Spatial Access Methods and Query Processing in the Object-Oriented GIS GODOT[1]

Volker Gaede
Institut für Wirtschaftsinformatik, Humboldt Universität zu Berlin,
Spandauer Str. 1, 10178 Berlin, Germany
gaede@wiwi.hu-berlin.de

Wolf-Fritz Riekert
FAW Ulm, Helmholtzstr. 16,
89081 Ulm, Germany
riekert@faw.uni-ulm.de

ABSTRACT. In this paper, we describe the spatial access method *z-ordering* and its application in the context of the research project GODOT, which is based on the commercial object-oriented database system ObjectStore [LLOW91]. After identifying a range of spatial predicates, we show that the intersection join is of crucial importance for spatial joins. Next, we propose an efficient method for query processing, which takes advantage of z-ordering and uses the conventional indexing mechanisms offered in current database systems (e.g., relational and object-oriented).

1. INTRODUCTION

Spatial data management is an important database application area where object-oriented database concepts can be utilized efficiently in a variety of ways [GL94]. As a result, such techniques are gradually being integrated into GIS (geographic information system) products, and object-oriented spatial database systems are being implemented on top of commercial OODB systems [GR93, SV92].
A major conceptual problem that arises in this context concerns the definition of the relevant spatial operations and their efficient computation. Many efforts have been undertaken to define a continuum of spatial queries, also known as *query space*, which is complete in a certain sense, and to define data structures which support the efficient computation of these queries (cf. [Ege94, Mol92]). One of the most important spatial operations is the *spatial join*. Günther [Gü93] gives the following

[1]The GODOT project has been conducted at FAW (Research Institute for Applied Knowledge Processing) in Ulm, Germany since 1992. GODOT was commissioned by the State of Baden-Württemberg, Siemens Nixdorf Informationssysteme AG and Siemens AG.

definition of spatial joins:
The spatial join of two relations R and S, denoted by $R \bowtie_{i\theta j} S$, is the set of tuples from $R \times S$ where the ith column of R and the jth column of S are of some spatial type, θ is some spatial predicate, and $R.i$ stands in relation θ to $S.j$. According to the terminology used in geographic information systems and object-oriented databases we will also refer to the tuples $r \in R$, $s \in S$, etc. as *spatial objects* or *geo-objects* and to the relations R, S, etc. as *object extents* or *sets of objects*. We assume (without loss of generality) that one dedicated column in the relations R, S, etc., is of some spatial data type and represents the *geometry* of the related spatial objects. We will therefore write $R \bowtie_\theta S$ as a shorthand for the spatial join $R \bowtie_{i\theta j} S$ whenever i and j denote these dedicated columns in the relations R and S, respectively.
A brief survey of the literature yields a wide variety of spatial predicates that may be used for spatial joins, including

- intersects(r, s)
- contains(r, s)
- is_enclosed_by(r, s)
- distance(r, s)ΘE, with $\Theta \in \{=, \leq, <, \geq, >\}$ and E a given real number
- northwest(r, s)
- adjacent(r, s)

These predicates check if some spatial relationship between two objects is satisfied. Another class of spatial operations which will become important in this context concerns the "transformation" of geo-objects. Examples are ($r \in R, s \in S$):
convex_hull(r), buffer(r, d), northwest(r), geometric_union(r, s), geometric_difference(r, s), etc. A closer inspection of these spatial predicates and operations shows that the *intersection* join $R \bowtie_{\text{intersects(r,s)}} S$ plays a crucial role for the computation of all the other predicates.
For predicates such as contains, encloses, or adjacent, the intersection join is an efficient filter that yields a set of tuples (r, s), typically much smaller than the Cartesian product $R \times S$, which still contains all solutions to the original query. That is, the intersection join is an efficient *preprocessing* step.
Most of the remaining predicates (for example, distance, northwest) can be computed by using the intersection join as a *postprocessing* step as follows. In order to compute the spatial join $R \bowtie_{\theta(r,s)} S$ for one of those θ-predicates, we can apply some function $\phi_\theta : R \to R'$ and $\psi_\theta : S \to S'$ to R and S respectively, then compute the intersection join $R' \bowtie_{\text{intersects(r,s)}} S'$. If the functions ϕ_θ and ψ_θ have been properly defined, a tuple (r, s) belongs to the spatial join $R \bowtie_{\theta(r,s)} S$ if and only if $R' \bowtie_{\text{intersects(r,s)}} S'$. For example, it is possible to compute the northwest query $R \bowtie_{\text{northwest(r,s)}} S$ (asking for all pairs (r, s), where s is situated in the northwest of r) in the following way[2]:

$$\text{northwest}(R) \bowtie_{\text{intersects(r,s)}} S$$

[2]Let the northwest-function for a relation in the following formula be defined as the element-wise application of the northwest-function on the elements of the relation. That is, by using the spatial attribute of the given tuple, the northwest function precomputes the area located to the northwest.

That is, instead of checking the northwest predicate for every pair of tuples in the Cartesian product of both relations, the northwest *function* is applied to the first relation involved, followed by an intersection join with the second relation
With this concept it is possible to reduce the most common spatial joins to an ordinary intersection join, which requires some pre- and/or postprocessing. The following table gives a concise overview of the required transformations.

Postprocessing

1.	$R \bowtie_{\mathsf{is\_enclosed\_by(r,s)}} S$	$\rightarrow$	$\sigma_{\mathsf{is\_enclosed\_by(r,s)}}(R \bowtie_{\mathsf{intersects(r,s)}} S)$
2.	$R \bowtie_{\mathsf{contains(r,s)}} S$	$\rightarrow$	$\sigma_{\mathsf{contains(r,s)}}(R \bowtie_{\mathsf{intersects(r,s)}} S)$
3.	$R \bowtie_{\mathsf{adjacent(r,s)}} S$	$\rightarrow$	$\sigma_{\mathsf{common\_border(r,s) \wedge not(overlap(r,s))}}$ $(R \bowtie_{\mathsf{intersects(r,s)}} S)$
4.	$R \bowtie_{\mathsf{overlay(r,s)}} S$	$\rightarrow$	$\mu_{\mathsf{compute\_intersection(r,s)}}(R \bowtie_{\mathsf{intersects(r,s)}} S)$

Preprocessing

5.	$R \bowtie_{\mathsf{distance(r,s)}<d} S$	$\rightarrow$	$\mathsf{buffer(R, d)} \bowtie_{\mathsf{intersects(r,s)}} S$
6.	$R \bowtie_{\mathsf{northwest(r,s)}} S$	$\rightarrow$	$\mathsf{northwest(R)} \bowtie_{\mathsf{intersects(r,s)}} S$
7.	$R \bowtie_{\mathsf{nearest\_neighbor(r,s)}} S$	$\rightarrow$	$\mathsf{min(buffer(R, d))} \bowtie_{\mathsf{intersects(r,s)}} S$

In the above table σ denotes the well-known selection operator and μ is the tuple constructor operator. It should be pointed out that the above transformations are independent of the actual representation of the geometry.
The remainder of this paper is organized as follows. In Section 2 we give a short introduction to z-ordering. In Section 3 we develop a more formal description of the underlying z-value calculus and section 4 shows that it can be used for processing and optimizing the most common spatial queries by using a minimal set of essentially four ϕ-functions. In Section 5, we illustrate the overall architecture and show how the technique of query rewriting can be applied to achieve the presented concepts.

2. Z-ORDERING

Although there is a large number of spatial access indexing techniques [Gü88, Kol90], most of these techniques cannot be easily integrated into a commercial database system. This paper, however, shows that the *z-ordering* scheme (see, for example, [OM88]) has the potential of being implemented on top of a commercial database system. One reason for this is that the z-ordering is a logical access method and not a physical one, even if physical clustering is beneficial and can be exploited by ObjectStore.
Z-ordering assumes a recursive decomposition of the plane into a hierarchical configuration of rectangular areas known as *z-regions*. These z-regions can be identified canonically by binary code strings known as *z-values* consisting of ones ('1') and zeros ('0'). If the z-value of a given region prefixes an other z-zvalue, then the former region encloses the latter; for example, `00` encloses `001`. Figure 1 shows some z-regions and their associated z-values. Any spatial object can be approximated by a set of z-regions which form the lowest upper bound to the spatial extension of the

