AN INDEXING METHOD FOR SPATIAL DATABASES

Kemal Akkaya and Adnan Yazici
Department of Computer Engineering
Middle East Technical University, 06531, Ankara/TURKEY
e-mail: {akkaya,yazici}@ceng.metu.edu.tr

Abstract
In this paper, we introduce an indexing method for accessing spatial databases. The index structure described here is multi-dimensional and is an extension of Multilevel Grid File (MLGF) combined with the z-ordering technique to efficiently handle indexing on the spatial components of the objects. The other important property of the proposed index structure is to be able to index on fuzzy information and process fuzzy querying in spatial databases. Handling spatial, aspatial data and fuzzy information in the physical database is necessary to satisfy some of the requirements of the spatial database applications, i.e., the geographic information systems (GIS) applications. With our proposed multi-dimensional index structure (we call it ExMLGF in this paper), one can create an index structure on aspatial (and fuzzy) data along with spatial data on the same index structure and process aspatial, spatial queries and fuzzy/crisp queries efficiently for the spatial database applications. The ExMLGF access structure is designed and implemented in a way that database users can have fuzzy queries on both homogenous and heterogeneous domains. In this paper we include a number of algorithms for processing different kinds of queries in spatial databases.

Keywords: Spatial Databases, Multi-dimensional Access Structures, Z-Ordering, Fuzziness and Uncertainty.

1. Introduction
Spatial data are a type of data with a spatial component that allows them to be located on, above, or below some base, such as Earth's surface and on, in, or around a structure. Spatial data exist in various application areas such as geographic information systems (GIS), image databases, mechanical CAD system, and so on. The effectiveness of these systems, (e.g., GISs) for decision support could be severely compromised if the spatial data are not properly managed in databases. Therefore, there has been considerable amount of work done in spatial data management in recent years. Since spatial data usually are complex and have a number of unique requirements (such as spatial components and fuzzy properties) the current access structures cannot manage this sort of complex and uncertain data existing in spatial database applications. For example, searching in spatial databases is not as easy as in conventional databases. Most of the queries of spatial databases often require much more complex and extra work, compared to those of the conventional ones.

In literature a number of multi-dimensional access methods have been proposed by various researchers in order to support spatial search operations in databases ([3]). Different methods have been used to store and retrieve extended and complex objects. Some of these methods are: (1) multiple layers, (2) overlapping regions, (3) clipping, (4) transformation(object mapping). The R-tree [6] and R*-tree [2] were proposed by using the overlapping regions method. R+-tree [9] uses clipping
method. One of the methods in transformation techniques is space filling curves. Z-Ordering [7], namely Peano curves, are widely used in commercial databases. The spatial objects are represented by a list of peano curves. The curves look like Z character, so they are also called z-codes. Z-Ordering gives chance to store extended objects in conventional access structures such as B-trees. Since none of the existing multidimensional access structures supports fuzzy queries in spatial databases and most of them process spatial and aspatial data separately, the existing index structures have some shortcomings in efficiently handling physical data representation and querying issues in spatial databases. In order to overcome these shortcomings, here we adapt MLGF (Multi-level Grid File) [10], [11] for spatial databases. MLGF is a multidimensional access structure that has been extended so that the user can make fuzzy queries in both homogeneous and heterogeneous domains [11], [12]. Considering the requirements of GIS applications, it is very important to contain fuzzy information on the index structure. Therefore, we combine z-ordering technique and MLGF so that the user can make spatial queries in addition to fuzzy aspatial and spatial queries.

Next section summarizes the properties of the Multilevel Grid File access structure. Section 3 gives detailed information about Z-Ordering. After that in Section 4 we explain our work of combining MLGF and z-ordering with an example. Section 5 explains the algorithms using the ExMLGF structure for some spatial and aspatial queries. Finally, the paper ends with the conclusion and future work.

