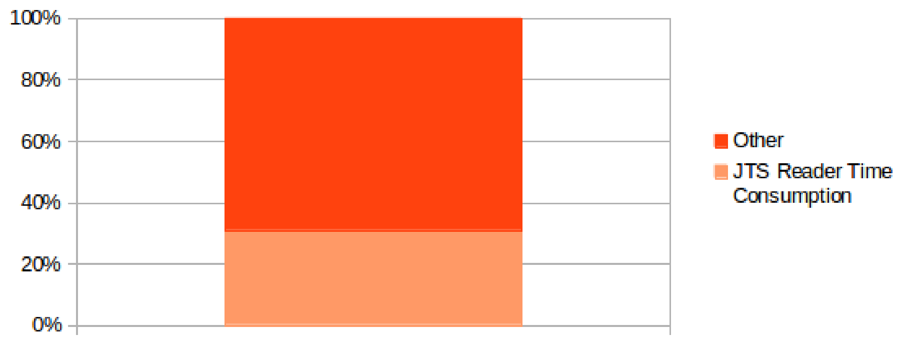
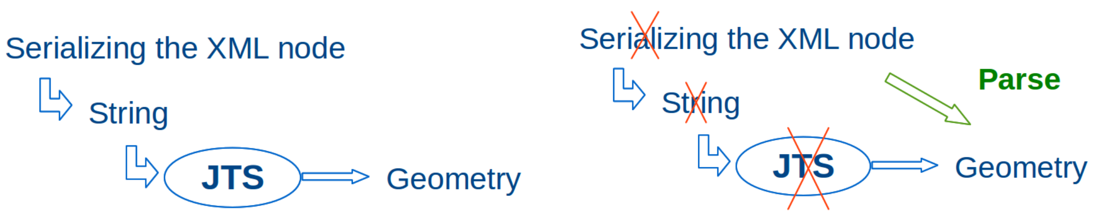


Figure 19: JTSReader time consumpion in total query time



(a) Through the JTS GMLReader
(b) Through the Custom GMLReader

Figure 20: Parsing process of the GML elements

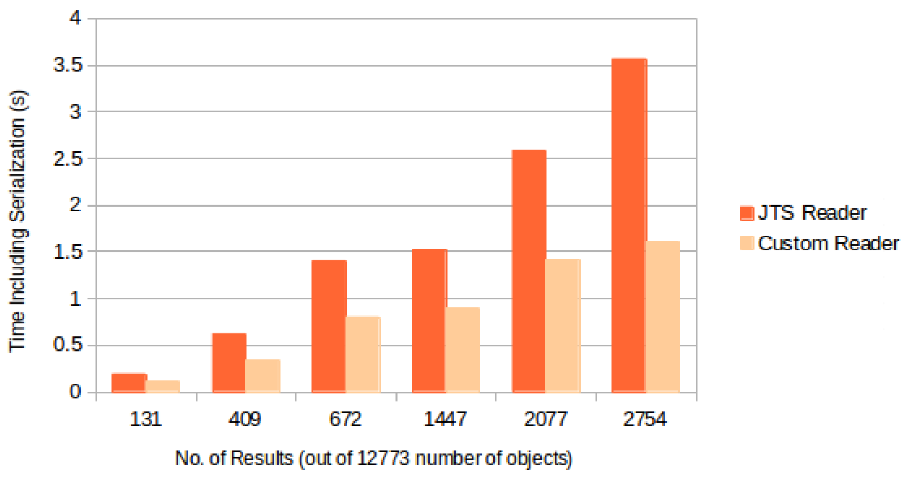


Figure 21: Custom GMLReader vs. JTS GMLReaer

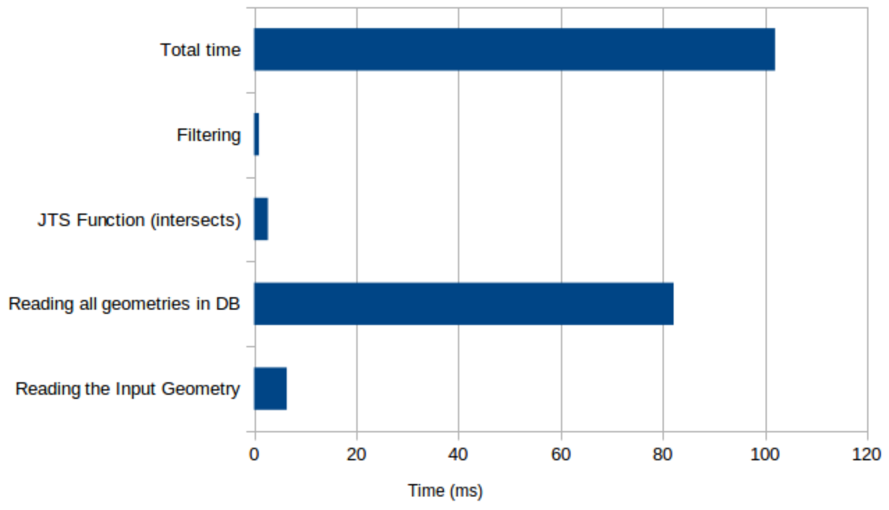


Figure 22: Detailed run-time evaluation in a query using geospatial index

drastically reduced by use of the new reader functions, although both have increasing trend proportional to the number of outputs.

A deeper look at the different parts of a query will be beneficial to find the parts where need to be improved. Suppose we run the query below:

```
let $a:= <gml:Polygon> ... </gml:Polygon>
return ( geo-index:filter("DB", $a) [geo:intersects( . , $a)]).
```

The total query time will be divided into the following parts:

1. reading the input geometry $\$a$,
2. filtering the geometries using *filter* function by pre values,
3. reading the filtered geometries from the database using selected pre values, and
4. applying *intersects* operation on the pair of input geometry and each selected geometry.

To see how each part influences the performance, we examine them separately. Figure 22 shows the times taken by the above-mentioned actions. It could be seen that the biggest amount of time is spent reading the geometries. Even reading the single input geometry seems to be expensive. Hence, the reading function should be observed more in detail.

We have used the Java profiling to get more precise information. The first few methods in profile output with the highest percentage of time occupation, ordered from the most-used to the least-used, are listed in Table 3. As the profile output in Table 3 indicates, the *split* function calls consume the greatest amount of time. Besides, *createPolygon* and *geo* functions used in *GmlReader* class are expensive. Therefore, these functions should be the focuse of performance tuning. Since the *split* function has the most frequent call, we discard it in favor of better performance. This function is used to divide the string value of *coordinate* tag regarding the delimiters to make coordinate pairs. We replaced it with a piece of code to parse this string. A coordinate value in GML 2.0 can include different points separated

rank	**self**	**accum**	**count**	**method**
1	6.95%	6.95%	94	org.basex.util.Token.split
2	5.33%	12.28%	72	org.expath.ns.GmlReader.createPolygon
3	4.59%	16.86%	62	org.basex.util.Token.split
4	4.22%	21.08%	57	org.basex.query.func.JavaModuleFunc.eval
5	3.55%	24.63%	48	org.expath.ns.Geo.geo
6	3.18%	27.81%	43	org.basex.util.Token.split
7	3.03%	30.84%	41	org.basex.util.Token.split
8	2.96%	33.80%	40	org.basex.util.Token.split
9	2.74%	36.54%	37	org.expath.ns.Geo.geo