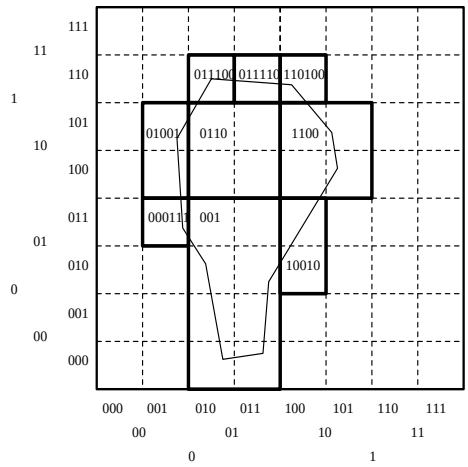

Figure 1: Decomposition of a given object using the z-value approximation.

object.[3]
Since these z-regions are uniquely identified by their associated z-values, the spatial extension of any spatial object can be approximated by a set of z-values.

$$Z = \{z_1, z_2, ..., z_n\} \quad (n \text{ finite})$$

The basic idea in the work presented here is to compute results of spatial queries (particularly of the intersection join) on the basis of z-value sets. As we will show in the following section, the computation of spatial queries using z-values can be performed by means of simple (boolean) operations rather than by computationally expensive geometric operations. Like most spatial access methods, however, the result of a query using z-values is only a set of candidate objects, which must be postprocessed using the exact geometries, to eventually find those objects that really satisfy the inquiry.

3. FORMAL MODEL OF QUERY PROCESSING USING Z-ORDERING

There are two possible views to describe a spatial database using z-ordering:

1. The z-value centered view: each existing z-value is bound to one or more objects.

2. The object centered view: each object is composed of a set of z-values which are stored as a set-valued attribute of the object.

Note that these two perspectives are totally different from the viewpoint of query processing. In the first case, a query using z-values would scan the z-value extent (a collection of all existing z-values) for matching z-values and then search objects using these z-values. In the second case, one has to retrieve first the objects of the (geo-) object extent (Ω) and then scan their extent for matching z-values. For our discussion we will use the latter model as a basis.
In the remainder, we will call the set of all (true) prefixes of the given z-value the

[3] As can be seen from this introduction, the z-ordering scheme can easily be extended to handle n-dimensional objects.

upper hull and all the z-values which are prefixed by a given z-value, the *lower hull*. Note that the region R corresponding to the z-value set Z_R is enclosed by all regions corresponding to z-values in the upper hull. Conversely, R *encloses* all regions corresponding to z-values in the lower hull.
In order to develop a more formal model of query processing using z-values, we introduce the following terminology.

- z_i: a sequence of values from $\{0, 1\}$
- $|z_i|$: length of z_i, i.e., the number of digits in the $0, 1$- sequence
- Z: a non-empty set of z-values
- $g = max(|z_i|)$: a measure for the granularity of z-values in the database
- $\mu_i(z_k)$: cutting operator, cuts the given z-value to the length i, i.e., leading digits are preserved up to the length i
- $\epsilon_i(z_k)$: extension operator, generates all possible extensions up to a given length i
- $\underline{Z} = \{\epsilon_k(z_i) || z_i| \leq k \leq g \ \& \ z_i \in Z\}$: lower hull
- $\overline{Z} = \{\mu_k(z_i) | 1 \leq k \leq |z_i| \ \& \ z_i \in Z\}$: upper hull
- $Z^+ = \underline{Z} \cup \overline{Z}$: hull closure
- Ω: extent of existing (geo-) objects
- $o_i.Z$: z-value set of object o_i

To evaluate the submitted queries, we have to define how they can be computed in the domain of z-values. For this purpose, we introduce a minimal set of essentially four different ϕ functions. A more precise definition of the ϕ-function is given in [GG94]. For our purpose it suffices to know, that each of ϕ-functions ϕ_c transforms a set of z-values Z into another set of z-values Z'. More formally:

$$\phi_c : Z \xrightarrow{c} Z'$$

where c denotes a certain constraint defined on this function.

