Implementing Geospatial Operations in an Object-Relational Database System*

Johann-Christoph Freytag, Miroslav Flasza[†], Michael Stillger[‡]
Institut für Informatik, Humboldt-Universität zu Berlin
freytag@dbis.informatik.hu-berlin.de

Abstract

Over the last decade the need to implement functions into a DBMS that are application-specific has increased. For this reason today most object-relational DBMS (ORDBMS) provide features that allow the user to include application-specific functions into the DBMS for their execution within database queries.

This paper reports on an implementation effort to include spatial operations into an ORDBMS as a basis to support geographic information systems (GIS). Based on the technique of multi-step query processing using z-values we show how to transform "straight forward" queries into more sophisticated ones that use the existing ORDBMS as is. That is, existing index methods and existing optimization techniques are sufficient to execute the rewritten queries efficiently. For this purpose we introduce several user-defined functions and types that provide the necessary basis for an efficient implementation.

To validate our implementation we used a subset of queries as defined in the SEQUOIA2000 benchmark. Our measurements show that the performance improvements between the original queries and the rewritten ones are dramatic.

1. Introduction

Over the last decade various proposals have been investigated how to extend current relational database management systems to better serve the more complex requirements of applications areas such as CAD/CAM or geographic information systems. Each kind of application requires specific tailoring of the DBMS with new data types, new operations, and new storage and access methods. To better support these kind of applications, a new generation of relational DBMSs has been developed, called object-relational DBMSs (ORDBMS). Those have specifically been designed for extensibility and adaptability. They are also based on the experience of the "first-generation" relational DBMSs.

Several ORDBMS support the concept of abstract data types – well known from programming languages – to provide a powerful "extension" mechanism. By allowing (sophisticated) users to add new data types and functions, the "kernel DBMS" can be tailored to the specific needs of different application domains. These (user-defined) functions (UDFs) and (user-defined) types (UDTs) provide the necessary efficiency that is needed for such systems.

While the functional integration is a major step in providing extensibility to customers, there exists the continuous need to improve the coordination and cooperation between UDT/UDF facilities with other database components. Specifically, the integration of these features into query optimization is a difficult task.

This paper reports on an implementation effort to investigate the integration of UDTs/UDFs and query optimization for IBM's object-relational DBMS DB2 UDB. For this purpose we choose to focus on *spatial data management* as our application domain. The implementation of a set of UDTs and UDFs provided the basis to better understand the intrinsic difficulties to use UDFs and UDTs in a challenging application environment.

Geographical information systems (GIS) have been studied carefully in the context of extensible database systems over the last years. Many research projects such as POSTGRES, PROBE, STARBURST focused on different aspects of extensibility; all of them used aspects of a GIS as one of their "show cases". Common "objects" in this domain are circles, lines, rectangles,

*This work was supported by a grant of the IBM Almaden Research Center, CA, USA

[†]Author's current address: DB2 Universal Database Development, IBM Toronto Lab, Canada, e-mail: miro.flasza@ca.ibm.com

[‡]Author's current address: IBM Almaden Research Center, CA, USA, e-mail: stillger@almaden.ibm.com

polygons (with an arbitrary number of edges) for the two-dimensional (2D) space, and cubes, cylinders and pyramids for the three-dimensional (3D) space. These objects lead to complex structures and relationships in large number. Their representation is usually based on the vector model, i.e. for the 2D space the data is a sequence of two-dimensional points; for 3D space those data are combinations of 2D objects. In many cases the most expensive operations on those objects are topological binary relations between spatial objects such as equal (in shape), disjoint, overlap, inside, distance, area, touches, or west-Of, south-Of all of which lead to spatial selections or spatial joins based on the topology of the objects related. These operations are usually CPU-intensive and require the DBMS to retrieve the topological description for each object, thus leading to a large I/O overhead. We show how to integrate such operations efficiently into an existing ORDBMS using "object-approximation" to realize a two-step optimization scheme. This approach ensure that spatial queries are executed very efficiently.

We are aware of IBM's Spatial Extender product which is an extension to IBM's DB2 UDB database system (see, for example IBM's WEB-page: http://www-4.ibm.com/software/data/db2/extenders/spatial.htm). The major difference between our and their approach is that we provide an extension "on-top", i.e. our approach does not change the DBMS while IBM's Spatial Extender changes the internals of DB2 UDB.

Our paper is organized as follows. Section 2 introduces the approximation-based filter step using z-values. We show how z-values allow us to perform selections and (spatial) joins efficiently using B-tree indexes. Section 3 shows how to implement the two-step query evaluation approach for a selected set of spatial predicates in DB2 UDB. In particular, we show how to rewrite an (user-submitted) query into one that uses the z-values in a two-step query evaluation step. In Section 4 we present performance results based on measurements we made using the SEQUOIA 2000 benchmark. Finally, Section 5 summarizes our work using DB2 UDB with respect to its extensibility in the GIS domain.

2. Spatial Query Processing

2.1. Object-Approximation based on Z-Values

One of the main approaches for executing queries that "access large amounts of multi-dimensional data" efficiently uses the idea of "*'approximation-based preprocessing*" which has also been called *geometric filtering* [16, 2]. This idea is based on the observation that many operations on spatial objects can be decided based on operations using their approximation. Those operations are often simpler and faster to execute.