Table 3: Java profiling output for the BaseX Geo Module

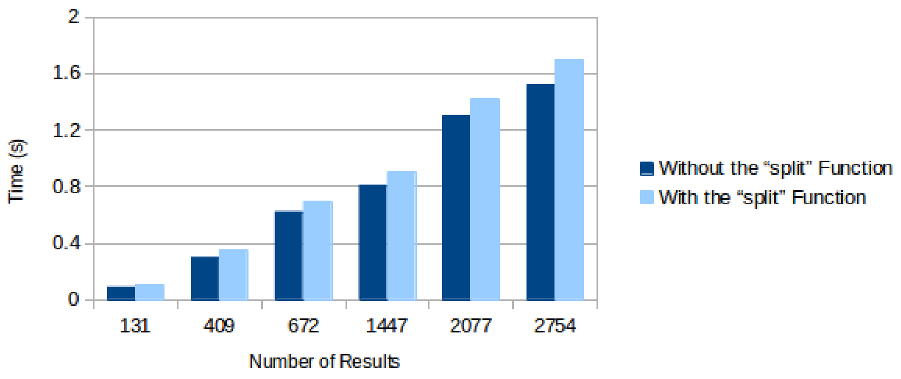


Figure 23: The query times of the selected queries before and after removing of the *split* function

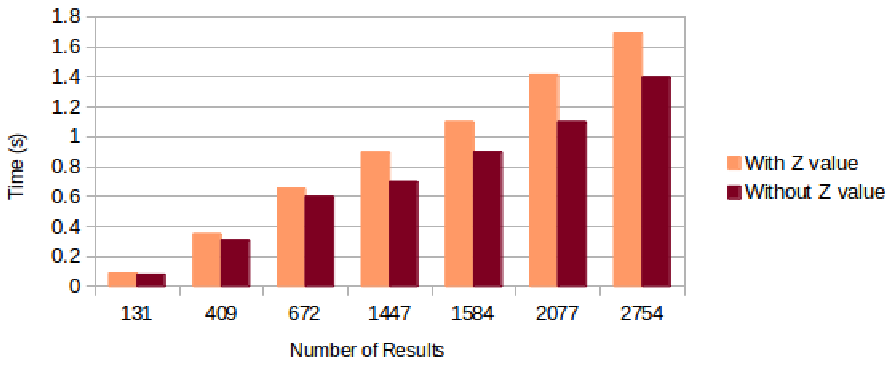


Figure 24: The effect of ignoring the Z values of coordinates in GML parsing on query times

with a space. Each point in a coordinate has two dimensions X and Y separated with a comma. Code 10 shows an example of the GML 2.0 coordinate. Based on this format, the new code reads the coordinate string and acts based on the position and type of delimiters and numbers. Therefore, the coordinate pairs are constructed from a valid string. Otherwise, an error message will be shown. Figure 23 represents the effect of removing the *split* function on the query times. This figure indicates that this effect grows as the number of results gets bigger.

Code 10: An example of GML 2.0 coordinate

```
<gml:coordinates>
   3.9,50.6  6,52.8  4.5,52.8  3.9,50.6
</gml:coordinates>
```

In addition to the *split* function, we omit the Z values in parsing process, since the Z value is ignored with the geospatial functions. It means that it is useless to read this value. It can be considered as a limitation of GML 2.0, which is improved in GML 3.0. Not surprisingly, excluding the Z values from the parsing process declines the query time that is displayed in Figure 24. This effect is minor in the smaller number of results and is more obvious with more output numbers.

Up to this point we have tried various ways to decrease the query times in the Geo Module. Further attempts could be done following the aforementioned Java profiling list (Table 3), if the implementation will be continued in the current direction.

4.5 Conclusion

In this Section, we discussed the Geo Module implementation details and assessment in BaseX, which provides a set of geometric functions based on the EXPath Geo Module specification. This module employs the JTS library to compute the geometric operations. To enhance the performance, the JTS STR-tree index structure is added that accelerates the execution time of queries such as intersects, within, inside, and touches. This is done by applying a filtering in the STR-tree structure and consequently reducing the number of processes in the database. Besides the index tree, a GML Reader class is developed to parse the GML elements directly regarding the inefficiency in JTS GMLReader. Although the performance is convincing, there is room for improvements. For example, the functions *split*, *geo*, and *createPolygon* can be modified in favor of better performance.

5 BaseX and MongoDB

An introduction of MongoDB geospatial features is provided before in Section 3.3.1. In this section, we will have a more-detailed overview of this functionality and a comparison between BaseX and MongoDB. However these two systems follow distinct approaches, it would be beneficial to review MongoDB as a rather new, recently developed, and widely used database system. MongoDB is among the few NoSQL systems providing geospatial features. Here, we will mainly see differences between these systems, particularly in performance of the same features. This comparison empowers us to improve the geospatial querying performance in BaseX.

As we discussed in Section 3.3.1, geospatial data can be represented in MongoDB either as planar or spherical maps. Since the earth is a spherical globe, the geospatial calculations on planar maps are only an approximation [62, 69]. As an example, the measurement of distances in planar maps are accurate only in a small region. It means that the spherical maps are the better representation when the calculations are expected as real ones on the earth. As our goal is to assess some geospatial features of MongoDB and compare it with BaseX Geo Module, we focus merely on the spherical approach. Correspondingly, WGS84 is used instead of legacy coordinate pairs to express the spherical maps.

Code 11: A GeoJSON file containing a *FeatureCollection* object

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
      },
      "properties": {
        "prop0": "value0"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
        ]
      },
      "properties": {
        "prop1": 0.0,
        "prop0": "value0"
      }
    }
  ]
}
```

Geospatial operators of MongoDB for spherical data support the GeoJSON [17] format. GeoJSON is a human-readable encoding format for representing the geographical features using JSON standard. It consists of an object which describes a geometry, *Feature*, or collection of *Features*. The type of geometries must be one of these types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. A *Feature* must have members with the names *geometry* and *properties*. A feature collection is an object with the type *FeatureCollection* which must have a member with the name *features*. Code 11 is an instance GeoJSON file with a *FeatureCollection* object.

The version 2.4 of MongoDB, which we have used for our testings, supports only three geometry types Point, LineString, and Polygon[5]. As mentioned in Section 3.3.1, a geospatial query in MongoDB can use *geoIntersects*, *geoWithin*, *near*, and *nearSphere* operators. In addition, there are geometry specifiers to define geometries in query conditions of these operators. For instance, the *$center* and *$centerSphere* are specifiers for a circle area in planar and spherical maps on which the user intends to do the query. Then, the user can find all geometries within a determined circular area. Another example is the *$maxDistance* specifier that determines the maximum distance from a point in order to find geometries within this particular distance. Geometry specifires are limited to, *$geometry*, *$maxDistance*, *$center*, *$centerSphere*, *$box*, and *$polygon*. An example demonstrating the usage of the specifiers in a query is:

```
db.<collection>.find( { <location field> :
                          { $geoWithin :
                            { $centerSphere :
                                [ [ <x>, <y> ] , <radius> ] }
                        } } )
```

In the following sections, we will discuss the query performance of these two databases. In Section 5.1, we analyze specific test queries. Then, the assessment continues utilizing the indexing structure in Section 5.2. Later, the update functionality of databases is investigated in Section 5.3. In Section 5.4 we explain an experimental approach to query a database by MongoDB via BaseX. A discussion about this approach is then introduced in Section 5.5. Finally, Section 5.6 contains the conclusion and further work.

5.1 Querying the Databases

To analyze the behavior of databases, designing the same test cases and evaluating the run-times is the most straight-forward approach. Based on the MongoDB geospatial features and properties, the Netherlands test dataset must be changed to be imported and tested. A brief explanation of the changes is provided here to clarify the rules and limitations. The main change was to either remove multipolygon from the dataset or transforming them to a set of distinct polygons, since multipolygon is not supported in the version of MongoDB that we have used. Besides, the coordinate system is converted to WGS84. At the end, there are 12773 polygons and no multipolygons in the data file. Since the geometries are changed and the results would be different, we repeat all queries with the new dataset in BaseX.

To start with, the GeoJSON file containing the geometries is imported into MongoDB. This can be done using Java API or Mongo Shell with appropriate *mongoimport* command. There are limitations to consider while importing a GeoJSON file in MongoDB. The main one is the *document* size limitation in a *collection*. Document and collection are two concepts in MongoDB that correspond to a table and

[5]The version 2.6 adds support for MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.

CPU	RAM	Hard disk	OS	Bits
Intel Core i3-2310M 2.10GHz	4 GB	80 GB	Ubuntu 12.04.5 LTS	64

Table 4: Hardware architecture used for testing

a record in this table, respectively. A collection can have one or more documents and a document is a set of key-values. A database contains one or more collections. To successfully import a document into a collection, it should be smaller than 16MB. Since it is common to have large geometries in real-world geographic data files, this limitation can be a problem. Another limitation is that the z coordinate is not supported in MongoDB. When a coordinate contains z dimension, although those geometries are not processed, no error message is shown during the import process and query. MultiPoint, MultiLineString, and MultiPolygon, which are not supported in the version 2.4 of MongoDB, are also skipped in queries without any error message.

The above-mentioned size limitation made us change the file structure again. When the file structure was in the form of Code 11, an error message regarding the size limitation was thrown. We changed the GeoJSON file structure to contain individual features as Code 12 in order to solve this problem.

Code 12: A GeoJSON file restructured regarding the size limitation

```
{
  "type": "Feature",
  "geometry": {...},
  "properties": {...}
},
{
  "type": "Feature",
  "geometry": {...},
  "properties": {...}
},
{ ... }, ..., { ... }
```

After importing the file, we examined the common features, i.e., intersection, within functionalities, in both databases. Each query is performed in a client-server architecture in both databases, using scripts to execute it 100 times. The time presented for each query is the average of all total internal times, excluding the first one. The results of queries have been read into the memory and have not been serialized and written out. All queries in this section have been performed with the version 7.8 of BaseX and version 2.4 of MongoDB. The system architecture that have been used are shown in Table 4.

The queries in Code 13 and Code 14 find the intersecting geometries from the Netherlands' dataset with the given polygon in BaseX and MongoDB, respectively. The polygon coordinates are determined for different queries in the following way. At first, the maximum and minimum coordinates are found in the dataset. After that, we generate different input polygons in regard to these boundaries. The polygons coordinate are altered such that the number of results differs in each query.

Code 13: Example query of the *intersects* function in BaseX

```
import module namespace geo = "http://expath.org/ns/Geo";
declare namespace gml="http://www.opengis.net/gml";
```

```
let $a:= <gml:Polygon>
            <gml:outerBoundaryIs>
              <gml:LinearRing>
                <gml:coordinates>
                  6,52.6 6.1,52.6 6.1,53 6,53 6,52.6
                </gml:coordinates>
              </gml:LinearRing>
            </gml:outerBoundaryIs>
          </gml:Polygon>
for $b in //gml:Polygon
return if (geo:intersects( $a, $b)) then $b else ()
```

Code 14: Example query of the *intersects* function in MongoDB

```
db.places.find( { geometry :
                  { $geoIntersects :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates:[[[6,52.6],[6.1,52.6],
                                      [6.1,53],[6,53],[6,52.6]]]
                } } } } )
```

Similar queries are designed to check the other functions. Here, we skip the queries and directly go to the results. Before, we briefly take a look at the MongoDB profiler through the *$explain* operator either in the forms:

```
db.collection.find()._addSpecial( "$explain", 1 )
db.collection.find( { $query: {}, $explain: 1 } )
```

or

```
db.collection.find().explain(),
```

which displays the profile of current query specified in the *find* function. This can also be done for any operation by querying the *system.profile* collection. Below the profile of a sample query is shown,

```
"cursor" : "BasicCursor",
"isMultiKey" : false,
"n" : 2756,
"nscannedObjects" : 12773,
"nscanned" : 12773,
...
"millis" : 43907,
...
"server" : "MongoServer"
```

In the provided information, number of the results and the query time is included in *n* and *millis* elements, respectively. The item *cursor* specifies the type of cursor used by the operation. Here, *BasicCursor* indicates that the query is merely performing a normal scan to find the results. This typical cursor reads the documents in natural order. That is, no index is used for this operation and the whole data is scanned in the original order. If a geospatial index is used, the cursor type will change. The value *n* reflects the number of geometries that match the query condition, i.e., the number of items on the cursor. The item *nscanned* is the number of scanned documents for this operation when no index is used, or scanned index entry in the range when an index is used. The item *nscannedObjects* is the number of scanned documents in this query to obtain the results. Queries in which an index structure is not used, as in the above sample, will have equal value for these two numbers. Otherwise, *nscannedObjects* may be lower, as shown in the query plan of Code 15 which uses a geospatial index. In other words, the following inequalities always hold:

$$nscanned \geq nscannedObjects \geq n.$$

Of course, the most optimal state is where $nscanned = nscannedObjects = n$.

Code 15: Query profile in MongoDB when the related query uses geospatial index

```
...,
"cursor" : "S2Cursor",
"isMultiKey" : true,
"n" : 2756,
"nscannedObjects" : 3370,
"nscanned" : 36941,
...,
```

As expected and therefore not shown here, queries of *intersects* and *within* functions without index, all have query times that are more or less the same in both databases, since the whole data is scanned. This is also obvious from the following query plan in MongoDB, which specifies the cursor type, number of the scanned index keys (*nscanned*), and number of the scanned documents (*nscannedObjects*):

```
...
"n" : 12773,
"nscannedObjects" : 12773,
"nscanned" : 12773,
...
```

Until now, the sample queries are applied to the databases without using any index. In the following section, we measure the running time again by applying the geospatial index. In this way, we also cover *near* and *nearSphere* queries in MongoDB that definitely need the geospatial index.

5.2 Indexing in the Databases

The assessment process is continued by applying the geospatial index. In MongoDB, there are two choices for using geospatial index, *2d* for data expressed in legacy coordinate pairs and *2dsphere* for both GeoJSON data objects and legacy coordinate pairs. We use *2dsphere* which fits the choice of spherical maps via the following shell command,

