Optimization Strategies for Spatial Query Processing*

Walid G. Aref
Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742

Abstract

The application of standard query processing and optimization techniques in the context of an integrated spatial database environment is discussed. In addition, some new processing and optimization strategies are shown to emerge from the nature of the underlying architecture used for the integration of spatial data. Other strategies are presented that are application-dependent. They are related to the different possible implementations of spatial operators where each one is preferrable under certain conditions. The underlying spatial database architecture that is used is called SAND (denoting Spatial And Non-spatial Data). SAND is a dual spatial database architecture in which the objects' spatial information is stored in separate spatial data structures and their non-spatial information is stored in database relations while maintaining appropriate links between the spatial and non-spatial components of each object. SAND provides an equal opportunity for both the spatial and non-spatial components of the data to participate in query processing and optimization. Aside from the application-dependent optimization strategies discussed in the paper, these techniques are not limited to spatial data. They can be extended to deal with multi-media databases as well.

1 Introduction

Today, many non-standard database applications rely heavily on spatial data. This has resulted in considerable research efforts towards supporting spatial objects in database environments. Supporting spatial objects and, more generally, complex objects, involves extending almost all levels of the database management system (DBMS). At the user interface level, the query language is extended to enable the definition and manipulation of complex objects while maintaining the non-procedural flavor of the language. At the physical and access method levels, alternative ways of storing, organizing, and accessing complex objects have to be added to the DBMS for efficient handling of complex objects. At the intermediate level, new ways for mapping the extended user query into the extended physical and access method level are necessary. Most of the extensions at this level are made to the query optimizer and to the query processor. In this paper, we concentrate on spatial query processing and optimization.

The query optimizer is an important component of a modern DBMS. Usually, the user's query, expressed in a non-procedural language, describes only the conditions that the final response must satisfy. It is the optimizer's responsibility to generate a query evaluation plan that computes the requested result efficiently. Many different strategies have been proposed for finding a query evaluation plan which evaluates a submitted query in a traditional database environment (see [Jarke and Koch, 1984] for a comprehensive overview of various query optimization techniques). However, when it comes to non-standard applications, query optimization techniques vary significantly from one application domain to the other.

In the past, very little attention was devoted to spatial query processing and optimization. Existing spatial database systems have basically ignored the optimization issues. Little is known about the different strategies and alternatives for processing spatial queries (i.e., when interspersing spatial and non-spatial operations) as well as about the cost of executing alternative query evaluation plans containing spatial operations.

It is interesting to observe that, at a first glance, query optimization techniques for heterogeneous

*This work was supported in part by the National Science Foundation under Grant IRI-9017393 and Hughes Research Laboratory.

database systems (e.g., [Dayal, 1984]) appears to be applicable to spatial query optimization. Unfortunately, query optimization of spatial data is different from that of heterogeneous databases because of the cost function. In particular, the emphasis in heterogeneous databases is on inter-database communication costs. Databases in a heterogeneous system are usually distributed over a long-haul network where the network communication cost dominates. Thus the goal of a query optimizer is to do as much processing as possible at each participating site thereby reducing the inter-database communication overhead [Dayal, 1984]. We assume that the spatial and non-spatial data are stored in the same database which is located in one node of a network.

A number of recently suggested spatial database architectures address the issue of spatial query optimization (e.g., [Güting, 1989; Orenstein, 1990; Sacks-Davis, et. al., 1987; Wolf, 1989]). However, they differ in the capabilities and degrees of freedom they provide to the spatial query optimizer as a result of the manner in which they integrate spatial data with non-spatial data. In particular, the underlying architecture may limit some feasible strategies for spatial query processing. For example, according to one spatial database architecture (e.g., GEO-Kernel [Wolf, 1989]), the query processor may be forced to perform the non-spatial operations first and then the spatial operations. A common feature of the aforementioned systems is that their architectures are biased towards the non-spatial aspect of the data. At best, they restrict the possible choices for the spatial optimizer, thereby making only a limited use of the spatial aspect of the data (i.e., its extent) in the optimization of mixed queries.

GEOQL's spatial query optimizer [Ooi, 1988; Sacks-Davis, et. al., 1987] extends the well-known query decomposition technique [Wong and Youssefi, 1976] to handle spatial queries as well. However, GEOQL is also biased towards the relational side. For example, spatial operations cannot be composed directly without building intermediate database relations. This limits the efficiency of spatial query processing. Moreover, GEOQL's extended optimizer only optimizes the cost of non-spatial operations and does not take into account the I/O cost of spatial operations.