Based on the size and the complexity of the data in GIS, the approximation-based approach seems to have an advantage compared to an approach that operates immediately on the exact geometry of spatial objects. First, an approximation usually take less space than the original geometry, due to less complexity in representation; Second, based on a simplified geometry of the approximation the operations are less costly, i.e. less CPU- and I/O-intensive. For example, to test overlapping between two rectangles (with parallel vertices) can be implemented with four comparison operations. There are the following requirements regarding a suitable approximation method:

1. The method should approximate the spatial object as well as possible
2. It should be the basis for efficiently supporting operations on spatial objects.
3. The approximation should use little storage spaces.
4. The resulting values of the approximation should be suited for being stored in an index.

Currently, the two most commonly used (conservative) approximation of geospatial objects are minimal bounding rectangles (MBRs) and approximation schemes based on overlaying the geospatial object with a regular grid pattern.

The MBR-approximation is a simple and intuitive way of approximating arbitrary geospatial objects. Its strength is its simplicity and it fulfills the requirement for minimal storage space. Rectangles with edges parallel to the X-/Y-axis can be represented by their lower left and upper right corner elements (four values in 2D space). Testing for overlap or for inclusion can be done with four arithmetic operations. These approximations can be stored easily in spatial index structures such as R-Trees [13, 20, 1]. The only requirement that is less well-supported is the quality of the approximation, which was studied in [3, 2].

An alternative method for approximating geospatial objects is to "partition" the complete "data space", i.e., a grid overlays the data space, thus approximating a given object by a set of grid elements. An important trade-off is the choice of the granularity of the grid: The finer the grid, the better the approximation. However, a grid of fine granularity also increases number of grid elements for the approximation, thus increasing the necessary storage space.

In order to store and manipulate a set of grid elements efficiently, we must provide an additional mapping function that allows us to number those grid elements. This enumerating function is a mapping from the 2-dimensional space into a one-dimensional space to provide a total ordering of all grid elements. To reflect the neighboring relationship of grid elements (i.e. grid elements that are neighbors in the 2D space should also have numbers next to each other) we must find an appropriate mapping and numbering function. Such functions, called space-filling curves, have been developed such as Hilbert curves [6, 14] and Peano Curves (also called z-ordering) [15, 16] (see Figure 1).

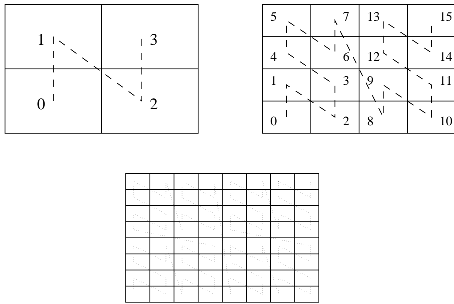

Figure 1. Z-Ordering: Enumeration of Grid Elements with different Granularity

The approximation approach based on grid elements fulfills the requirements set above. It can be tuned by increasing the granularity to increase the quality of the approximation at the same time increasing the amount of storage space. We can reduce the storage requirement for each element to the space requirement of an integer value with the right choice of mapping function (as described later). Using grid elements allows us to implement efficient algorithms to test for inclusion and overlap. The major advantage of the grid-based approximation method is that the grid elements can be stored in existing (one-dimensional) index structures to improve the evaluation of queries.

Any of the two alternative approximations can be used to implement a 2-step approach to process any spatial query. In the first step (also called "filtering step") a set of possible candidates for the result are determined using the object approximations. This candidate set contains at least all the objects that belong to the final result. In a second step (also called "refinement step") each member of the candidate set is tested on its exact geometry (using the original spatial function) whether it belongs to the final answer or not. The underlying assumption is that the fist step results in a much smaller set of spatial object to which to apply the (costly) spatial function compared to an execution without approximations (see Figure 2).

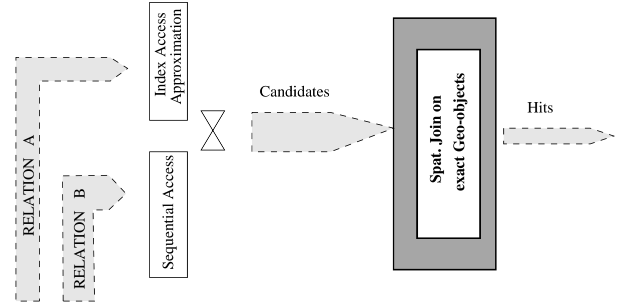

Figure 2. Z-Ordering: 2-step Execution of Spatial Queries

2.2. Properties of Z-Values

To approximate a given geospatial object, we overlay the object with an initial rectangle such that the object is included in the rectangle. By recursively splitting the rectangle vertically or horizontally in alternation, we construct a grid whose elements become smaller and smaller. Thus the initial approximation of the object by the (initial) rectangle is "refined".