1. $\phi_{\text{buffer(d)}}(Z)$: generate, for a given set Z, a z-value buffer of distance d; for convenience we assume that distances with respect to z-regions are measured in units of the grid size of the underlying cell structure
2. $\phi_{\text{adjacent}}(Z) = \phi_{\text{buffer(1)}}(Z)$: generate, for a given set of z-values Z, all possible values of adjacent cells (the distance 1 stands for the grid quantum)
3. $\phi_\gamma(Z)$: $\gamma \in \{\text{north}, \text{south}, \text{east}, \text{west}\}$: generate all z-values situated in the direction γ.
4. $\underline{\phi}(Z) = \underline{Z}$: compute the lower hull

5. $\overline{\phi}(Z) = \overline{Z}$: compute the upper hull

6. $\phi^+(Z) = \underline{Z} \cup \overline{Z}$: carry out the hull closure for the given set; for brevity, will often write Z^+ instead of $\phi^+(Z)$

It should be pointed out that these functions are defined on the level of z-values and not on the exact object geometries. Hence, all operations can be carried out very fast.
Examples:

1. $\underline{\phi}$: given $Z = \{001, 01\} \Longrightarrow \underline{\phi}(Z) = \{001*, 01*\}$,
   that is, generate all z-values having the same prefix[4]

2. $\overline{\phi}$: given $Z = \{001, 01\} \Longrightarrow \overline{\phi}(Z) = \{001, 01, 00, 0\}$,
   that is, apply the cutting operator μ_i successively to the given set Z.

3. $\phi_{\text{buffer(d)}}$: The effect of the buffer operator is depicted in figure 2.

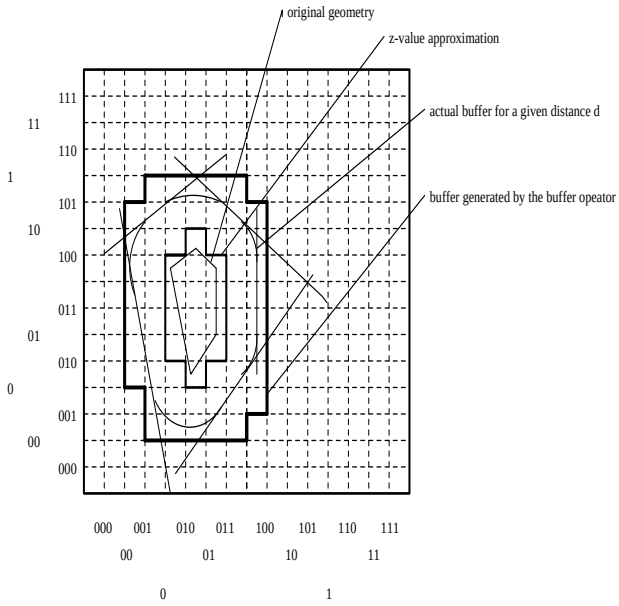

Figure 2: For the given geo-object and its z-value extension a buffer of distance d is build. Both, the exact buffer and its z-value buffer a depicted. Note, however, that the z-value buffer is build on basis of the original z-value set.

After introducing these fundamental functions, we can define the most important operations and equivalences to be used in the sequel:

1. intersection:

$$\begin{aligned} \mathsf{intersects}(Z', Z'') &\iff \underline{Z'} \cap \underline{Z''} \neq \emptyset \\ &\iff \{\exists z' \in Z' : z' = \mu_{|z'|}(z'') \ \&\ z'' \in Z''\} \vee \\ &\quad\ \{\exists z'' \in Z'' : z'' = \mu_{|z''|}(z') \ \&\ z' \in Z'\} \end{aligned}$$

[4] We will elaborate on how to represent these values in terms of intervals in section 4.1.

$$\iff \exists z' \in Z' : z' \in Z''^{+} \iff Z' \cap Z''^{+} \neq \emptyset$$
$$\iff \exists z'' \in Z'' : z'' \in Z'^{+} \iff Z'' \cap Z'^{+} \neq \emptyset$$

For efficiency reasons, the equivalence of the last two lines very important, since the complexity depends on the cardinality of the participating sets $Z'(Z'')$. Due to the above equivalences, the spatial intersection (for z-values) can be reduced to a simple enclosure test for one given value at best.