Extensible database systems offer a different approach for query processing and optimization. One of the main design goals of extensible systems is to be as application-independent and as general as possible in order to support a wider range of non-standard database applications. To meet these requirements, extensible systems mostly use rules for describing query transformations as well as for choosing among different implementations of primitive database and application-dependent operations for query processing and optimization. Some examples of such systems are the EXODUS optimizer generator [Graefe and DeWitt, 1987] and Starburst's query processor [Haas, et. al., 1989].

GEO-Kernel [Wolf, 1989] and Gral [Güting, 1989] are two spatial database systems that are based on an extensible architecture. GEO-Kernel implements a geometric data model on top of the DASDBS kernel system [Schek and Waterfeld, 1986; Wolf, 1989] that supports Non-First-Normal-Form (NF^2) relations [Schek and Scholl, 1989]. Gral extends the relational model by geometric operations. Gral also features a rule-based optimizer and query processor [Becker and Güting, 1989; Güting, 1989]. In both GEO-Kernel and Gral, spatial information is stored in textual form as an attribute value in a relation. Nevertheless, during query evaluation appropriate spatial data structures are used to operate on spatial data. Thus there is a need for conversion procedures to toggle between these data structures and the textual or byte-string form for each spatial data type. Notice that in order to perform the operation *intersects* or *closest*, for example, the whole set of spatial objects in the relevant relations have to be down-loaded into the spatial data structures. This is an expensive task and its cost has to be included when considering different query evaluation plans.

The architecture of our underlying spatial database system, SAND (denoting Spatial And Non-spatial Data) [Aref and Samet, 1990; Aref and Samet, 1991], differs from the above systems in the manner in which spatial and non-spatial data are stored and linked to each other. Also, SAND assumes that the spatial description of objects is stored in disk-based spatial data structures which are linked properly to the rest of the objects' non-spatial description. For more details about the SAND architecture, as well as a more thorough comparison between different spatial database architectures, see [Aref and Samet, 1991].

In this paper, we present a variety of feasible strategies for answering spatial and mixed queries in the SAND spatial database environment. SAND is unbiased with respect to the spatial and non-spatial aspects of the data so that the contribution of each aspect in the optimization process is maximized. By providing a variety of strategies for answering spatial queries, the query optimizer will have more alternatives to choose from thereby enabling more efficient execution of spatial queries. Another motivation for this work is that the spatial query processing strategies presented in this paper, as well as the SAND spatial database architecture, can be ported on top of existing extensible database systems as a validation step for both the SAND and the extensible system architectures.

The rest of the paper is organized as follows. Section 2 summarizes SAND, the underlying spatial database architecture that is used throughout the paper. Alternative query processing strategies for use in SAND are presented in Section 3. These include strategies for processing relational and spatial selection and join operations. Section 4 contains concluding remarks.

2 An Overview of SAND: A Spatial Database Architecture

In [Aref and Samet, 1991] we present the SAND spatial database architecture in much more detail. Below, we give a brief overview of some of the features of the SAND architecture that are used in our presentation. For further details about the SAND system, see [Aref and Samet, 1990; Aref and Samet, 1991].

Throughout the paper we will often refer to the schema definitions given in Figures 1 and 2 which define the `land-use` and `roads` spatial databases. Notice that `location` is a spatial attribute of type `REGION` while `road_coords` is a spatial attribute of type `LINE_SEGMENT`. We use an SQL-like syntax.

```
(create table land-use
 name  CHAR[40],
 address CHAR[100],
 location REGION,
 usage CHAR[40],
 zip_code NUMBER,
 importance NUMBER);
```

Figure 1: Land-use spatial database schema

```
(create table roads
 road_id NUMBER,
 road_name CHAR(30),
 road_trafficability NUMBER,
 road_lanes NUMBER,
 road_coords LINE_SEGMENT);
```

Figure 2: Roads spatial database schema

In general, a spatial object is described by two sets of attributes: spatial and non-spatial. Objects that are spatially related to each other (i.e., are in proximity, or belong to a given region) and that are of the same data type such as line data, are logically clustered in the database (e.g., stored together in database relations or in suitable spatial data structures).

A spatial data structure is associated with each spatial attribute in the schema and is used to store all data instances of that spatial attribute over the set of homogeneous objects. The spatial data structure is used as an index for spatial objects as well as a medium for performing spatially-related operations (e.g., rotation and scaling for images, editing, buffer-zoning, polygon intersection, area of a region, connected component retrieval, proximity queries, etc.). Depending on the attribute's spatial data type (e.g., region, line, or point), a spatial data structure suitable for handling this type is selected.