Each grid element is assigned a unique "identifier" as follows. For each each vertical (respectively horizontal) split, the newly generated rectangle to the left (respectively bottom) of the splitting line is assigned an identifier that is created by appending a "0" to the already existing identifier for the initial rectangle currently split. Similarly, the newly-generated rectangle to the right (respectively top) of the splitting line is assigned an identifier that is created by appending a "1" to the already existing identifier for the initial rectangle that is currently split. Figure 3 shows the recursive procedure for splitting and for approximating the given polygon by a set of z-values. As already mentioned previously, we must determine the granularity of the grid in advance, i.e. what the maximum depth of the recursion is.

In case of using an approximation with a grid of fixed granularity testing two objects for overlap is trivial: If the intersection of the two sets of grid elements is non-empty then the approximations overlap; if the intersection is empty both objects do not overlap. Testing the two sets of elements is trivial as well: either two elements are identical or they are distinct.

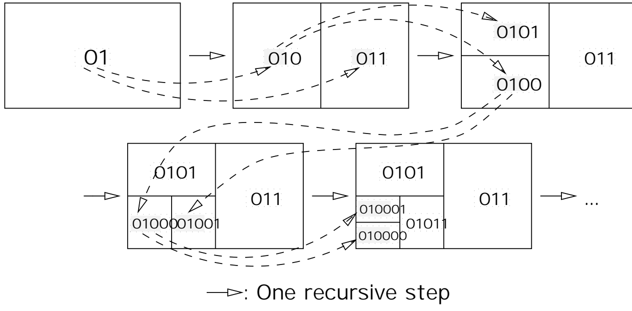

Figure 3. Recursive Construction of Z-Values

However, the test for intersection becomes more difficult if we allow grids of variable granularity to approximate objects. The set of grid elements is not homogeneous any more; we need the additional operation *inclusion* to relate two grid elements. Using z-ordering, we can identify the granularity of each element based on the length of the corresponding bit sequence. For example, if the z-values resemble a coding of length between 1 and $maxlng$, then the "finest" grid elements are of length $maxlng$. In contrast, the z-value of length 1 resembles a grid element which represents half of the complete data space.

Given two z-values z_1 and z_2, we need an efficient approach to determine their topological relationship. For this reason we use the following property based on the z-ordering used:

Lemma 1 (Prefix Property) *Let e_1, e_2 be two grid elements, $z(e_1)$, $z(e_2)$ be the corresponding encodings using z-values with a particular ordering. Then e_1 is included in e_2, if and only if $z(e_2)$ is a prefix of $z(e_1)$.*

As an immediate consequence of Lemma 1 we can derive some additional properties which will later help us to determine efficiently the containment of one element in another.

Lemma 2 *Let $maxlng$ be the maximum length for any code of the grid elements using the z-ordering; let e_1 be an element of the coding of length n, $n < maxlng$ represented by a bit sequence $b_1 \ldots b_n$. Then all coding contained in e_1 (that is, all coding having been derived from e_1 and therefore having e_1 as a prefix) share the following property:*

$$b_1 \ldots b_n \overset{lex}{\leq} z(e) \overset{lex}{\leq} \underbrace{b_1 \ldots b_n 1 \ldots 1}_{maxlng},$$

for all e contained in e_1, and with $\overset{lex}{\leq}$ being the lexicographical comparison operator.

Therefore, we can determine topological relationships between spatial objects by using the approximations using grid elements. If we take into account variable grid sizes, there are only three relationships possible between two grid elements. Either they do not overlap, or e_1 is contained in e_2, or e_2 is contained in e_1. We treat identity of two elements as a special case of overlap. According to Lemma 2, this simple relationship is reflected by the following equivalence relationship:

$$e_1, e_2\ overlap \iff \begin{pmatrix} z(e_1) \overset{lex}{\leq} z(e_2) \overset{lex}{\leq} zHi(e_1) \\ or \\ z(e_2) \overset{lex}{\leq} z(e_1) \overset{lex}{\leq} zHi(e_2) \end{pmatrix} \quad (1)$$

According to [15], $zHi(e)$ denotes the coding of element e as a bit sequence extended with "1" up to its maximal length.

To determine if two objects overlap based on their z-values, one could apply the above formula on the crossproduct of all z-value elements. However, this would lead to an $O(nm)$ complexity with n and m being the cardinality of the corresponding z-value sets. In [16] Orenstein and Manola describe an efficient algorithm for merging two **ordered** sequences of z-values. Similar to a "sweep-line" algorithm, this algorithm achieve an $O(n+m)$ time complexity. For more details we refer to [16]. There, they suggest a recursive coding procedure that generates a bit sequence by appending either "0" or "1" to the code generated previously. Such a bit sequence is then stored in a B-Tree. In contrast, Gaede suggests to use an integer representation instead of a bit sequence [8]. He constructs a mapping function from bit sequences to integers that maintains all the properties of the encoding as described in [16]. The advantage of the integer encoding is reduced space for storage and faster execution of comparison operations (integer versus variable-length bit sequences). Specifically, the integer representation allows us to use comparison operators for integers rather than lexicographical comparison operators on the bit values. The space needed for each z-value is therefore static and independent of the resolution used for the grid elements. Altogether this choice will speed up the use of z-values during query execution. Furthermore, z-values are stored as elements in existing B-Trees. Any z-value can be part of any spatial object and any object is approximated by a set of z-values. The B-Tree structure is well suited to handle this kind of relationship using z-value approximations.