2. Multilevel Grid File (MLGF) Index Structure
Multilevel Grid File (MLGF)[10] is an access structure to overcome the shortcomings of Grid File. This structure prevents the problems encountered in Grid File without loosing the multi-attribute search capabilities of Grid File. MLGF handles merge and split operations on directories. The grid directory is maintained as a multi-level structure where each directory entry points to a lower level directory block. In MLGF, splitting and merging a directory is performed locally, thereby decreasing the amount of I/O that is required for a global split or merge. This cause the structure to be flexible during record insertion and deletion operations. Besides, empty directory entries do not exist in MLGF; therefore the directory size is more compact compared to that of Grid File. The organizing attributes are turned into bit patterns. Bit patterns of each organizing attribute are merged to form a key bit pattern. In MLGF a directory entry is formed of a region vector and pointer. The region vector is a composite bit pattern that is composed of hashed bit patterns of the organizing attributes.

3. Z-Ordering
Space filling curves can be used to represent the extended objects (spatial objects). The complex object is divided into smaller cells and the union of these grid cells approximates the object. There are different kinds of space filling curves such as Hilbert, Gray and Peano (z-order). We utilize z-ordering technique (Orenstein and Merett 1984) in this study. This technique can be summarized as follows: Starting from the fixed universe containing the data object, space is split recursively into halves by d-1 dimensional hyperplanes. The directions of hyperplanes alternate cyclically. This process is done until at least one of the following three conditions hold: (1) The current subspace does not overlap the data object. (2) The current subspace is fully enclosed in the data object. (3) Some given level of accuracy has been reached ([5]).
Z-Ordering is a mapping from two or higher dimensional space into one dimensional space such that any two object that are spatially close in the higher dimensional space are also close to each other in the one dimensional sorted sequence. A simple example is shown in Figure 3.1 on the object numbered by 5. In this example we have a two-dimensional universe and we begin splitting by a vertical line. The splitting operation alternates by vertical and horizontal lines. The regions on the left of a vertical line have the first bit as 0 and the regions on the right of a vertical line have the first bit as 1. Same logic is valid for the horizontal lines. That is, the regions on the up of the line have the second bit 1 and the regions on the bottom of a horizontal line have the second bit as 0. The z-codes of the regions are found by bit interleaving of x and y coordinates. For instance the z-code of region pointed by the arrow in figure 3.1 is:

Shuffle 0110 and 0110 ==> 00 11 11 00

All the regions approximating the object is found by this method and a list of z-codes is obtained for representing the complex object. And finally these z-codes can be stored to an index structure for efficient accessing.

4 Integration of MLGF and Z-Ordering

MLGF is a multi-dimensional structure. We use this property in order to create an index which supports both spatial and non-spatial data access. MLGF is extended in [11] to process fuzzy queries. By further extending this structure, we could handle fuzzy aspatial queries as well as spatial queries.

4.1 MLGF as a Fuzzy Index Structure

GISs usually involve in fuzzy and complex information. For example consider the population of a city. In this example, we may not know the exact population of a city and specify it as a fuzzy value like "crowded". On the other hand, most of the time the
population of a city is known beforehand and stored as a crisp value like 1 million. Therefore it is necessary to handle both crisp and fuzzy data, thus heterogeneous domain. MLGF is adapted as a fuzzy index in [11] and fuzzy queries such as "retrieve the cities which are very crowded" can be handled with a heterogeneous domain. This is done by using special record structures for representing fuzzy and crisp data. The possible values for the fuzzy terms are written and divided into groups. After that, for each value membership function is calculated. The group of values, the membership function and a bit representing whether the value is crisp or fuzzy are stored in a record. Those records are indexed by using the MLGF structure and then used during fuzzy querying. The details of using MLGF as a fuzzy index structure are given in [11].