The data instances of the set of non-spatial attributes are stored in database relations. Each tuple in the relation corresponds to one object. Figure 3 illustrates how we link spatial and non-spatial attribute values of an object. In particular, we maintain two logical links between the spatial and non-spatial data instances of an object: *forward* and *backward* links. Forward links are used to retrieve the spatial information of an object given the object's non-spatial information. On the other hand, backward links are used to retrieve the non-spatial information of an object given the object's spatial information. Since the non-spatial information of an object is stored in a tuple, the backward link can be the tuple-id. Since spatial data structures hold the spatial information of an object and spatially index the object space, the forward link can be a spatial index value of this object inside the data structure that uniquely selects the object. One example of a forward link is a candidate point inside a region to uniquely select a regional object in an object space consisting of non-overlapping regions.

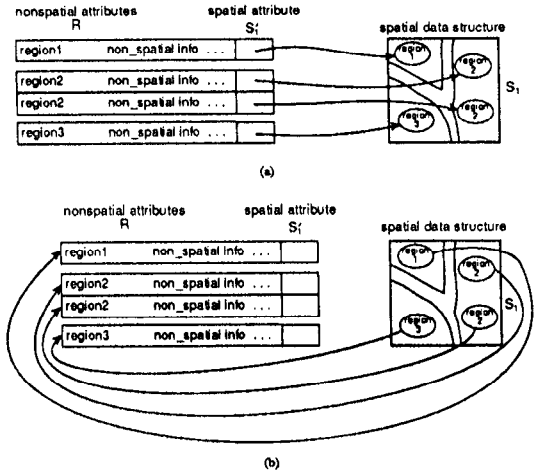

Figure 3: (a) Forward links and (b) backward links for the land-use database

Maintaining forward and backward links between the spatial and non-spatial aspects of a set of objects facilitates browsing in the two parts as well as permits efficient query processing. Flexibility in the interaction

between spatial and non-spatial attributes enable operations (whether spatial or non-spatial) to be performed in their most natural environment. In [Aref and Samet, 1990] we showed how forward and backward links facilitate query processing and query optimization. This is also demonstrated here in Section 3.

We assume that we have a collection of objects O referred to by the pair $< R, S >$, where R is a relation that stores the non-spatial attribute instances of O and S is a spatial data structure that stores the spatial attribute instances of O. We assume that the set O is described by only one spatial attribute and hence has one spatial data structure associated with it. Relaxing this assumption is straightforward. We use the notation $op(U)$ where U is either R or S or the pair $< R, S >$ (depending on the context) to indicate that operation op is applied to U.

One important requirement of the SAND architecture is that there needs to be a mechanism for keeping the spatial component in sync with its non-spatial component. Spatial, as well as relational, operators must always keep the pair $< R, S >$ consistent. For example, consider a certain spatial operator op_s that operates on the spatial data structure S to produce another spatial data structure S_1 (e.g., a window operator generates a subset of the input data structure). In order for the selected objects, say O_1, in S_1 to be fully described, O_1's non-spatial information has to be selected as well from relation R to form a new relation R_1 which is also a subset of R and represents the objects resulting from applying the spatial operator op_s. In summary, given the pair $< R, S >$, op_s returns the pair $< R_1, S_1 >$ instead of just S_1. This reasoning is also applicable to relational operators, say op_r, such as *selection*.

In addition to the above synchronization requirement, operating on multiple spatial attributes (and hence multiple spatial data structures) and querying on the various relationships between spatial objects introduces the concept of *spatial joins*. Consider the following query, which retrieves all the regions within 5 miles from universities in the land-use database:

```
select all
  from land-use l, land-use k
 where within(l.location,k.location,5)
       and l.usage = "University"
```

There may exist more than one university in the database, and hence more than one tuple could be selected by the condition `l.usage = "University"`. Let the set of selected tuples be L. Then the spatial `within` condition generates the regions within 5 miles from each member of L. The result of the `within` condition should consist of a join relation. In particular, two tuples are merged if their corresponding spatial objects are within 5 miles of each other. The resulting relation contains all the attributes of the two participating relations, including the two spatial attributes (i.e., the `location` attribute) and their corresponding spatial data structures. There is one spatial data structure for the university regions and another for the neighboring regions (i.e., the ones within 5 miles) because the join is performed on two spatial attributes. We refer to the `within` condition as a *spatial join*. This is because it has the same effect as a regular join. Namely, a spatial join combines related entities from two entity sets into single entities whenever the combination satisfies the spatial join condition (e.g., if they are within n miles from each other). Table 1 lists some spatial join conditions. Notice the use of s_i to specify instance values of spatial attributes.

Table 1: Some Spatial Join Conditions

Spatial Join	Description
adjacent_to	True if s_i is an adjacent neighbor of s_j
contained	True if s_i is contained in s_j
within n	True if s_i is within n units of distance from s_j
intersect	True if s_i and s_j overlap

We realize the requirements of the SAND database architecture by defining what we call *extended operators*. In general, extended operators provide a proper interface for integrating not only spatial data but also any other multi-media data into a database environment.