For this reason we introduce the z-values as integer coding directly with the following function. In each step we generate the new encoding for the sub-grid by

the formula:

$$
\begin{aligned}
leftOrBottom(zVal) &= zVal + 1 \\
rightOrTop(zVal) &= zVal + 2^{(maxD - curD)}
\end{aligned}
\tag{2}
$$

leftOrBottom (respectively *rightOrTop*) denotes the coding of the left or lower grid element (respectively right or upper element) depending on a vertical or horizontal splitting line. *zVal* denotes the element to be split, and *curD* and *maxD* represent the current depth and the maximum depth of the recursion to run the algorithm, respectively. The maximal recursion depth must be present and determines the desired grid resolution of the geospatial object to be approximated and thus the granularity of the approximation. Figure 4 shows an example for our coding function to generate z-values. The maximal recursion depth is 6 ($maxD = 6$). To demonstrate the splitting of elements we pick as an example the coding with $zVal = 33$. Since this element was previously generated by a horizontal split, we must generate two "child elements" by splitting vertically. According to our coding function we generate the left child as $33 + 1 = 34$ and the right child by the coding $33 + 2^{6-2} = 33 + 2^4 = 33 + 16 = 49$. In order to enumerate all grid elements generated by the function *decompose* we need exactly $2^{(maxD+1)} - 1$ z-values. In our example, the number of z-values is $2^{6+1} - 1 = 2^7 - 1 = 127$.

In [16] Orenstein describes a method that uses the prefix-property of two bit-sequences. This property was described by Lemma 1 and Lemma 2. Our coding functions using integers, provides the same order as the one designed by Orenstein: it is a space-filling Peano curve, also called z-ordering. However, the lexicographical ordering of bit sequences is replaced by the canonical ordering of integer values. The prefix property as used for bit sequences, does not map into the integer-based representation of z-values. Therefore, the prefix property must be replaced by another one that correctly reflects the parent-child relationship between z-values.

We say that grid element e_1 is a **parent** element of e_2 if and only if $e_1 = e_2$ or if e_2 has been generated from e_1 by one or more splits. Similarly, we say that e_2 is a **child** element of e_1 if the same relationship holds. This relationship hold for the z-values (coding of grid elements) as well: $z(e_1)$ is a parent of $z(e_2)$ if e_1 is a parent of e_2. The same holds for the child relationship.

In [8], Gaede calls the set of all ancestors of z the $upperHull(z)$ of z. Therefore the set $upperHull(z)$ is equal to the set of all codings in which z is contained. Similarly, the set of all z-values that have z as their prefix are called the $lowerHull(z)$ of z and is equivalent to all descendants of z, i.e., the set of codings of all grid

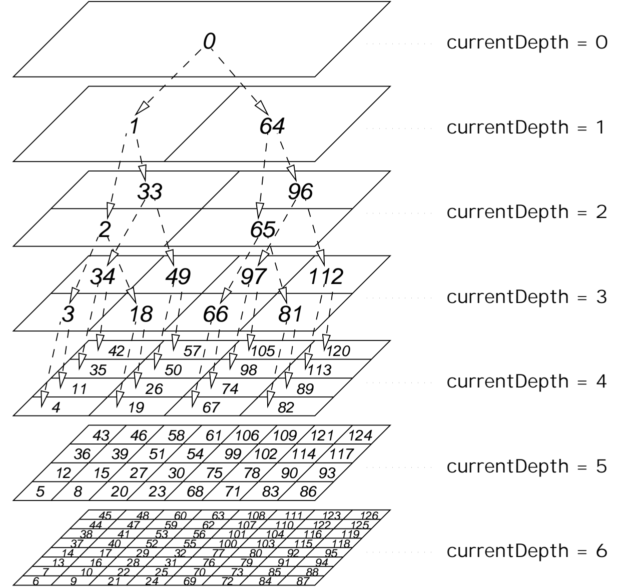

Figure 4. Coding of Z-Values with a maximal depth of recursion of 6

elements that are contained in the element coded by z. As an example for the upper and lower hull we use all z-values as shown in Figure 4. The upper hull for grid element 33 is $upperHull(33) = \{0, 1, 33\}$ and the lower hull for element 33 is $lowerHull(33) = \{33, \ldots, 63\}$.

Based on these definitions we are now ready to express several properties that hold for our z-values as representatives of the grid elements as generated by our coding function:

$$
\begin{aligned}
e_2 \textit{ contains } e_1 &\iff z(e_1) \in lowerHull(z(e_2)) \\
&\iff z(e_2) \in upperHull(z(e_1))
\end{aligned}
\tag{3}
$$

The correctness of these equations follows from the definition of the upper and lower hull and the generation of z-values.

To compute the lower hull for a given z-value, we use an important property of our coding function that guarantees an enumeration of grid elements according to our z-order. We observe that all codings of grid elements that are generated by splitting a specific element e with coding $z(e)$ (i.e. elements of $lowerHull(z(e))$) are contained in the interval of consecutive elements. The interval is limited by $z(e)$ and $zHi(z(e))$, i.e. $[z(e), zHi(z(e))]$ with $zHi(z(e))$ being the maximum element in $lowerHull(z(e))$. In our encoding, $zHi(z(e))$ is the element that can be found on the highest recursion level in the upper right corner of

the rectangle that contains child elements of e. To illustrate, we again use the grid element with coding 33 (on level 2) in Figure 4. The value for $zHi(33)$ is equal to 63, the highest-numbered descendant of 33, which will be the upper-right element at the lowest level of the recursion (level 6).