```
db.Collection.ensureIndex({geometry:"2dsphere"})
```

The same step is also taken in BaseX. Running the same queries again using a geospatial index, supposedly gives declined query times and an upward trend in performance as the number of results go higher in both databases (see Figure 25). As explained in Section 4, index is applied through the *filter* function in BaseX while MongoDB follows another approach. MongoDB applies the index automatically after executing the above-mentioned command and there is no need to change the

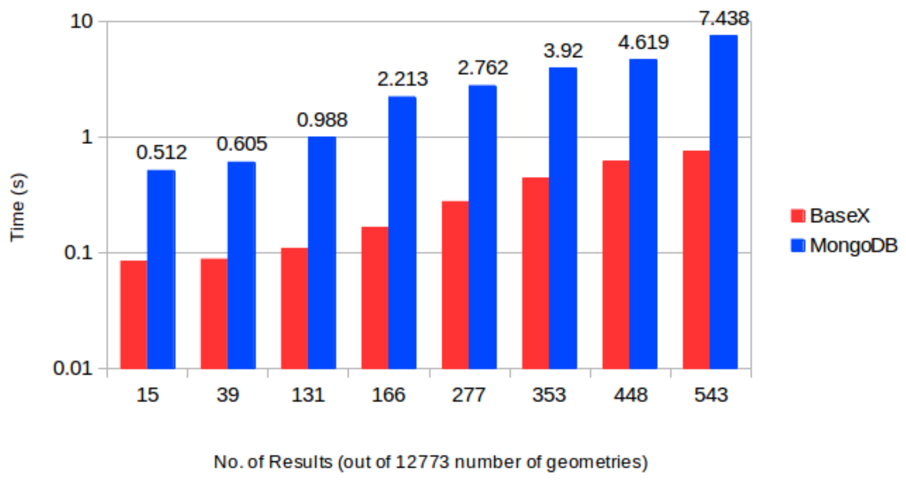


Figure 25: Query time of *intersect* operation with geospatial index

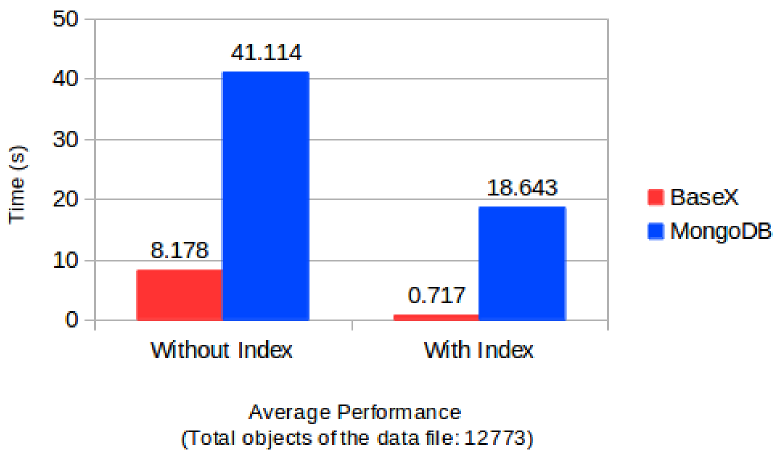
queries in general. Figure 26 illustrates how BaseX and MongoDB perform when running the two main geospatial functions, *within* (Figure 26(a)) and *intersect* (Figure 26(b)). For each function, the average time of eight different queries is shown which are executed with and without index. For the *intersect* function, we calculated the average for the same results shown in Figure 25 and for the *within* function, we have chosen another queries with the range of results from 64 to 1871 numbers. Not surprisingly, using the index structure improves the performance while queries without using the geospatial index performance remains stable. As it can be seen in this figure, in both functions BaseX outperforms MongoDB.

The next widely used type of query is finding the near places or geometries up to a distance from a specific point. It could be also expressed as finding the geometries inside a circle with the distance value as the radius length. This feature is provided in MongoDB via *$near* and *$nearSphere* operators, which simply get the reference point and the distance and hereafter returns the whole geometries in the specified distance from the point. Both near operators need a geospatial index, either *2d* or *2dsphere*. A specific distance in meter is defined as a condition to filter those geometries within this area. In Code 16, we provide a sample query of the *$nearSphere* operator.

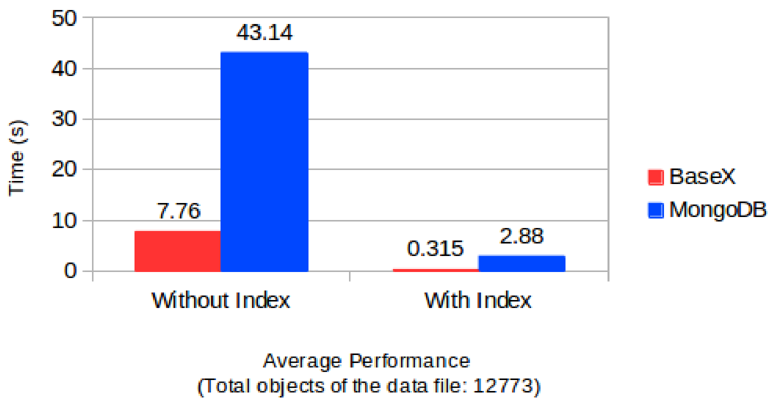
Code 16: Sample query using *$nearSphere* in MongoDB

```
db.collection.find({{geometry:
                        {$nearSphere:
                          {$geometry:
                            {type:"Point", coordinates:[4.5,51.95]},
                              $maxDistance : 100
                     }}}})
```

BaseX Geo Module currently provides this feature via *distance* or combination of *buffer* and *within* functions. In both cases, if the coordinate system in which the data is provided is a projected coordinate system, the distance can be specified in meter or any other metric unit. In a projected coordinate system, all the areas, lengths, and angels are defined on a flat two-dimensional space. In such a system, the user is provided with this *near* feature in BaseX by using the above-mentioned functions as in Code 17.



(a) Average performance of the *within* function



(b) Average performance of the *intersect* function

Figure 26: Performance comparison of the two main geospatial functions *within* and *intersect* in BaseX and MongoDB

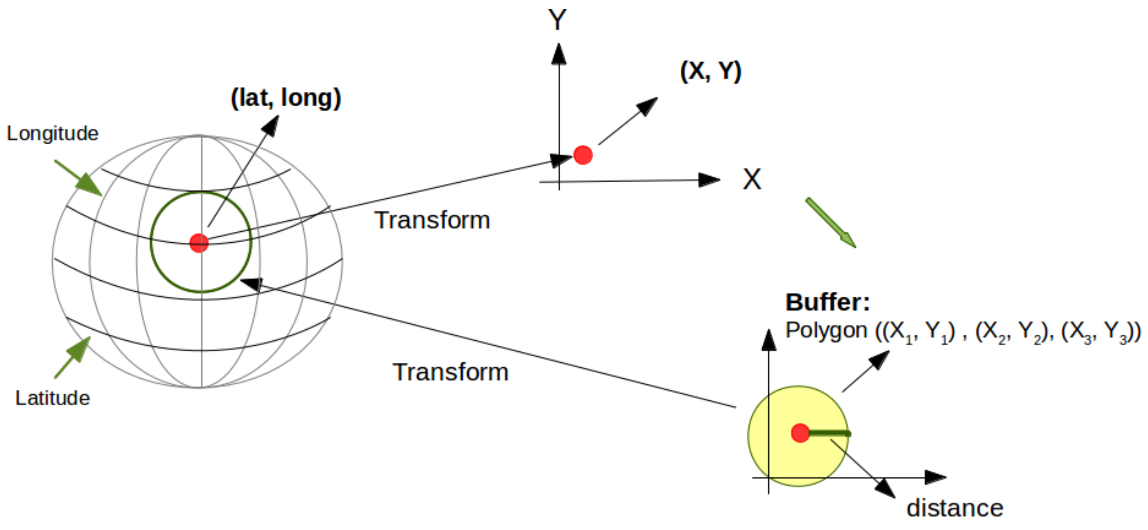


Figure 27: Having the *near* feature for WGS84 data using transformation in BaseX

Code 17: Sample query to the *near* feature in BaseX