There are two types of extended operators: relational-based operators x_op_r and spatial-based operators x_op_s (r denotes relational while s denotes spatial). We define these extended operators in terms of their un-extended counterparts and the operators sp_extract and db_extract described below. As an illustration, here we only discuss the extended *select* operation (i.e., both the spatial and relational variants). The reader is referred to [Aref and Samet, 1991] for the definition of the extended join.

The extended relational select operation $< T, U >= x\_op_r(< R, S >)$ first performs the relational select op_r on R. This results in the relational component $T = op_r(R)$. The spatial component U is built by executing the operator sp_extract to extract from S the spatial objects corresponding to the tuples in the resulting relation T. The extended spatial select x_op_s has an analogous description. It uses the operator db_extract to extract the tuples corresponding to the objects selected by the spatial operation op_s. More formally,

$$x\_op_r(< R, S >) = < op_r(R), \text{sp\_extract}(op_r(R), S) >, \text{ and}$$
$$x\_op_s(< R, S >) = < \text{db\_extract}(R, op_s(S)), op_s(S) > .$$

Describing spatial and relational operators using extended operators is very general. Depending on the context, we can replace this general form with simpler versions of the same operator and still get the same query answer. These simplified versions are mainly useful for query optimization. In fact, as we demonstrate in the Section 3, several optimizations can take place.

3 Query Processing Strategies

We demonstrate the strategies for spatial query processing by giving examples using the database schema definitions of Figures 1 and 2.

Example 1: Find all roads other than "Route 1" that pass through a given window w.

```
select all
  from roads
 where in_window(road_coords,w)
       and road_name != "Route 1"
```

Below, we give several strategies for processing this query as well as others.

Plan 1 - Un-optimized: Plan 1, given in Figure 4, uses the notion of extended operators without any further optimizations. We can rephrase Plan 1, as given

$$
\begin{aligned}
< T_1, S_1 > &\leftarrow x\_sp\_window(< R, S >, w) \\
< T_2, S_2 > &\leftarrow x\_db\_select(< T_1, S_1 >, db\_cond)
\end{aligned}
$$

Figure 4: Summary of Query Plan 1

in Figure 5. This helps to clarify the optimization steps demonstrated in the following plans. Notice that a re-

$$
\begin{aligned}
S_1 &\leftarrow sp\_window(S, w) \\
T_1 &\leftarrow db\_extract(R, S_1) \\
T_2 &\leftarrow db\_select(T_1, db\_cond) \\
S_2 &\leftarrow sp\_extract(S_1, T_2)
\end{aligned}
$$

Figure 5: Rephrasing of Query Plan 1

ordering of the operations is also possible as given in Figure 6. Other reorderings are given in Plan 2.

Plan 2 - Further reorderings: Figure 7 gives an alternative reordering of the operators in Plan 1. Here the database process and the spatial process each work independently on a different portion of the input data. The results are merged at a later step.

$$
\begin{aligned}
T_1' &\leftarrow db\_select(R, db\_cond) \\
S_1' &\leftarrow sp\_extract(S, T_1') \\
S_2' &\leftarrow sp\_window(S_1', w) \\
T_2' &\leftarrow db\_extract(T_1', S_2')
\end{aligned}
$$

Figure 6: Reordering of the operations of Query Plan 1

$$
\begin{aligned}
S_1 &\leftarrow sp\_window(S, w) \\
T_1 &\leftarrow db\_select(R, db\_cond) \\
< T_2, S_2 > &\leftarrow merge(T_1, S_1)
\end{aligned}
$$

Figure 7: Query Plan 2

The purpose of the merging step is to find the objects that exist in both the input spatial data structure, say S_1, and the input relation, say T_1, and generate an output pair, say $< T_2, S_2 >$, that contains all these common objects. Figure 8 illustrates the merging operation.

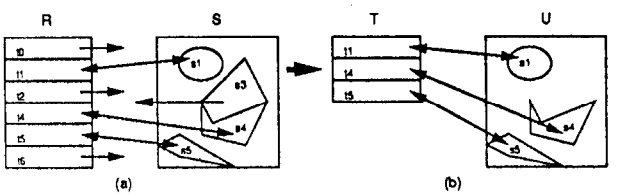

Figure 8: (a) Relation R and spatial data structure S to be merged, (b) relation T and spatial data structure U contain the result of merging R and S.