2. is_enclosed_by:

$$\begin{aligned} \mathsf{is\_enclosed\_by}(Z', Z'') &\iff Z' \subseteq \underline{Z''} \\ &\iff \forall z' \in Z' : z' \in \underline{Z''} \end{aligned}$$

In the above equation all elements are tested for enclosure in $\underline{Z''}$. A slight variation of the above query reveals a way how this test can be done in a more efficient way.

$$\nexists z' \in Z' : z' \notin \underline{Z''}$$

Instead of testing all elements of z' for enclosure in $\underline{Z''}$, the negation of the above equation can be tested more efficiently, i.e., it is sufficient to find one element of Z' not in $\underline{Z''}$ to negate the assumption of enclosure. Informally stated:

$$\neg\mathsf{is\_enclosed\_by}(Z', Z'') \iff \exists z' \in Z' : z' \notin \underline{Z''}$$

3. contains: $\underline{Z'} \supseteq Z''$

$$\mathsf{contains}(Z', Z'') = \mathsf{is\_enclosed\_by}(Z'', Z')$$

This equivalence enables the optimizer to reduce the contains test to an is_enclosed_by test or vice versa.

4. QUERIES IN DETAIL

In the next few sections, we will argue different kinds of queries and discuss their implementation.

1. Point Query
   Given a point p, find all objects enclosing this point.
   The given point p is represented by its z-value z_p. Therefore the query asking for all objects containing this point would be

$$Q_{\text{point}} = \{o | z_p \in \underline{o.Z} \ \&\ o \in \Omega\}$$

Processing this query would be inefficient since for each object o in the database one would have to compute the lower hull. A more efficient way is:

$$Q_{\text{point}} = \{o | \exists z \in \overline{z_p} : z \in o.Z \ \&\ o \in \Omega\}$$

This means that it suffices to find one z-value in the object extent which is included in the upper hull of the z_p.

2. Window Query
   Given a window w, find all objects o intersecting this window. The given window w will be represented by its z-values Z_w. This is in general a set of z-values, since the given window is only congruent in exceptional cases with the underlying grid structure. To find all objects, which are enclosed by this window, we have to carry out a hull closure, i.e., generate all interesting z-values.

$$\begin{aligned} Q_{\text{window}} &= \{o | Z_w^+ \cap o.Z \neq \emptyset \ \& \ o \in \Omega\} \\ &= \{o | \exists z \in Z_w^+ : z \in o.Z \ \& \ o \in \Omega\} \end{aligned}$$

3. Region Query
   Given a region of arbitrary shape r, find all objects intersecting this region. In contrast to the window query, where the search region has rectangular shape, the region query encompasses more general shapes. However, from the view of query processing using z-values the region query can be handled analogously to the window query, i.e., for a given region r and its corresponding set of z-values Z_r, the query could be expressed in the following way:

$$\begin{aligned} Q_{\text{region}} &= \{o | Z_r^+ \cap o.Z \neq \emptyset \ \& \ o \in \Omega\} \\ &= \{o | \exists z \in Z_r^+ : z \in o.Z \ \& \ o \in \Omega\} \end{aligned}$$

4. Intersection Query
   Given an object o_k (or a set), find all pairs of objects (o, o_k) intersecting each other:

$$Q_{\text{intersection}} = \{(o, o_k) | \exists z \in o_k.Z : z \in \underline{o.Z} \ \& \ o \in \Omega\}$$

   Note that is not possible to express such a query using the ObjectStore query interface, since ObjectStore does not offer a tuple constructor. A more relaxed version of the above query which can be answered using the ObjectStore query facility would be

$$Q_{\text{intersection}} = \{o | \exists z \in o.Z : z \in (o_k.Z)^+ \ \& \ o \in \Omega\}$$