```
import module namespace geo = "http://expath.org/ns/Geo";
declare namespace gml="http://www.opengis.net/gml";
let $a:= <gml:Point>
              <gml:coordinates>
                4.5,51.95
               </gml:coordinates>
           </gml:Point>
for $b in //gml:Polygon
return if (geo:distance( $a, $b) le 500) then $b else ()
```

For the data provided in a geographic coordinate system, calculations do not work with the metric values. A geographic coordinate system, such as WGS84, defines the areas and locations in a three-dimensional spherical space. Since there is not a specific unit for the current Geo Module functions, the *distance* function interprets the input distance in the coordinate system of data. For example, the Netherlands sample dataset is in a metric coordinate system. Hence, the distance value in queries can be specified in meters.

The aforementioned *near* feature as a common use case is available in BaseX only for metric coordinate systems. To provide this feature also for geographic systems via the current geo functions, we need to convert the geographic coordinates to metric ones. In this way, we convert the specified input geometry in WGS84 to a metric system. Then, we compute the buffer of converted geometry up to the input distance in the same metric system and convert the buffer back to the original geographic coordinate system. A buffer of geometry is an identified region within a specific distance of the geometry. At the end, we find the all geometries within the buffer in geographic coordinate system. This approach benefits from the geospatial index structure via the *within* function while the approach with the *distance* function do not utilize the geospatial index. This process is illustrated in Figure 27. This procedure is expressed as below, using the *buffer* and *within* functions and should return the same result as the previous query,