Therefore we are able to determine inclusion for our encoding for grid elements based on our z-values as follows:

$$(z(e_1) \geq z(e_2)) \wedge (z(e_1) \leq zHi(z(e_2))) \implies z(e_1) \in lowerHull(z(e_2)) \tag{4}$$

Therefore e_2 contains e_1 with zHi to be defined by:

$$zHi(zVal) = zVal + 2^{(maxD-(depth(zVal)-1))} - 2 \tag{5}$$

The test for overlap of two grid elements can now be reduced to testing the inclusion of one element in another. Thus, we can immediately apply Formula 4 to derive the following relationship:

$$\begin{array}{c} (z(e_1) \geq z(e_2)) \wedge (z(e_1) \leq zHi(z(e_2))) \\ or \\ (z(e_2) \geq z(e_1)) \wedge (z(e_2) \leq zHi(z(e_1))) \\ \implies \quad e_1 \ overlaps \ e_2 \end{array} \tag{6}$$

This is equivalent to:

$$\begin{array}{c} (z(e_1) \geq z(e_2)) \wedge (z(e_1) \leq zHi(z(e_2))) \\ or \\ z(e_1) \in upperHull(z(e_2) \\ \implies \quad e_1 \ overlaps \ e_2 \end{array} \tag{7}$$

Later we use this equation in our queries to support query processing based on the approximation-based approach.

3. Implementation in DB2 UDB

Based on the general idea as discussed in the previous chapter we implemented a set of data types and a set of functions using the vector-oriented representation of geospatial spatial data [17]. The implementation was done for DB2 UDB Version 5 Beta. Furthermore, the goal of the implementation was to apply the ideas of the approximation based preprocessing of geospatial queries (geometric filters)) based on z-ordering of Orenstein [15, 16] in combination with a 2-step query processing approach. This section describes the different aspects of the implementation and the design decisions made in detail.

With respect to UDFs and UDTs in DB2 UDB we used the language constructs as described for the product V5.0. Regarding the implementation of UDFs we use both functions, that return a single value (so-called *scalar* functions), and functions that return sets of tuples (so-called *table* functions). The latter option is used to generate temporary relations in the FROM-clause of queries by providing parameters to the functions defined. We use the language C to program the different geospatial UDFs after having specified their interfaces.

3.1. User-Defined Types (UDTs)

We limited our geospatial functions to represent and to manipulate two-dimensional (2D) objects such as lines, points, and regions (polygons) [11]. These "objects" provide the basic building blocks for more complex geometric entities. A similar collection of objects has been implemented in other research projects as well such as PSQL [18], O_2 [19], and the ROSE Algebra [12].

For our prototypical implementation in DB2 UDB, we focused our implementation on those data types and functions that were necessary to support the SEQUOIA 2000 benchmark [21], i.e. we implemented (2D) points, rectangles (with edges parallel to the x-axis and y-axis), circles, and "simple polygons" (without "holes"). For this purpose we defined several

GEO Entity	C Data Type Definition	UDT Definition (DB2 DDL)

Point	`struct cGeoPoint {`
	`long x;`
	`long y;`
	`};`
		`CREATE DISTINCT TYPE`
		`GeoPoint AS`
		`CHAR(8) FOR BIT DATA`
		`WITH COMPARISONS;`

Circle	`struct cGeoCircle {`
	`cGeoPoint cntr;`
	`long rad;`
	`};`
		`CREATE DISTINCT TYPE`
		`GeoCircle AS`
		`CHAR(12) FOR BIT DATA`
		`WITH COMPARISONS;`

Iso Rectangle	`struct cGeoRect {`
	`cGeoPoint ll;`
	`cGeoPoint ur;`
	`};`
		`CREATE DISTINCT TYPE`
		`GeoRect AS`
		`CHAR(16) FOR BIT DATA`
		`WITH COMPARISONS;`

Simple Polygon (without holes)	`struct cGeoPoly {`
	`int size;`
	`cGeoPoint inside;`
	`cGeoRect bbox;`
	`cGeoPoint boundary[];`
	`};`
		`CREATE DISTINCT TYPE`
		`GeoPoly AS`
		`BLOB(64K);`

Figure 5. Geo-Datatypes for Main Memory Representation and for DB2 UDB

UDTs: points, circles, 2-D rectangles, (2D) polygons. Their definitions are summarized in Figure 5.

3.2. User-defined Functions (UDFs)

We distinguish four types of geospatial operations which we implemented as functions in DB2 UDB: functions that construct spatial objects of different types, functions that access attributes of spatial objects, functions that determine the (geospatial) relationship between different spatial objects, and utility functions.

For each of our user-defined types we provide one constructor function that takes one of more integer values and turns them in to the appropriate type. For example, the function GeoPoint takes two integer values returning an object of type point. Similar functions are provided for circles, rectangles, and polygons.