5. Enclosure Query
   Given an object o_k, find all objects enclosing this object completely.

$$Q_{\text{enclosure}} = \{o | \forall z \in o_k.Z : z \in \underline{o.Z} \ \& \ o \in \Omega\}$$

   This query would be again inefficient since for all (geo-) objects of extent Ω the lower hull would be computed. More efficient variations are:

$$Q_{\text{enclosure}} = \{o | \forall z \in o_k.Z, \exists i \geq |z| : \mu_i(z) \in o.Z \ \& \ o \in \Omega\}$$

   and

$$Q_{\text{enclosure}} = \{o | \forall z \in o_k.Z, \exists o.z \in \overline{z} \ \& \ o \in \Omega\}$$

6. Containment Query
   Given an object o_k, find all objects enclosed by o_k.

$$Q_{\text{containment}} = \{o | \forall z \in o.Z : z \in \underline{o_k.Z} \ \& \ o \in \Omega\}$$

7. Distance Query
   Given an object o_k, find all objects within a distance d to o_k.

$$Q_{\text{distance}} = \{o | \exists z \in (\phi_{\text{buffer(d)}}(o_k.Z))^+ : z \in o.Z \ \& \ o \in \Omega\}$$

8. Neighborhood-Query
   Find all neighbors of a given object o_k.

$$Q_{\text{neighbor}} = \{o | \exists z \in (\phi_{\text{buffer(1)}}(o_k.Z))^+ : z \in o.Z \ \& \ o \in \Omega\}$$

9. Northwest-Query
   Find all objects situated in the northwest direction of a given object o_k. This query can be expressed by combining two of the introduced ϕ-functions.

$$Q_{\text{northwest}} = \{o | \exists z \in ((\phi_{\text{north}}\phi_{\text{west}})(o_k.Z))^+ : z \in o.Z \ \& \ o \in \Omega\}$$

10. Nearest (Farthest) Object(s)
    This kind of query can only be expressed in an unsatisfactory way, since one has to find first the nearest (farthest) distance before the hull closure can be conducted.

$$Q_{\text{nearest\_object}} = \{o | \exists z \in (\phi_{\text{min(buffer(d))}}(o_k.Z))^+ : z \in o.Z \ \& \ o \in \Omega\}$$

$$Q_{\text{farthest\_object}} = \{o | \exists z \in (\phi_{\text{max(buffer(d))}}(o_k.Z))^+ : z \in o.Z \ \& \ o \in \Omega\}$$

As the reader may have noticed, the first three queries could have been subsumed under the intersection query, although the given formulation of the intersection query is geared toward existing objects.

4.1. Representation of Z-values

To handle the z-values in the context of the database systems efficiently, one has to find a data structure that can be manipulated easily and efficiently. To do this, we found that transforming the tuple (z-value, z-value-length) to an unsigned integer is the most promising way. One possible transformation is

$$z' \ = \ b \cdot n + l$$

with

- z': new z-value;
- b: a zero-padded binary representation of the z-value;

- n: arbitrary number, which has to be at least greater than a certain minimum;
- l: number of valid digits, i.e., the z-value length.

The above formula transforms the tuple (z-value, z-value-length) to one unique `integer`-value. This transformation is not dense, but it is topology preserving (in terms of z-ordering), which means that region queries can be expressed in the most natural way.
For example, for the granularity of 6 the z-value `001` is padded to `001000 = 8`. The length of the given z-values is 3 and thus one gets $z' = 8 \cdot n + 3$ for some given n. By choosing n greater than a certain number, it is possible to extract easily the length of the z-value. To guarantee a fast transformation, n should preferably be chosen to the basis 2.