Basically, there are two ways of performing a merge: spatial-driven (*sp_merge*) and relational-driven (*db_merge*). sp_merge traverses the spatial data structure S and for each spatial object that it encounters, say o, sp_merge tries to retrieve o's corresponding tuple, say t, through the tuple-id stored with o. If t is found, then t and o are stored in T and U, respectively. Otherwise, o is not part of the result - i.e., it is skipped. A schematic listing of sp_merge is given in Figure 9. db_merge is the same as sp_merge except that the relation R is traversed and the spatial objects (if any) that correspond to the tuples in R are retrieved and stored into the output spatial data structure U. Tuples with no corresponding spatial objects are discarded, while tuples with matching spatial objects are stored into the output relation T. A schematic listing of db_merge is given in Figure 10.

Plan 3 - Intersection of pointers: A third method of merging the results of conjunctive selections, in addition to the spatial-based merging and relation-based merging described in Plan 2 above, is by intersection of pointers.

```
sp_merge(R,S)
/* Merge relation R with spatial data structure S
   based on the common objects in them.
   The results are stored in relation T and spatial
   data structure U. */
begin
  initialize relation T;
  initialize spatial data structure U;
  traverse S;
  for each spatial object o in S do
    begin
      tid := get o's tuple-id;
      if tid in R then
        begin
          retrieve tid's tuple t from R;
          append t into T;
          insert o into U;
        end;
    end;
end;
```

Figure 9: Spatial-driven merging

```
db_merge(R,S)
/* Merge relation R with spatial data structure S
   based on the common objects in them.
   The results are stored in relation T and spatial
   data structure U. */
begin
  initialize T, U;
  traverse  R;
  for each tuple t in R do
    begin
      sid := get t's spatial-id;
      if sid in S then
        begin
          retrieve sid's spatial object o from S;
          insert o into U;
          append t into T;
        end;
    end;
end;
```

Figure 10: Relation-driven merging

This is a well-known technique for answering conjunctive selections where the tuple-ids resulting from each selection are intersected [Elmasri and Navathe, 1989]. Intersecting pointers is possible if the selections that are performed generate tuple-ids. This situation can arise when the attributes that comprise the selection condition can be accessed via a secondary index that contains the associated tuple-ids.

In SAND we have two types of object-ids, namely tuple-ids and spatial-ids. In addition, the conjunctive selections may be spatial, relational, or both. Incorporating the intersection of pointers technique in the SAND environment can be done in two different ways, depending on whether we intersect the tuple-ids or the spatial-ids. This is illustrated by the following example.

Example 2: Find all 4-lane roads that are within `r` miles of point `(x,y)`.

```
select road_name
  from roads land-use
 where in_circle(road_coords,x,y,r)
       and road_lanes = 4
```

1 - Intersection of tuple-ids: If a secondary index is present on the attribute `road_lanes`, then when performing the selection based on `road_lanes` (i.e., `road_lanes = 4`) would generate a set of tuple-ids (we use tuple-ids as indexes so that we are able to intersect them with tuple-ids generated from the spatial side). The spatial selection `in_circle` generates the spatial objects that lie inside the specified circle and stores them in a temporary spatial data structure. To get the tuple-ids of the objects selected by `in_circle`, we need to traverse this data structure and collect the corresponding tuple-ids. These can be intersected with the ones generated from the database selection. The result of the intersection is then materialized both from the relational as well as the spatial side. Notice that the operation `in_circle` can generate the list of tuple-ids directly without the need of an extra traversal of the spatial data structure. Figure 11 gives the resulting plan. Operator *sp_in_circle_tid* is a simplified version of the operator *in_circle* which returns just the backward link information (tuple-ids) of the selected spatial objects. *db_select_tid* is a secondary index selection that returns the tuple-ids. Operation *list_intersect_tid* is given in Figure 12.

$$
\begin{aligned}
L_{sp} &\leftarrow sp\_in\_circle\_tid(S, c) \\
L_{db} &\leftarrow db\_select\_tid(R, db\_cond) \\
< T_2, S_2 > &\leftarrow list\_intersect\_tid(L_{db}, L_{sp}, R, S)
\end{aligned}
$$

Figure 11: Intersecting tuple-ids generated from both the spatial and relational selections. c denotes the coordinates of the circle and its radius.