To express queries over spatial data we need spatial operations and functions that can be embedded into the query. Specifically, we need functions to determine spatial relationships between geometric objects. In [11] Güting distinguishes three categories of relationships between spatial objects: topological relationships, "positional" relationships, and metrical relationships.

According to Egenhofer there are six possible topological relationship between 2D-regions [5]: Disjoint, touches, equal, covers (covered by), contains (inside), and overlaps. Accordingly, there are three different topological relationships possible between 2D-regions and points: disjoint, borderContains (onborder), contains (inside). We also implemented "positional" and "metrical" functions such as *equal*, *disjoint*, *inside*, *contains*, *overlap*, and *touches*; however due to lack of space we do not give any further details. More details can be found in [7]. The result of all geospatial functions for spatial relationships are Boolean values. Since DB2 UDB does not support the data type BOOLEAN as a return value for scalar UDFs, we instead used the integer values *1* and *0* as return values, respectively.

For the approximation-based query processing using z-values we implemented several "utility" functions as already introduced in the previous section:

- **ZDecompose**: decomposes a given spatial object into a set of z-values;
- **zHi**: computes the values *zHi* for a given z-values as previously described;
- **zUpperHull**: computes a set of z-values for a given z-values that represents the upper hull.

3.3. Multi-Step Evaluation of Spatial Queries

To implement the multi-step query processing approach in an existing DBMS we described the general idea using a simple example. This approach is used later for all queries in the SEQUIOIA200 Benchmark to include a 2-step query processing approach that includes the approximation-based filtering step.

Consider the following simple database schema consisting of two relations:

```
POLYGON(id: Int, attr1: ..., ...,
                 attrN: ...,
                 shape: GeoPoly)
POLYGON_Z(id: Int, zval: GeoZValue)
```

Relation POLYGON stores the polygon data together with other non-spatial attributes attr1, ..., attrN; attribute shape contains the spatial description of the polygon. Relation POLYGON_Z contains the z-value approximations of all polygons stored in POLYGON. Furthermore, we assume that storing POLYGON uses more space than storing relation POLYGON_Z. In addition, we assume that there exist clustering indexes on POLYGON.id and POLYGON_Z.zval, which we call P_INDX_ID and PZ_INDX_Z, respectively.

Consider the following query which perform a spatial selection using a "query window". It returns all polygons that overlap with the rectangles specified by the lower left and upper right corner (X0, Y0) and (X1, Y1).

```
SELECT *
FROM   POLYGON P
WHERE  overlaps(P.shape,
                GeoRect(X0,Y0,X1,Y1)) = 1
```

To preprocess this query, we must check if part of the approximation of the query window is contained in any of the object approximations using Formula 7. To apply this formula we need to compute the approximation of the query window by using the table UDF ZDecompose and to compute the union of all upper hulls by calling the table function ZUpperHull.

```
SELECT DISTINCT PZ.id
FROM   POLYGON_Z PZ,
       TABLE(ZDecompose(
        GeoRect(X0,Y0,X1,Y1))) ZD,
       TABLE(ZUpperHull(ZD.zval)) ZU
WHERE  (PZ.zval BETWEEN ZD.zval
           AND ZHi(ZD.zval))
       OR (PZ.zval = ZU.zval)
```

This filtering step results in a set of object identifiers whose z-values overlap with the (dynamically created) approximation of the query window. Preferably, the DBMS uses index PZ_INDX_Z to evaluate the predicate (once for the interval predicate and once for the quality predicate) efficiently, assuming that the number of elements in the dynamically created tables is much smaller than in POLYGON_Z. Additionally, index-ORing provides an almost optimal evaluation of the filtering predicate (for index-ORing see [9]).

While this first step (called preprocessing step) identifies the set of possible candidates for the final result, it is now necessary to evaluate the original spatial predicate on each candidate identified by its object identifier to compute the final result. For this reason the

previous SQL statement must be "embedded" into the original query as follows

```
SELECT P.*
FROM POLYGON P,
     (SELECT DISTINCT PZ.id AS id
      FROM   POLYGON_Z PZ,
             TABLE(ZDecompose
              (GeoRect(X0,Y0,X1,Y1))) ZD,
             TABLE(
              ZUpperHull(ZD.zval)) ZU
      WHERE  (PZ.zval BETWEEN ZD.zval
                     AND ZHi(ZD.zval))
             OR (PZ.zval = ZU.zval)
     ) AS ZFILTER
WHERE P.id = ZFILTER.id AND
       overlaps(P.shape,
         GeoRect(X0,Y0,X1,Y1)) = 1
```

4. Benchmarking

To better understand the improvements of our implementation we decided to benchmark our solution in DB2 UDB V5. While there already exist several benchmarks for the business world [10], there are few acknowledged benchmarks for the GIS domain. One of them is the SEQUOIA 2000 Benchmark, which was developed as part of the SEQUOIA 2000 project [21]. However, we did not implement all features of the SEQUOIA 2000 benchmark. We focused on those parts that could be easily implemented using the spatial operations as described in the previous section. The parts of the benchmark we implemented, conform to the requirements that were described in the original benchmark. Due to space limitations we focus on two queries of the SEQUOIA2000., other can be found in [7].