5. QUERY REWRITING

5.1. ObjectStore Query Interface

The syntax of ObjectStore queries[5] is quite simple if one assumes only (very) simple operations. The general structure of a query is `any_extent[: int_expression :]`, whereby we assume in the sequel, that each class has an extent consisting of all objects of this specific class. This query returns all the objects satisfying the `int_expression`, i.e., the `int_expression` is unequal to 0. Existential queries can be expressed by enclosing the "where clause" into `[% %]` brackets. This query returns only one object of the extent satisfying the given predicate. If none is found, an exception is raised.
It should be mentioned that it is not possible to express a join between two extents in a declarative fashion. Possible work arounds prohibit query optimization. A more detailed description of the query interface and query optimization can be found in [OHMS92].

5.2. Optimizing Queries Containing Spatial Predicates

Optimizing spatial queries is somewhat different to "conventional" query optimization since queries frequently consist of a spatial and a non-spatial (thematic) part whereby the spatial part normally includes expensive predicates in terms of computational cost. Our query interface, which is built on top of the ObjectStore query interface, elaborates only on the spatial part, that is, rewrites and optimizes the spatial part using z-values. This rewritten query is submitted together with the non-spatial part to the ObjectStore query optimizer. The overall architecture is depicted in figure 3. Generally, all possible queries containing spatial predicates and taking advantage of z-ordering have to undergo the following transition:

$$\text{GEO-QUERY}(...) \rightarrow \text{z-value-condition(s)} + \text{geo-query}(...)$$

[5] We will restrict our explanations, however, to the program interface since they apply to the ad-hoc interface with some restrictions too.

This statement illustrates that the original query is transformed to an augmented query, whereby the first selection is performed using the z-value condition (filter step), generated in the process of rewriting the query. All objects which have passed this z-value test successfully form a set of candidates, which has to be postprocessed by applying the remaining predicates. Postprocessing is necessary since the final evaluation must be done using the exact geometries.
As can be seen from figure 3 it is necessary to hold some additional information to evaluate the submitted query. To avoid, for example, the superfluous generation of not existing z-values while applying the ϕ-functions, it is beneficial to keep track of existing z-values. One further needs the ObjectStore query interface or the meta-object protocol to acquire further information.

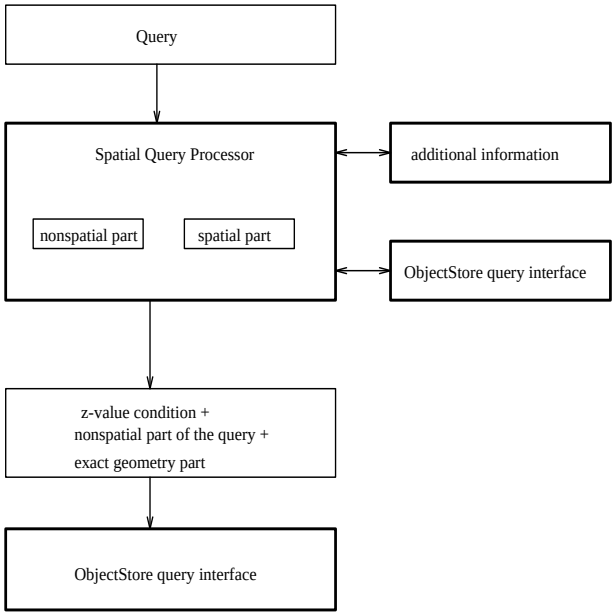

Figure 3: Architecture of the Query Processor

5.3. Example: Intersection Query

Assume that we are interested in finding the place where the television tower in East-Berlin (`tvteb`) is located. The tower is represented in our example by the z-value ``1000'' [6] (extended object, i.e., not point). Let us further assume that the maximum length (granularity) of the z-values is six ($g = 6$) and that there exists an extent of z-values, whereby each z-value can occur multiple times and each existing z-value is at least associated with one object.
Hence, the intersection query would be expressed in the following way:

```
Extended_Object * tvteb;
extent[: INTERSECTION(tvteb) :];
```

By using

$$Q_{\text{intersection}} = \{o | \exists z \in o.Z : z \in (o_k.Z)^+ \ \& \ o \in \Omega\}$$

[6] We give the z-values using their `0,1`-representation since we found it more instructive.