2 - Intersection of spatial-ids: The same strategy can be applied when we consider intersecting spatial-ids instead of tuple-ids. Consider the plan given in Figure 13. Operation *sp_in_circle_sid(S,c)* is a simplified version of operation *sp_in_circle* which returns just the spatial-ids of the qualified objects (i.e., the ones lying inside the circle c). In order to return the spatial-ids as a result of the database selection (i.e., based on `road_lanes = 4`), we need to return the value of the spatial attribute for each qualifying tuple in the se-

```
list_intersect_tid(Lo,Lr,R,S)
/* Intersect lists Lo and Lr where each list
   contain tuple-ids and retrieve the tuples
   and spatial objects corresponding to the common
   tuple-ids. The results are stored in
   relation T and spatial data structure U. */
begin
  initialize T, U;
  I := intersect Lr and Lo;
  for each tuple-id tid in I do
    begin
      retrieve tid's tuple t from R;
      append t into T;
      sid := get t's spatial-id;
      retrieve sid's spatial object o from S;
      insert o into U;
    end;
end;
```

Figure 12: Conjunctive selection using intersection of tuple-ids. Tuple-ids in the intersection can be sorted for faster retrieval.

lection. This strategy is useful when the cost of retrieving the spatial description of objects is expected to be high in comparison to that of retrieving tuples from the database (e.g., when the volume of the spatial data is high in comparison to that of the non-spatial data). The operation *list_intersect_sid*(L_{db}, L_{sp}, R, S) intersects the two spatial-id lists resulting from the spatial and relational selections. Then, it retrieves the spatial and non-spatial description of the objects in the intersection. Operation *list_intersect_sid* is given in Figure 14.

$$
\begin{aligned}
L_{sp} &\leftarrow sp\_in\_circle\_sid(S, c) \\
L_{db} &\leftarrow db\_select\_sid(R, db\_cond) \\
< T_2, S_2 > &\leftarrow list\_intersect\_sid(L_{db}, L_{sp}, R, S)
\end{aligned}
$$

Figure 13: Intersecting spatial-ids generated from both the spatial and relational selections. c denotes the coordinates of the circle and its radius.

Plan 4 - Pushing spatial operations into sp_extract: Consider the plan given in Figure 6 to answer the query of Example 1. Spatial data structure S'_1 is written by operator sp_extract and then read by operator sp_window. To avoid an extra traversal of S'_1 as well as the read/write overhead, we can perform some spatial conditions on the fly along with operator sp_extract. This technique may be desirable under some but not all circumstances. For example, if the cardinality of the spatial objects is low and if the spatial test to be performed is relatively simple (e.g., if a line intersects a given window), then it is indeed more economical to perform this spatial test on the fly along with the sp_extract operator instead of storing the result and then retraversing the whole structure. Figure 15 gives

```
list_intersect_sid(Lo,Lr,R,S)
/* Intersect lists Lo and Lr where each list
   contain spatial-ids and retrieve the tuples
   and spatial objects corresponding to the common
   spatial-ids. The results are stored in
   relation T and spatial data structure U. */
begin
  initialize T, U;
  I := intersect Lr and Lo;
  for each spatial-id sid in I do
    begin
      retrieve sid's spatial object o from S;
      insert o into U;
      tid := get o's tuple-id;
      retrieve tid's tuple t from R;
      append t into T;
    end;
end;
```

Figure 14: Conjunctive selection using intersection of spatial-ids.

$$
\begin{aligned}
T'_1 &\leftarrow db\_select(R, db\_cond) \\
S'_2 &\leftarrow sp\_extract\_f(S, T'_1, in\_window(w)) \\
T'_2 &\leftarrow db\_extract(T'_1, S'_2)
\end{aligned}
$$

Figure 15: Plan 4: pushing window selection into operator *sp_extract_f*. The output is stored in the pair $< T'_2, S'_2 >$. Relation T'_1 is a temporary relation.

the resulting plan. Notice the use of the new operator *sp_extract_f* (f denotes filter) which has one additional argument over sp_extract. This argument serves as a spatial selection condition. All the spatial objects extracted should satisfy this condition. Also notice that Plan 4 uses only one temporary spatial data structure, namely S'_2, which is also the output data structure while Plan 1 uses two temporary data structures, namely S_1 and S_2, where S_2 is also the output data structure.

Plan 5 - Pushing database selection into db_extract: Consider the plan given in Figure 5 to answer the query of Example 1. The relation T_1 is written by operator db_extract and then read by operator db_select. To avoid an extra traversal of the temporary relations, we can perform the database selection at the same time that we extract the corresponding tuples. This is one form of the use of the pipelining technique to save from creating needless temporary relations and to save on traversal time. Figure 16 lists the modified plan. Notice the use of the new operator *db_extract_f*

$$
\begin{aligned}
S_1 &\leftarrow sp\_window(S, w) \\
T_2 &\leftarrow db\_extract\_f(R, S_1, db\_cond) \\
S_2 &\leftarrow sp\_extract(S_1, T_2)
\end{aligned}
$$

Figure 16: Plan 5: pushing database selection into *db_extract_f* operator. The output is stored in the pair $< T_2, S_2 >$. S_1 is a temporary spatial data structure.