All measurements were performed on a SUN SPARCstation 10 with two TI TMS390Z50 SuperSPARC/50MHz processors using symmetric multiprocessing (SMP); the memory size was 128 MB. The operating system was Solaris 2.5.1. We used the latest DB2 UDB available to us in June of 1997 (Beta 5). Since we wanted to take advantage of parallelism, all measurements were performed twice: once without using the SMP capabilities, and once with. The buffer space for both cases was limited to 16MB (according to the measurements reported in [4]). The data was stored in two UNIX-file containers using a 2.1GB SCSI disk of type Seagate ST42400N.

To perform the benchmark, we used part of the official data set that is publicly available on the ftp-server s2k-ftp.cs.berkeley.edu. We restricted our set to the local data of California. Within this set we find four different kinds of data: point data, polygon data, graph data, and raster data. We restricted our data set to point and polygon data.

All point data represent points of interest in California. They are recorded with their name and their position in meters in a predefined coordinate system. There are 62537 point entities described in the local data set, which occupies about 2MB of disc space. The set of polygon data consists of 79607 data entries, each of which represent areas with a particular land-use within California. The boundaries of each polygon are described by a list of coordinates that represent the edges of the polygon. Among the 79607 polygons there are 21021 which contain "holes", so-called "swiss cheese" polygons. Each polygon is classified according to its land-use, based on the classification used in the United States Geological Survey (USGS). The classification consists of about 35 different categories. The number of edges for each polygon varies between 1 and 5583; the average number of edges is 49. The total number of edges of all polygons is about 3,900,000.

As a database schema we used the following tables and indexes:

```
CREATE TABLE POINT
       ( name     VARCHAR(80) NOT NULL,
         location GeoPoint NOT NULL,
         zval     GeoZValue NOT NULL
                    WITH DEFAULT );
CREATE TABLE POLYGON
       ( landuse  INTEGER NOT NULL,
         shape    GeoPoly NOT NULL
                    NOT LOGGED COMPACT,
         id       INTEGER NOT NULL );
CREATE TABLE ISLAND
       ( landuse  INTEGER NOT NULL,
         shape    GeoPoly NOT NULL
                    NOT LOGGED COMPACT
         id       INTEGER NOT NULL );
CREATE TABLE POLYGON_Z
       ( id       INTEGER NOT NULL,
         zval     GeoZValue NOT NULL );
CREATE TABLE ISLAND_Z
       ( id       INTEGER NOT NULL,
         zval     GeoZValue NOT NULL );

CREATE INDEX PT_NM ON POINT(name);
CREATE INDEX PT_Z ON POINT(zval);
CREATE INDEX POLY_LU ON POLYGON(landuse);
CREATE UNIQUE INDEX POLY_ID
              ON POLYGON(id);
CREATE UNIQUE INDEX POLY_Z
              ON POLYGON_Z(zval, id);
CREATE UNIQUE INDEX ISLAND_ID
```

```
                ON ISLAND(id);
  CREATE UNIQUE INDEX ISLAND_Z
                ON ISLAND_Z(zval, id);
```

Relation ISLAND contains the "holes" for those polygons that are stored in relation POLYGON. The combination of both relations allows us to represent the set of "swiss cheese" polygons. We realize that this representation is only suboptimal. However, we followed the suggestions of [21] and [4], who used the same representation in their benchmarks.

The benchmark database was generated in three steps: First we loaded the relations using the text-based representation of the data. In a second step all necessary indexes were generated. We note that the database size increases only minimally when including z-values. The additional space required is about 8MB, i.e. an increase of about 7% when compared with the original database without z-values.

To demonstrate the large improvements of a two-step query processing approach using z-values for spatial approximations we show the measurements for QUERY 7 and QUERY 10 of the SEQUOIA200 Benchmark.

SEQUOIA 2000 Query 7 *This query returns all polygons that are included in a circle as specified in the query and whose area is larger than a given value FLOAT-VALUE.*

```
SELECT P.landuse, P.shape
FROM   POLYGON P
WHERE  inside(P.shape,
         GeoCircle(CNTR, RAD)) = 1 AND
       area(P.shape) > FLOAT-VALUE;
```

Using z-values the query was rewritten as follows:

```
SELECT P.landuse, P.shape
FROM   POLYGON P,
       (SELECT DISTINCT PZ.id AS id
        FROM   POLYGON_Z PZ,
        TABLE(ZDecompose(
           GeoCircle(CNTR, RAD))
        WHERE  PZ.zval BETWEEN ZD.zval
               AND ZHi(ZD.zval)) AS ZSEL
WHERE  P.id = ZSEL.id AND
       inside(P.shape,
          GeoCircle(CNTR, RAD)) = 1 AND
       area(P.shape) > FLOAT-VALUE;
```

The following table shows that the execution time improves by a factor of 100 in both cases, i.e. the uniprocessor case and the two-processor case.