```
within(geometry, transform(buffer(
        transform(myPoint_in_wgs84, other_metric_cs), 500),wgs84))
```

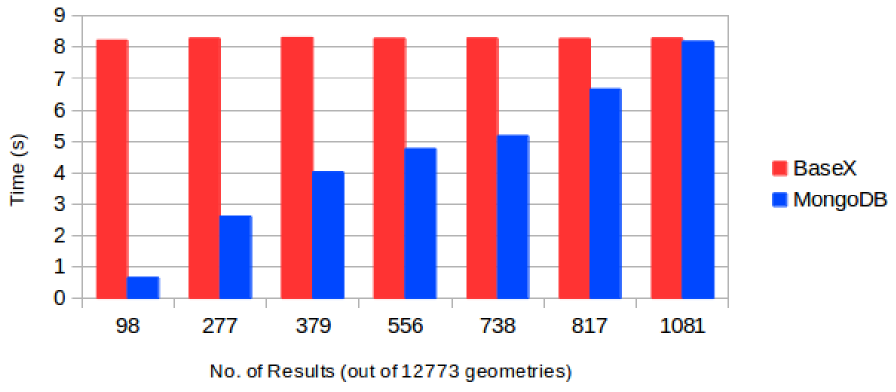


Figure 28: Query time of the *near* feature

Having the same use cases, we have tested this feature in both databases, with the run-times shown in Figure 28. BaseX uses the distance function while *$nearSphere* operator finds the geometries in MongoDB. MongoDB outperforms BaseX with smaller number of results. The query times and the number of results are directly proportional in MongoDB, since the geospatial index is used and both changes accompany each other. BaseX does not use the geospatial index and consequently returns every result more or less in the same time. In the real-world applications, finding the nearby objects or places in closer surrounding areas are highly applicable and demanded. Therefore, having this functionality in a better performance is advantageous.

As mentioned in Section 4.2, BaseX records the index structure in a file and reads it back into main memory when the geospatial index is required. A point that we should contemplate is to include the time for index opening in query times, since in memory management the index file might be removed from the memory and read back again. This time is not counted in the presented charts.

5.3 Database Update

Updating a database as a significant interaction supplied by the database system generally comprises three different operations: insert, delete, and modify. In addition to the whole issues related to the database update on disk and memory management, reconstructing the indexes is of great importance. That is, since the index is constructed based on the old database state, the updated database needs a new index.

Currently in BaseX, updating is not applied to the geospatial index after the database update operations. It means that the STR-tree does not get updated automatically and the geospatial index has to be rebuilt manually, which causes redundant constructions.

Considering the manual update of the geospatial index in BaseX, the time of index creation plus reopening the index file into the main memory are overhead which take a huge amount of total time (see Figure 29). Including these times in evaluations, not surprisingly drops the performance of update operations in BaseX compared to MongoDB. However, each database follows a different approach. One provides a static index structure optimized for the data with no or very few updates and the other provides a dynamic index structure. Besides implementing a dynamic geospatial index structure, accessing MongoDB geospatial features and index struc-

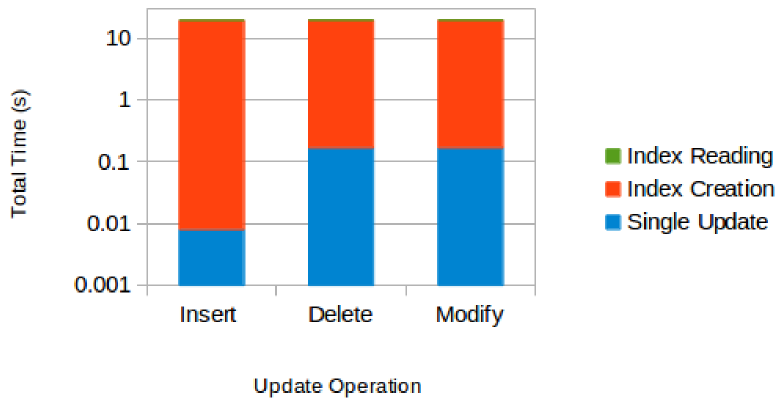


Figure 29: Time consumption proportion in a single update operations in BaseX

ture through BaseX can be an alternative approach. Additionally, we can benefit the *near* feature of MongoDB for small number of results through this way, since this feature is frequently requested to find a few entities around a particular point. In the following section, we discuss this method.

5.4 Querying by MongoDB via BaseX

Up to this point, we have investigated both BaseX and MongoDB to gain some improvement ideas for BaseX. We go further in this way by querying the data with MongoDB through BaseX. A conceivable goal for this approach is supplying the missing features or make the features with better performance in MongoDB accessible from BaseX. We test this approach by connecting to MongoDB via BaseX. Then, the provided queries in MongoDB syntax are executed by MongoDB and the results are shown through BaseX. For the missing features, this would add a functionality to BaseX. But, for the common queries we should evaluate the new query times to see how the performance changes. The test is implemented by sending the query to the MongoDB server and receiving the results in a test function that call some other private functions. The test function can be called in XQuery, defining the arbitrary arguments. For instance, for the *near* feature the center point and a distance value must be specified. Since MongoDB gives the result in GeoJSON, they must be converted to XML in BaseX. To test the *near* feature, we run queries with *$nearSphere* operator. As mentioned the closer distances and consequently smaller number of surrounding objects are of great interest. Searching for the five hundred restaurants around a house rarely happens, while finding the ten nearest ones is more requested. The test performance depicts that this approach provides better performance of this feature than the way BaseX does (see Figure 30). The constant performance trend in BaseX is due to the absence of geospatial index. Hence, the user can benefit from this functionality in MongoDB through BaseX.

This process is done for the update operations, such that the update query modifies the database in MongoDB. Update operations do not return any objects and there is no need to do any conversion from JSON to XML. Considering the geospatial index update, update queries run-time indicate improvements in performance. It means that the performance improvement is significant in regards to the fact that MongoDB updates the index structure. Correspondingly, it should be decided to continue the querying either in MongoDB or in BaseX after repeating the update queries and rebuilding the index structure.

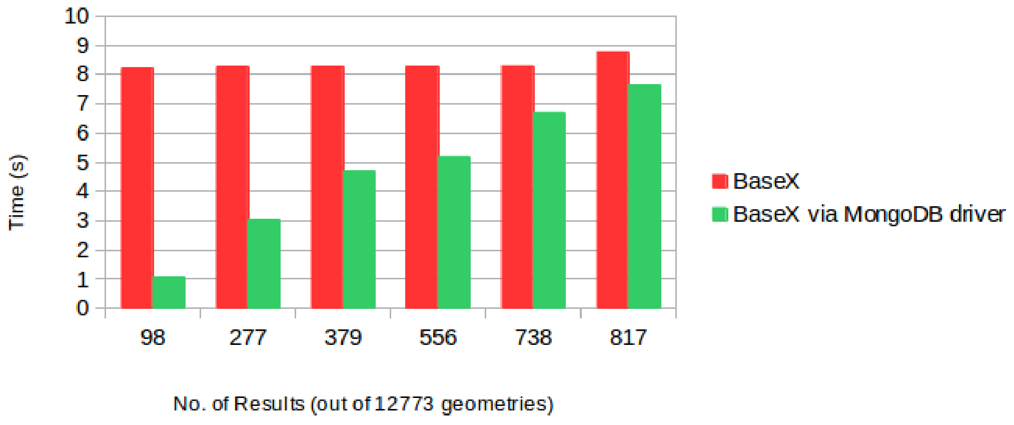


Figure 30: Performance comparison of the *near* feature in BaseX vs. the MongoDB driver in BaseX

5.5 Discussion

Throughout this section, we investigated two strategies to efficiently process the geospatial data in BaseX. Based on these investigations, here we will discuss the previously-mentioned approaches of the geospatial processing in BaseX and cover the pros and cons of the available approaches, particularly the most recent one in Section 5.4.

To start with, we suppose that a user has a geospatial dataset in GML and they would like to work in this format in BaseX, since their general requirements are met by BaseX. Based on our examinations, there are some queries in which BaseX is not able to cover them or cannot provide some features in a proper time. It means, those cases either are not covered, like updating the index structure or are covered with an apparently poor performance in comparison with the other similar query times, such as the *near* feature. Therefore, geospatial queries such as *intersects* or *within*, can be applied efficiently while some other like *near* and update, cannot. Furthermore, regarding to our assumption, some functionality of BaseX convinces the user to work with GML. We call them *GML-related* features. We consider these co-called *GML-related* features in our discussion, as they might influence the geospatial processing approach that will be preferred.

Table 5 summarizes the statement above. *MongoDB through BaseX* in this table means that BaseX connects to MongoDB to benefit by the features which are not available or available with poor performance in BaseX, as discussed in Section 5.4. *GML-related* features points the aforementioned features which are supposed here to be satisfying enough for the user, indicated with the notation "+". This assumption is made in order to concentrate on the evaluation of those cases in which the user choose to use BaseX and we try to cover the missing geospatial features. With "++", we mean that option is very efficient.

As is expressed in Table 5, the geospatial queries like *intersects* and *within* are working with an acceptable performance in BaseX. On the other hand, for the *near* and update queries connecting to MongoDB seems to be the appropriate approach. Hence, a fast conclusion would be to have the advantage of using MongoDB through BaseX. However, the synchronization of the database instance in BaseX and the database instance in MongoDB should be precisely examined. If the user intends to modify the database, which also means the modification of index without any further querying regarding the changes, the faster strategy is to update the MongoDB

Tasks	**BaseX**	**MongoDB through BaseX**
GML-related Queries	+	
Geospatial Queries, like *intersects/within*	+	
The *near* Query		++
DB Update Queries		++

Table 5: The comparison of two strategies; using BaseX or connect to MongoDB via BaseX

Scenarios	**What to do?**
Needs a *intersects/within* query and no update and no *near* query	Using BaseX only
Needs a *near* query and no update	Using BaseX and MongoDB together
Needs modifications in the database	Using MongoDB only

Table 6: Possible scenarios and appropriate actions suggested to take

instance, while the database instance in BaseX remains untouched. But, in case that the user wants to do further querying based on the changes, this approach is not optimized.

One way to discard the above-mentioned problem could be limiting the user to do all geospatial queries regarding the index in MongoDB and use BaseX just as a connector. However, as we see in Table 5, doing some queries in BaseX is more efficient and connecting to MongoDB might be a disadvantage, since this adds at least the connection time to the query time. Moreover, the complete dependency of BaseX on MongoDB for all geospatial queries would be too much and disadvantageous, while each system is designed for different requirements and follow various goals. Thus, it will be more appropriate to provide the user with both possibilities.