(f denotes filter) which has one additional argument over db_extract. This argument serves as a relational selection condition. All the tuples extracted should satisfy this condition. Also notice that Plan 5 uses only one temporary relation, namely T_2, which is also the output relation while Plan 1 uses two temporary relations, namely T_1 and T_2, where T_2 is also the output relation.

Plan 6 - Further Pipelining: Notice that in Plans 4 and 5 we can get rid of the temporary relations and data structures T_1' and S_1 in Figures 15 and 16, respectively. This is achieved by directly piping the results of database selection in Plan 4 and the results of spatial windowing in Plan 5 into the next stage. Notice that some communication overhead is incurred due to the pipelining of data items between concurrent processes (the spatial process and the DBMS process), and due to the distinction between set-at-a-time or tuple-at-a-time communication. This has to be taken into consideration when deciding if this strategy is to be applied.

Plan 7 - Deletion instead of insertion: In some cases (e.g., in database selections or spatial operations with very high selectivity values) it is easier to delete the disqualified data items from the existing structure or relation than to create a new one with almost all the same data items that are in the input structure or relation except for a few missing data items. Of course, this depends on the relative cost of deletion versus insertion in conjunction with the value of the selectivity factor. Other factors are that in some cases deletion is not feasible. This is especially true when we are overwriting or destroying the input structures. It is mostly useful with intermediate temporary structures. Furthermore, the nature of the temporary structures used to speed the processing of queries may not allow deletions in the first place. Figure 17 gives one example modification of Plan 1 as given in Figure 5 where deletion is used instead of insertion in the last step. In this case S_2 is formed by removing from S_1 all of the spatial objects whose corresponding tuples are not in T_2. This is performed by the operation *sp_exclude*. S_1 no longer exists after executing *sp_exclude*.

$$
\begin{aligned}
S_1 &\leftarrow sp\_window(S, w) \\
T_1 &\leftarrow db\_extract(R, S_1) \\
T_2 &\leftarrow db\_select(T_1, db\_cond) \\
S_2 &\leftarrow sp\_exclude(S_1, T_2)
\end{aligned}
$$

Figure 17: Use of deletion instead of extraction when spatial selectivity is high.

Plan 8 - Performing projection as early as possible: This is a standard technique in query optimization. However, in our spatial database architecture there is more to it. When a spatial attribute is not part of the final answer, we can stop maintaining the spatial data structure associated with this attribute as early as possible. This saves needless execution of the operator sp_extract. The same technique also applies when no non-spatial attributes are part of the projection list. In such a case, we can stop maintaining the corresponding database relation as early as possible and avoid needless executions of the operator db_extract.

Example 3: Suppose that the query in Example 1 is slightly modified to be:

```
select road_name
  from roads
 where in_window(road_coords,w)
       and road_name != "Route 1"
```

Notice that only the attribute `road_name` is to be projected. Therefore, Plan 1 in Figure 5 can be expressed as given in Figure 18 taking into consideration the pro-

$$
\begin{aligned}
S_1 &\leftarrow sp\_window(S, w) \\
T_1 &\leftarrow db\_extract\_p(R, S_1, road\_name) \\
T_2 &\leftarrow db\_select(T_1, db\_cond)
\end{aligned}
$$

Figure 18: Effect of projection on Query Plan 1. Only attribute `road_name` is to be returned

jection rule mentioned here. Projection is performed along with the extraction operation through the operator *db_extract_p* (here, p denotes project). Notice that the invocation of operator sp_extract has been eliminated from the plan. Also, notice that we can eliminate relation T_1 by applying the pipelining rule.

Plan 9 - Composing operations: This strategy applies only when there are at least two spatial or relational operations that refer to the same spatial attribute or to the same relation, respectively. In either case, only

one execution of the operator sp_extract or db_extract needs be performed after all the composed operations are executed.

Example 4: Find the road passing through the University of Maryland campus that is nearest to the Computer Science Department (for simplicity, specified here by its coordinate values (cx,cy)).

```
select road_name
  from roads land-use
 where pass_through(road_coords,location)
       and name = "University of Maryland"
       and nearest_to(road_coords,cx,cy)
```

Originally, the operator db_extract would be executed twice, as part of the execution of each of the extended spatial operators pass_through and nearest_to to build up the relational result of the execution of each operator. Since both spatial operators refer to the same spatial attribute, namely **road_coords**, both spatial operations can be cascaded on the same spatial data structure followed by only one execution of db_extract.

Example 5: Find all industrial regions and airports that lie in the same zip code region.

```
select all
  from land-use l1, land-use l2
 where l1.usage = "industrial"
       and l2.usage = "airport"
       and l1.zip_code = l2.zip_code
```

Since no spatial operations exist in the above query, there is no need to maintain the spatial data structures (via the operator sp_extract) until the end of the three relational operations. At that stage, only a final execution of sp_extract is needed to build the data structure containing the selected objects (i.e., the airports and industrial regions).