the query is tranformed to [7]

```
extent[: (zvalue == "1000*" || zvalue == "100" || zvalue == "10" ||
        zvalue == "1") && intersection(geometry, tvteb->geometry) :];
```

Assuming the transformation given in section 4.1, the intersection query can be further transformed to ($n = 5$) intervals and integers. The final final query would be:

```
extent[: (161 >= zvalue && zvalue <= 181)
                && intersection(geometry, tvteb->geometry) :];
```

This query returns the the place we are looking for: the Alexanderplatz.
As can be seen from the above query it is not necessary for all elements to test explicitly if they are enclosed in the (discrete) set of z-values. The transformation given above enables one to regroup the z-value set and identify intervals. From the viewpoint of query processing, this test can be performed much more faster. It is further noteworthy, that this query can be processed in at least an index-supported manner by creating indices on z-values.

6. CONCLUSION AND FUTURE WORK

In this paper we have shown that the technique of z-ordering is general enough to be applied in combination with (object-oriented) database systems. One gets the most out of the offered database functionality without substantial loss of performance compared to proprietary systems.
It should be noted that the z-ordering approach described in this paper does not rely on precomputed relationships between spatial objects. Whenever a new spatial object is entered into the database, it is not necessary to determine and to establish topological relations to all its neighboring objects. It is sufficient to compute the associated z-value set. Existing objects may be disregarded while inserting a new spatial object into the database. In particular, this makes it easy to compute spatial joins between objects coming from different sources.
Furthermore, it has been demonstrated that the concept of the ϕ-function can be applied in this setting to enable the optimization and processing of user-defined functions in at least an index-supported way. Hence, the combination of z-ordering and ϕ-function seems very useful, since one can use spatial indexing and user-defined functions to compute spatial joins in a straightforward way with a (hopefully) substantial performance gain. Although we can not report performance results at the moment, first tests are promising.
Current and future work on the optimizer includes the integration of not yet implemented query functionality and further optimization.

ACKNOWLEDGEMENT

We gratefully acknowledge the interesting discussions with Oliver Günther and his careful revision of drafts of this paper.

[7] Writing ``1000*'' is not allowed in ObjectStore and is only used here to express the prefix feature of the z-values, i.e., all intersecting z-regions share the same prefix. The star is used as a wildcard for zero or more digits following the prefixing sequence of digits.

REFERENCES

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proc. 9th Int. Conf. on Data Engineering*, pages 40–49, 1993.

[Ege94] Max J. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions of Knowledge and Data Engineering*, 6(1), February 1994.

[GG94] Volker Gaede and Oliver Günther. Processing joins with user-defined functions. Technical Report 94-013, ICSI, Berkeley, California, March 1994.

[GL94] Oliver Günther and Johannes Lamberts. Object-oriented techniques for the management of geographic and environmental data. *The computer journal*, 37(1), 1994.

[GR93] Oliver Günther and Wolf-Fritz Riekert. The design of GODOT: An object-oriented geographic information system. In *IEEE Data Engineering Bulletin*, September 1993.

[Gü88] Oliver Günther. *Efficient Structures for Geometric Data Management*. Springer-Verlag, 1988.

[Gü93] Oliver Günther. Efficient computation of spatial joins. In *Proc. 9th Int. Conf. on Data Engineering*, 1993.

[Kol90] Curt P. Kolovson. *Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems*. PhD thesis, University of California at Berkeley, 1990.

[LLOW91] Charles Lamb, G. Landis, Jack Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 10(34), October 1991.

[Mol92] Martien Molenaar. Formal data structures and query spaces. In *Konzeption und Einsatz von Umweltinformationssystemen*. Springer-Verlag, 1992.

[OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. In *Proc. of the 1993 ACM Int. Conf. on the Management of Data, SIGMOD Record*, volume 22, pages 403–412, June 1992.

[OM88] Jack A. Orenstein and Frank Manola. Probe: Spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14:611–629, May 1988.

[SV92] Michel Scholl and Agnès Voisard. Object oriented database system for geographic applications: An experiment with O_2. In *The O_2 BOOK*, pages 585–618. Morgan Kaufmann, San Mateo, California, 1992.