Table 6 summarizes the sample scenarios that could happen based on the existing features to process the geospatial data. In addition, this table shows the relevant and appropriate actions the user can apply as an efficient strategy toward her/his goal.

The first scenario in Table 6 is the use case in which the BaseX *Geo Module* functions and index structure fulfills all the requirements and there is no need for further queries, while the database remains unchanged. In this case, everything can be done by BaseX. In the second scenario of this table, the user needs the *near* feature as a missing one, based on the untouched state of the database. For this case, there are instances of both BaseX and MongoDB and the database is created in both instances. Then, the user can connect to MongoDB in order to get the result of the *near* query easier and faster. It should be mentioned that the issues related to the coordinate systems must be managed before applying the *near* query. The last scenario mentioned in the table is the case in which the database will be updated by MongoDB and the next queries are based on the new state of the database. In this case, which BaseX does not provide a straightforward way, accessing MongoDB update facilities and applying the subsequent queries in MongoDB hereafter is the advantageous approach.

5.6 Conclusion

In this section, we mainly concentrated on the geospatial features of MongoDB in order to find ways to improve the BaseX Geo Module efficiency. We started with the investigation of geospatial features to see the different viewpoints in MongoDB. Based on the common features in both databases, the performance was represented and explained with and without geospatial index, to see the influence of geospatial index. Among the queries, finding the geometries within a specific distance, called the *near* feature, is of a great importance, as a widely-used feature. MongoDB provides this feature in both WGS84 for data on the earth as well as legacy coordinate pairs, while BaseX supports only for the projected coordinate systems, since geospatial functions in BaseX do spatial calculations on the flat space. To add the support of data in WGS84 in BaseX, new functions should be implemented. Testing this feature in both databases shows a better performance in MongoDB.

Besides, updating in both databases is discussed. Update in MongoDB implicitly updates the index structure, while the geospatial index in BaseX must be reconstructed. Therefore, query time severely increases considering the time consumed by index updating.

Since the performance of two functionalities, i.e., the *near* feature and updating, are not satisfying in BaseX and MongoDB provides a faster way, we connect to MongoDB and run these queries through BaseX. The results of the *near* feature by this approach seem convincing enough to have this alternative way of querying in BaseX. In addition, updating the database by MongoDB via BaseX is faster than updating in BaseX. However, using this approach is controversial. The discussion is in regards to the subsequent queries after the database modification. Indeed, the changes happen in MongoDB and the database in BaseX remains untouched. Therefore, further querying based on the new changes is possible merely in MongoDB. In case that the next geospatial queries need to be done in BaseX, this way will not worth trying, because the whole update and index reconstruction process must be repeated in BaseX. The approach sets the limitation for the use cases in which the database will be updated, to do the geospatial querying completely in MongoDB.

6 Future Work

As a result of this work, BaseX is now capable of processing geospatial data. However, there is still room for future improvements. In particular, the storage and indexing are at the center of attention. Right now, the whole indexing tree is stored in the main memory which results in poor I/O efficiency. One of the first ideas is to bring some parts of the tree, which are addressed repeatedly, into RAM. Regarding the indexing, an interesting future work is to practically compare the current STR-tree implementation with other indexing trees in the existence of different kinds of geospatial data. For example, a promising research would be the comparison between STR-tree and Quad-tree having dynamic datasets. Moreover, the support of higher versions of GML and other geospatial data formats alike, could be a topic to work on.

In this thesis, we have considered MongoDB as an alternative for querying and indexing. A feasible improvement would be the implementation of a flexible interface for this database. As we discussed in Section 5.6, there would be an inconsistency between the geospatial data and indexing in BaseX and the simultaneously indexed data in MongoDB. A future improvement is to implement a module to give the user the whole control over these two possibilities. Besides, we can follow this idea by connection to the other well-known databases.

7 Summary

In this thesis, we discussed several topics related to geospatial data processing. As main focus, we examined issues related to geospatial features in the native XML database BaseX.

Knowing the geospatial properties and definitions, the first topic that we covered as the most important one was the geospatial indexing structure. We chose some major indexing structures and their variations, which are studied broadly in the literature. Numerous structures are introduced later based on the few ones, such as KD-tree or R-tree. We explained each algorithm together with its pros and cons by collecting information from various sources. This explanation provides a perspective of the different indexing structures for future developments in BaseX. As stated in Section 2.3, each index structure is developed to meet a particular requirement. For example, dynamic variants of the R-tree are appropriate for the data which changes frequently. Consequently, a suitable index structure should be chosen based on the system properties and features. However there are numerous publications on this topic, choosing and implementing an index structure is not straightforward. Since the R-tree is a widely used structure, we chose an advanced static variant of the R-tree, STR-tree, to implement a experimental index structure in BaseX. The implementation that is integrated into BaseX is from a well-known open-source API, JTS.

Another topic we discussed in this thesis is the related areas to focus when providing geospatial features. Storage and querying are the main topics and we introduced some proposed or implemented ideas. Besides, we explained the geospatial features in some database systems to show how the implemented ideas work.

The Geo Module implementation in BaseX as our focus starts with geospatial functions and proceeds with adding the STR-tree as the index structure. The evaluations demonstrate how the index implementation and other optimizations helped us to reach a better performance.

While studying geospatial features of MongoDB, we found that in some cases BaseX outperforms MongoDB and vice-versa. Hence, we ran the queries in which MongoDB provides better performance through a driver. We represented the evaluations that are convincing to follow this approach as a parallel way. As we discussed in Section 6, the BaseX Geo Module can be expanded by adding the support of other geospatial data formats and an index structure for dynamic data like the Quad-tree. Additionally, the idea of connecting to MongoDB can be integrated into BaseX to benefit from the specific index structure and features for GeoJSON.

References

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] Jesús Almendros-Jimenez, Antonio Becerra-Terón, and Francisco García-García. XPath for Querying GML-based Representation of Urban Maps. In Beniamino Murgante, Osvaldo Gervasi, Andrés Iglesias, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, volume 6782 of *Lecture Notes in Computer Science*, pages 177–191. Springer Berlin Heidelberg, 2011.

[3] Jesús Almendros-Jimenez, Antonio Becerra-Terón, and Francisco García-García. Development of a Query Language for GML based on XPath. In Laura Kovacs and Temur Kutsia, editors, *WWV 2010*, volume 18 of *EPiC Series*, pages 51–64. EasyChair, 2013.

[4] Jay Banerjee and Won Kim. Supporting VLSI Geometry Operations in a Database System. In *Proceedings of the Second International Conference on Data Engineering*, pages 409–415, Washington, DC, USA, 1986. IEEE Computer Society.

[5] Kyle Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[6] R. Bayer and Edward McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[8] David Beddoe, Paul Cotton, Robert Uleman, Sandra Johnson, and John R. Herring. OpenGIS Simple Features Specification For SQL. Technical report, OGC, May 1999.

[9] Alberto Belussi, Omar Boucelma, Barbara Catania, Yassine Lassoued, and Paola Podestà. Towards similarity-based topological query languages. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 675–686. Springer Berlin Heidelberg, 2006.

[10] Jon L. Bentley. Multidimensional Binary Search Trees in Database Applications. *Software Engineering, IEEE Transactions on*, SE-5(4):333–340, July 1979.

[11] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[12] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. pages 28–39, 1996.

[13] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 479–490, New York, NY, USA, 2006. ACM.