Plan 10 - Application-dependent alternative operations: Depending on the context we can replace some of the operations by alternative ones that yield the same results. This may lead to better performance in some cases. Two examples are given below.

Nearer vs. nearest: If the nearest spatial object to a given point does not meet all the conditions in the query, then we may want to perform another nearest object computation with the remaining spatial objects. An alternative option is to always defer the nearest object computation to the end until all other query conditions are met. However, this option restricts the order in which the query conditions are executed. Another option is to perform a *nearer* object computation instead of a nearest object computation. The operator nearer builds a list of spatial objects that are sorted by their distance from the given point (i.e., it returns the nearest object, 2nd nearest, 3rd nearest, etc.). Hence, if the nearest object does not meet all the query conditions, then we pick the first object from the top of the list that meets them. Therefore, we do not need to repeat the nearest object computation several times. The best option can be chosen based on a cost model.

Simplified operations: Consider the following query that finds the names of the airports in the State of Maryland.

```
select l.name
  from land-use l states s
 where s.name = "Maryland"
       and l.usage = "airport"
       and intersect(l.location,s.location)
```

This query assumes a relation, called **states**, containing information about different states. It has one spatial attribute, called **location**, of type **REGION**. Only the name of the airport is to be returned by the above query. As mentioned in Plan 8, there is no need to maintain the temporary spatial data structures. In addition to the fact that this saves execution time, it may also simplify the spatial algorithms involved. In this case, the intersect operator need only return the tuple-ids of the intersecting regions. The intersection algorithm may be totally different if we do not want to output the spatial details of the intersecting regions, but only the fact that they do intersect. In summary, having multiple versions of operators can result in different cost estimates for each version. This would help in plan selection in the optimization process. Other examples of simplified operators are also demonstrated in Query Plans 3 and 4 where two versions of the in_circle and in_window operators are considered, respectively.

4 Conclusions

We have shown how standard query processing and optimization strategies can be adapted to spatial database systems. When the following requirements are met, all the query processing and optimization strategies mentioned in this paper (except for the application-dependent ones such as Plan 10) can be applied to other types of data in different application domains, not necessarily for spatial data:

- a process to perform the specific algorithms for handling the complex data object,
- a relational DBMS to store the thematic description of the complex object,
- the maintenance of a dual architecture in which both the complex object handler and the relational DBMS are linked together via forward and backward links between each complex object and its thematic counterpart, and
- the existence of *extended operators* that preserve the consistency between both components of a complex object.

Future research includes building a cost model for analyzing the suggested spatial query processing and optimization strategies as well as building a spatial query optimizer to experiment with such strategies.

References

W. G. Aref and H. Samet. An approach to information management in geographical applications. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 589–598, Zurich, Switzerland, (July 1990).

W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In *Second Symposium on Large Spatial Databases*, Zurich, Switzerland, (August 1991).

L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. Technical Report 312, Dortmund University, Dortmund, West Germany, (August 1989).

U. Dayal. Query processing in a multidatabase system. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 81–108. Springer-Verlag, New York, (1984).

R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, CA, (1989).

G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, volume 16, pages 160–172, San Francisco, (May 1987).

R. H. Güting. Gral: An extensible relational system for geometric applications. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, pages 33–44, Amsterdam, (August 1989).

L. M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 377–388, Portland, OR, (June 1989).

M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, (June 1984).

B. C. Ooi. *Efficient Query Processing for Geographic Information Systems*. PhD thesis, Monash University, Victoria, Australia, (1988). (Lecture Notes in Computer Science 471, Springer-Verlag, Berlin, 1990).

J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, DC, (May 1986).

R. Sacks-Davis, K. J. McDonell, and B. C. Ooi. GEOQL - A query language for geographic information systems. Technical Report 87/2, Monash University, Victoria, Australia, (July 1987).

H. Schek and M. Scholl. The two roles of nested relations in the DASDBS project. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, number 361, pages 50–68. Springer-Verlag, Berlin, (1989).

H. Schek and W. Waterfeld. A database kernel system for geoscientific applications. In *Proceedings of the 2nd International Symposium on Spatial Data Handling*, pages 273–288, Seattle, WA, (July 1986).

A. Wolf. The DASDBS GEO-Kernel: Concepts, experiences, and the second step. In *Design and Implementation of Large Spatial Databases, Proceedings of the First Symposium SSD'89*, pages 67–88, Santa Barbara, CA, (July 1989).

E. Wong and K. Youssefi. Decomposition - A strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, (September 1976).