4.2 Z-Ordering in MLGF

The Extended MLGF structure (ExMLGF) uses the z-ordering technique to access the spatial data. Non-spatial data is included as one of the other dimensions in the ExMLGF structure. In this structure each z-region is treated separately and stored into the ExMLGF structure with their object ids. The id number identifies the object that they belong to. Here we follow a z-value centered view rather than an object centered view, since this view is more efficient than the object centered view. The properties of z-value centered view are; (1) all z-regions for an object are stored into ExMLGF, (2) the number of entries is very big, (3) spatial search is easy. We use a z-value centered view to store the cities with their populations into ExMLGF. The structure of ExMLGF used in our implementation is shown in Figure 4.1. Each record is stored to the struct data record and a directory block is an array of directory entry structs. Our root of ExMLGF is a pointer to a directory block. Directory entries store region vectors and a link to the next level. The is_leaf entry represents whether that is a leaf node or not.

In Figure 3.1 we have 5 cities with the following populations (in thousands). Trabzon(184), Giresun(90), Rize(124), Gumushane(60-small) and Erzurum(420-large).

We divide the population values into three groups: small (50-150), medium (150-400) and large(400-500). The membership function for the fuzzy term short is as follows:

\[
\text{small}(x) = \begin{cases} 
    1 + \left( \frac{c}{(x-\alpha)} \right)^2 & (55 \leq x < \alpha) \\
    1 & (x < 55) \\
    0 & (x \geq \alpha) 
\end{cases}
\]

where \( c = 52 \) and \( \alpha = 160 \).

![Figure 4.1: Logical Structure of ExMLGF](image)
medium(x) and large(x) (their membership functions are not included here) can also be calculated in the same way. Using the membership functions for these fuzzy terms, we can obtain the following membership values and records for each population.

<table>
<thead>
<tr>
<th>Location</th>
<th>Membership Function</th>
<th>Membership Value</th>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giresun</td>
<td>small(90)</td>
<td>0.64</td>
<td>000100010101101101</td>
</tr>
<tr>
<td>Trabzon</td>
<td>medium(184)</td>
<td>0.17</td>
<td>01000100101111110</td>
</tr>
<tr>
<td>Rize</td>
<td>small(124)</td>
<td>0.32</td>
<td>0010000111011110</td>
</tr>
<tr>
<td>Gumushane</td>
<td>small(60)</td>
<td>0.81</td>
<td>0000000111100010</td>
</tr>
<tr>
<td>Erzurum</td>
<td>large(420)</td>
<td>0.20</td>
<td>1001000010000000</td>
</tr>
</tbody>
</table>

Table 4.1 shows all the records to be inserted into ExMLGF. We inserted these values (population and z-codes) into ExMLGF having 2 (two) dimensions. Final structure is depicted in Figure 4.2.

5 Algorithms for ExMLGF

Insertion of a single record to ExMLGF consists of searching for the region that the record belongs to and adding the record in the appropriate block. Deletion of a record consists of searching for the record to be deleted and deleting it subsequently from the appropriate block. Due to limitations on space, we do not include the detailed algorithms for insertion, deletion, basic search and some other algorithms in this paper, but included in [1].

5.1 Algorithms for Point and Aspatial Queries

We get single elements such as a point, population of a city etc., as the input and we search this input in ExMLGF. This algorithm can be utilized for aspatial queries such as "give the cities which have a population 1 million". The value of the population is our input. This search requires a partial match. Notice that we have three kinds of queries to deal with. These are exact match query, partial match query, and range query.

The point in a spatial database is specified either in the form of z-code or as x, y coordinates. Given a point x, y the z-code of this point can be found by interleaving the binary representation of decimal numbers x and y.

\[ Z_{x,y} = X_2 \text{SHUFFLE} Y_2 \]

5.2 Fuzzy and nonfuzzy aspatial queries

In order to deal with fuzzy aspatial queries and nonfuzzy queries efficiently, we include preprocessing and postprocessing steps. For instance, a query such as "Find the cities that are very crowded" is a fuzzy query and must be dealt with in a special way. For these kind of queries we have to access to the database and generate bit representations for the fuzzy data. Then the representation of the population becomes homogenous in the structure. As the result of the post processing step for all aspatial queries we can eliminate the excess data. Since we have the same population for every z-value of an object, more than one value for the population of a city are generated. With the post processing step we eliminate the records having the same object name.