	1 Processor	
	original	with Z-Values
Time (sec)	72.41	0.67
	SMP with 2 Processors	
	original	with Z-Values
Time (sec)	40.21	0.43

Table 1. Execution Time for Query 7

SEQUOIA 2000 Query 10 *The query returns all points that are located within a polygon of a particular category for land use. Since polygons might contain "holes" this query also accesses relation* ISLAND *to check if the points are not located inside a hole.*

```
  SELECT PT.name
  FROM   POLYGON P, POINT PT0
  WHERE  P.landuse = CODE AND
         inside(PT0.location,
                P.shape) = 1
EXCEPT
  SELECT PT.name
  FROM   ISLAND I, POINT PT1
  WHERE  inside(PT1.location,
                I.shape) = 1;
```

Including z-values for spatial approximation the Query 10 is rewritten as follows:

```
  SELECT PT1.name
  FROM   POLYGON P, POLYGON_Z PZ,
         POINT PT1
  WHERE  PT1.zval BETWEEN PZ.zval AND
         ZHi(PZ.zval) AND
         P.id = PZ.id AND
         P.landuse = CODE AND
         inside(PT1.location,
                P.shape) = 1
EXCEPT
  SELECT PT2.name
  FROM   ISLAND I, ISLAND_Z IZ,
         POINT PT2
  WHERE  PT2.zval BETWEEN IZ.zval AND
         ZHi(IZ.zval) AND
         I.id = IZ.id AND
         inside(PT2.location,
                I.shape) = 1;
```

This measurements for this query clearly demonstrate the advantage of our approach. While the original query had to be terminated after 19,000 seconds (more than five hours) of execution time without returning a result, the rewritten query returns the results in less

than three minutes in the uniprocessor case, and in about 1.5 minutes for the two-processor case.

	1 Processor	
	original	with Z-Values
Time (sec)	19000+	171.98
	SMP with 2 Processors	
	original	with Z-Values
Time (sec)	19000+	94.83

Table 2. Execution Time for Query 10

Finally, we would like to mention that our measurements compare favorably with measurements done on other systems using the SEQUOIA2000 Benchmark described in the literature [21, 4]. The only exception is QUERY 10 which does not perform very well on DB2. However, this in due to our choice of representing "swiss cheese polygons" in two relations while other implementations use one relation storing the spatial extend of such objects in one BLOB thus avoiding an additional join. More discussion of the measurements can be found in [7].

5. Conclusion

This paper shows how to incorporate application-oriented functions into an ORDBMS. We used DB2 UDB as our ORDBMS using the existing UDT and UDF features to include application functions and functions to approximate (2D) spatial object. The latter functions implement a 2-step query processing approach that significantly improves the execution time of queries as demonstrated by executing several queries of the SEQUOIA2000 Benchmark. DB2 UDB turns out to be an ideal platform to implement and to execute queries with user-defined functions. The optimizer can deal with UDFs during query execution generating an highly efficient query execution plan.

References

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[2] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-Step Processing of Spatial Joins. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 197–208, Minneapolis, MN, May 1994.

[3] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 40–49, Vienna, Austria, April 1993.

[4] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-Server Paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 558–569, Santiago de Chile, Chile, September 1994. Morgan Kaufmann.

[5] Max J. Egenhofer. Reasoning About Binary Topological Relations. In Oliver Günther and Hans-Jörg Schek, editors, *Proceedings of the 2nd International Symposium SSD'91 on Advances in Spatial Databases*, volume 525 of *Lecture Notes in Computer Science*, pages 143–160, Zürich, Switzerland, August 1991. Springer.

[6] Christos Faloutsos and Shari Roseman. Fractals for Secondary Key Retrieval. In *Proceedings of the Eight ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252, Philadelphia, PA, March 1989.

[7] Miroslav Flasza. *Implementation von Geo-Funktionalität in einem erweiterbaren relationalen DBMS, Diplomarbeit (in German)*. Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, 1997.

[8] Volker Gaede. *Extending Query Optimization for Spatial Database Systems.* PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Germany, 1996.

[9] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query Optimization in the IBM DB2 Family. *IEEE Data Engineering Bulletin*, 16(4):4–18, December 1993.

[10] Jim Gray, editor. *The Benchmark Handbook for Databases and Transaction Processing.* Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 1993.

[11] Ralf Hartmut Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3(4):357–400, October 1994.

[12] Ralf Hartmut Güting, Thomas de Ridder, and Markus Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In Max J. Egenhofer and John R. Herring, editors, *Proceedings of the 4th International Symposium SSD'95 on Advances in Spatial Databases*, volume 951 of *Lecture Notes in Computer Science*, pages 216–239, Portland, ME, August 1995. Springer.

[13] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Beatrice Yormark, editor, *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[14] H. V. Jagadish. Linear Clustering of Objects with Multiple Atributes. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342, Atlantic City, NJ, May 1990.

[15] Jack A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, D.C., May 1986.

[16] Jack A. Orenstein and Frank Manola. PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

[17] Donna J. Peuquet. A Conceptual Framework and Comparison of Spatial Data Models. *Cartographica*, 21(4):66–113, 1984.

[18] Nick Roussopoulos, Christos Faloutsos, and Timos K. Sellis. An Efficient Pictorial Database System for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639–650, May 1988.

[19] Michell Scholl and Agnès Voisard. Object-Oriented Database Systems for Geographic Applications: An Experiment with O2. In François Bancilhon, Claude Delobel, and Paris C. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, San Mateo, CA, 1992.

[20] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, Brighton, England, September 1987. Morgan Kaufmann.

[21] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 Benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 2–11, Washington, D.C., May 1993.