[14] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, et al. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml, November 2008.

[15] Leonard Brown and Le Gruenwald. Tree-Based Indexes for Image Data. *Journal of Visual Communication and Image Representation, Volume 9, Number*, 4:300–313, 1998.

[16] Greg Buehler and Carl Reed. Open Geospatial Consortium (OGC) Orientation Slides, 2011.

[17] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Tim Schaub, and Christopher Schmidt. The GeoJSON Format Specification. http://geojson.org/geojson-spec.html, June 2008.

[18] Jianhua Chen, Binbin He, and Weihong Wang. GXQuery: A GML spatial data query language. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 271–275, April 2010.

[19] Beng Chin, Ooi Ron Sacks-davis, and Jiawei Han. Indexing in spatial databases, 1993.

[20] GeoTools Project Management Committee. GeoTools, The Open Source Java GIS Toolkit. http://geotools.org/, 2012.

[21] Jose E. Córcoles, Pascual González, and Victor López Jaquero. Integration of Spatial XML Documents with RDF. In Juan Manuel Cueva Lovelle, Bernardo Martín González Rodríguez, Jose Emilio Labra Gayo, María Puerto Paule Ruiz, and Luis Joyanes Aguilar, editors, *Web Engineering*, volume 2722 of *Lecture Notes in Computer Science*, pages 407–410. Springer Berlin Heidelberg, 2003.

[22] MarkLogic Corporation. Search developers guide, November 2014.

[23] Simon Cox. *Geography Markup Language (GML)*, pages 195–196. SAGE Publications, Inc., 0 edition, 2008.

[24] Martin Davis. Secrets of the JTS Topology Suite. In *Proceedings of the annual Free and Open Source Software for Geospatial (FOSS4G) conferencea*, 2007.

[25] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, pages 1–9, 1974.

[26] Christian Grün. BaseX – The XML Database for Processing, Querying and Visualizing large XML data. http://basex.org, October 2010.

[27] Oliver Günther. *Efficient Structures for Geometric Data Management*, volume 337 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin-Heidelberg-New York, 1988.

[28] Oliver Günther. *Efficient Structures for Geometric Data Management*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[29] Mariella Gutiérrez and Andrea Rodríguez. Querying heterogeneous spatial databases: Combining an ontology with similarity functions. In Shan Wang, Katsumi Tanaka, Shuigeng Zhou, Tok-Wang Ling, Jihong Guan, Dong-qing Yang, Fabio Grandi, Eleni E. Mangina, Il-Yeol Song, and Heinrich C. Mayr, editors, *Conceptual Modeling for Advanced Application Domains*, volume 3289 of *Lecture Notes in Computer Science*, pages 160–171. Springer Berlin Heidelberg, 2004.

[30] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[31] William E. Huxhold. *An Introduction to Urban Geographic Information Systems*. Number 9780195065350 in OUP Catalogue. Oxford University Press, October 1991.

[32] Apple Inc. Location and Maps Programming Guide: Displaying Maps, March 2014.

[33] Open Geospatial Consortium Inc. *OpenGIS Implementation Standard for Geographic information - Simple feature access-Part 1: Common architecture*. 1.2.1 edition, 2011.

[34] Hosagrahar V. Jagadish. On Indexing Line Segments. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 614–625, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[35] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[36] Gershon Kedem. The Quad-CIF Tree: A Data Structure for Hierarchical On-Line Algorithms. In *Design Automation, 1982. 19th Conference on*, pages 352–357, June 1982.

[37] Sam Kleinman. MongoDB – 2D Index Internals, February 2014.

[38] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[39] Wolfgang Kresse and David M. Danko. *Springer Handbook of Geographic Information*. Springer Handbook of Geographic Information. Springer Verlag, 2011.

[40] Taewon Lee and Sukho Lee. OMT: Overlap Minimizing Top-Down Bulk Loading Algorithm for R-tree. In Johann Eder and Tatjana Welzer, editors, *CAiSE Short Paper Proceedings*, volume 74 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[41] Scott Leutenegger, Mario A. Lopez, and J. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. pages 497–506, 1997.

[42] Xia (Lisa) Li. XQuery as a Spatial Query Language. In *Proceedings of the 2006 International Conference on Internet Computing and Conference on Computer Games Development, ICOMP*. USA, January 2006.

[43] Yuzhen Li, Jun Li, and Shuigeng Zhou. GML Storage: A Spatial Database Approach. In *Conceptual Modeling for Advanced Application Domains*, volume 3289 of *Lecture Notes in Computer Science*, pages 55–66. Springer Berlin Heidelberg, 2004.

[44] David B. Lomet and Betty Salzberg. The HB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, December 1990.

[45] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[46] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[47] Takashi Matsuyama, Le Viet Hao, and Makoto Nagao. A file organization for geographic information systems based on spatial proximity. pages 303–318, 1984.

[48] Andreas Neumann and Marco Hugentobler. Open-Source GIS Libraries. In *Encyclopedia of GIS*, pages 816–820. Springer US, 2008.

[49] Ooi and Beng Chin. The Spatial KD-tree. In *Efficient Query Processing in Geographic Information Systems*, volume 471 of *Lecture Notes in Computer Science*, pages 80–115. Springer Berlin Heidelberg, 1990.

[50] Ooi, K. J. Mcdonell, and Sacks R. Davis. Spatial KD-tree: An Indexing Mechanism for Spatial Database. *COMPSAC conf.*, pages 433–438, 1987.

[51] Open Geospatial Consortium. OGC KML Standard - Version 2.2. http://www.opengeospatial.org/standards/kml.

[52] B. Padmaja, R. Sateesh, and K. Dhanasree. Shortest Path Finding Using Spatial Ranking. *International Journal Of Computational Engineering (IJCER)*, 2(5):1186–1189, September 2012.

[53] Apostolos N. Papadopoulos. *Nearest Neighbor Search: A Database Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[54] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing.* Apress, Berkely, CA, USA, 1st edition, 2010.

[55] Gavin Powell. *Beginning XML Databases.* Programmer to programmer. Wiley, 2007.

[56] John T. Robinson. The KDB-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18. ACM, 1981.

[57] J. Andrew Rogers. What are the limitations of geohashing?, February 2011.

[58] Nick Roussopoulos and D. Leifker. An introduction to PSQL: A pictorial structured query language. In *Proc. IEEE Workshop on Visual Languages*, pages 77 – 87, 1984.

[59] Nick Roussopoulos and Daniel Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 17–31. ACM Press, 1985.

[60] Peter Rusforth. Geo module: A Geospatial API. http://expath.org/spec/geo, September 2010.

[61] Neal Sample, Matthew Haines, Mark Arnold, and Timothy Purcell. Optimizing Search Strategies in KD-trees. In *5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001)*, July 2001.

[62] Peter Schiess. Projections and Coordinate Systems, September 2010.

[63] Thomas Schwarz, Nicola Hönle, Matthias Großmann, and Daniela Nicklas. A library for managing spatial context using arbitrary coordinate systems. In *PerCom Workshops*, pages 48–54. IEEE Computer Society, 2004.

[64] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[65] Masoumeh Seydi. Geo Module. http://docs.basex.org/wiki/Geo_Module, June 2013.

[66] Shashi Shekhar and Hui Xiong. Java Topology Suite (JTS). In Shashi Shekhar and Hui Xiong, editors, *Encyclopedia of GIS*, pages 601–601. Springer US, 2008.

[67] Shashi Shekhar and Hui Xiong. MX-Quadtree. In *Encyclopedia of GIS*, pages 765–765. Springer US, 2008.

[68] Erik Siegel and Adam Retter. *eXist: A NoSQL Document Database and Application Platform.* O'Reilly, January 2015.

[69] Tobias Trelle. Spring Data - Part 4: Geopatial Queries with MongoDB, February 2012.

[70] Maarten Vermeij, Wilko Quak, Martin Kersten, and Niels Nes. MonetDB, A Novel Spatial Column-Store DBMS. In *Proceedings of the academic track of the 2008 Free and Open Source Software for Geospatial (FOSS4G) Conference, incorporating the GISSA 2008 Conference*, pages 193–199, 2008.

[71] Wei Wang, Jiong Yang, and Richard Muntz. PK-tree: A Spatial Index Structure for High Dimensional Point Data. In *Proc. 5th Int. Conf. on Foundation of Data Organizations (FODO*, pages 27–36, 1998.

[72] Qingting Wei. A query-friendly compression for gml documents. In *Proceedings of the 16th International Conference on Database Systems for Advanced Applications*, DASFAA'11, pages 77–88, Berlin, Heidelberg, 2011. Springer-Verlag.

[73] Shulian Zhang, Jiayan Gan, Jiehui Xu, and Guonian Lv. Study on Native XML Database Based GML Storage Model. In *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences.*, volume XXXVII of *Part B4*. Beijing, 2008.

[74] Tian Zhao, Chuanrong Zhang, Mingzhen Wei, and Zhong-Ren Peng. Ontology-based geospatial data query and integration. In *Proceedings of the 5th International Conference on Geographic Information Science*, GIScience '08, pages 370–392, Berlin, Heidelberg, 2008. Springer-Verlag.

[75] Fubao Zhu, Hui Chen, and Jihong Guan. Querying GML documents: An XQuery based approach. In *Computer and Communication Technologies in Agriculture Engineering (CCTAE), 2010 International Conference On*, volume 1, pages 393–396, June 2010.

[76] Fubao Zhu, Qianqian Guo, and Jinmei Yang. Storing GML Documents: A Model-Mapping Based Approach. In Jianliang Xu, Ge Yu, Shuigeng Zhou, and Rainer Unland, editors, *Database Systems for Adanced Applications*, volume 6637 of *Lecture Notes in Computer Science*, pages 89–100. Springer Berlin Heidelberg, 2011.

List of Figures

List of Tables

Listings