5.3 Containment and closest object queries for point data

An example for a containment query may be "Find all data objects containing the given point." The z-value of the point is searched by deleting the last bit every time. Because the subvalues of the z-value may exist in the object list.
<table>
<thead>
<tr>
<th>Record NO</th>
<th>Cityname</th>
<th>Population</th>
<th>Z-code</th>
<th>Region no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Trabzon</td>
<td>01000100101110111000</td>
<td>010110</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Trabzon</td>
<td>01000100101110111000</td>
<td>0100111</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Trabzon</td>
<td>01000100101110111000</td>
<td>011100</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Trabzon</td>
<td>01000100101110111000</td>
<td>0111100</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Trabzon</td>
<td>01000100101110111000</td>
<td>01100101</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Giresun</td>
<td>00010001010110111010</td>
<td>010100</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Giresun</td>
<td>00010001010110111010</td>
<td>0101100</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Rize</td>
<td>00100000111111111000</td>
<td>01110</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>Rize</td>
<td>00100000111111111000</td>
<td>110100</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>Rize</td>
<td>00100000111111111000</td>
<td>1101100</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>Gumushane</td>
<td>0000000011110001001</td>
<td>011001</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>Gumushane</td>
<td>0000000011110001001</td>
<td>0110110</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>Erzurum</td>
<td>1001000010000000000</td>
<td>11000000</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>Erzurum</td>
<td>1001000010000000000</td>
<td>11000010</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>Erzurum</td>
<td>1001000010000000000</td>
<td>001111</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>Erzurum</td>
<td>1001000010000000000</td>
<td>100101</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>Erzurum</td>
<td>1001000010000000000</td>
<td>01101010</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.1: The table for z-codes and population of cities

Figure 4.2: ExMLGF Structure after inserting the records
An example for a closest object query may be "Find the data object closest to the given point." For the point we first find the left, right, up, down, upleft, upright, downright and downleft points and look whether these points overlap with any object in ExMLGF. This operation is done until an overlapping object is found. Actually we make a circle (buffer) around the point. The first found object is the closest object to the point. For a fuzzy search we assign a threshold value for performing this buffer operation. This time the queries such as 'the objects near to the given point' will also be possible. The detailed algorithms for these queries are given in [1].

5.4 Algorithms for Polygon Objects

We also handle extended objects such as polygon, rectangle, etc. For an extended object we look at containment query, overlapping query and closest object query. In order to deal with an extended object, first of all, the z-values of the object is stored into a data structure (i.e., an array) and then a join operation between the data structure and MLGF is performed. Suppose that we have an extended object $O$ with z-values $Z_i$ where $i = 1, 2, ..., n$. Then OBJ.z[i] contains all z-values and OBJ.name contains the city name.

After storing the extended object into an array, we should consider different query types. The algorithms for overlapping queries, e.g., "Find all data objects overlapping with the given object", containment queries, e.g., "Find all data objects containing the given object", and closest object query, e.g., "Find the data object closest to the given object" have been developed. The pseudo codes of these algorithms are included in [1].

6 Conclusion

In this paper we introduced an index structure that combines MLGF and z-ordering technique for handling not only spatial and aspatial data, but also fuzzy querying in spatial databases. The algorithms developed are implemented (by using C language) and tested with various data. The dimensions of ExMLGF can be increased so that more aspatial attributes or alphanumeric information for spatial data can be included in the index structure proposed here. At this time it seems that the size of index structure can be nearly 4 (four) times the number of objects. This is because, each object has more than one z-value and each z-value is stored into the structure. To decrease such excess information for aspatial queries, we also introduced a post-processing step in the algorithm.

Optimizing the size of the index structure and query processing is our future work. We wish to use our indexing method in a real GIS application. Our effort for designing an efficient algorithm for the spatial join of two ExMGLF structures is also our ongoing research.

References