In *O. Gunther and H.-J; Schek, editors Advances in Spatial Databases–2nd Symposium, SSD'91*, vol. 525 of Springer-Verlag Lecture Notes in Computer Science, pages 299-318, Zurich, Switzerland, August 1991.

Extending a DBMS with Spatial Operations [1]

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

A central problem in modern database design is how to resolve spatial operations with normal database operations in an extended relational database environment. A data architecture that matches the requirements for efficient processing of spatial queries in the extended database environment is proposed. It provides an equal opportunity for both the spatial components and the non-spatial components of the data to participate in query processing and optimization. The notion of *extended operators* to integrate homogeneously both spatial and non-spatial operations is introduced. Although intended primarily for spatial data, extended operators also provide a proper interface for integrating multi-media data into a database environment. The implications of this data architecture are presented. They include their effects on standard database operations, how spatial operations are integrated into the database management system (DBMS) for efficient processing, and how query processing and optimization are performed in this architecture. The operations of insertion and deletion, relational-based selection and join, and spatial-based selection and join are redefined in terms of extended operators. Spatial query processing is also described using extended operators. This data architecture can be built on top of an extensible database management system. Since it is dedicated towards efficient spatial query processing, this architecture can be used for testing and validating the extensibility of such systems and their effectiveness for supporting spatial data.

1 Introduction

Today, one of the most important database problems is how to support classes of applications that are not well served by conventional relational systems. Some examples of non-standard applications are computer-aided design (CAD), geographic information systems, and image databases. These applications are characterized by the existence of complex objects.

Much attention has been devoted to extending the relational model to provide better support for such applications (e.g., earlier attempts by [25, 34, 37]). Many prototype systems are suggested that support complex objects. Some systems are dedicated towards certain applications (e.g.,

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

CAD databases), while others address a wider spectrum of applications and hence are more general. Although dedicated systems provide efficient processing of data in their application domain [12], they lack high-level data definition facilities (e.g., [32, 38]). Also, no flexibility is provided to perform actions not previously envisioned by the system's designers.

Extensible DBMSs (e.g., [5, 6, 13, 14, 31, 36]) as well as object-oriented DBMSs (e.g., [11, 17]) are attempts at constructing a generalized DBMS that can support unconventional applications easily. Each system adds some new constructs to offer additional modeling power. Included among the new constructs are support for abstract data types (e.g., [7]), procedural fields (e.g., [35, 36]), complex objects (e.g., [7, 16, 29]), set-valued attributes (e.g., [9, 27, 33, 42]), etc. A good survey of some of the above constructs as well as other constructs is presented in [15].

Three issues need to be investigated for an application domain before deciding if a given extensible system is suitable for supporting that domain. First, we need to determine the requirements of the application domain that are necessary for efficient processing. Second, we need to design a data architecture that fulfills these requirements, and hence matches the efficiency goal. Third, this design should be capable of being prototyped and built on top of an existing extensible system for testing and validation. In this paper we address the second issue. Our application domain is spatial databases. We design a data architecture for efficient spatial data processing. It can be built either on top of an existing extensible database management system or in a specially designed efficient spatial database management system.

Several researchers have addressed the requirements of spatial data models and spatial data handling. A good discussion of these requirements can be found in recent work by van Oosterom [39]. Because of the different nature of spatial and non-spatial data, one major requirement is that each data type be represented by a specific data structure that suits its operational needs. However, for efficient processing, these data structures have to be tightly integrated in the spatial data model.

Many prototype systems are suggested to support spatial objects. A general criticism of most of these systems is the way in which they are developed. One way is as dedicated systems towards certain applications without a full understanding of database issues such as the lack of high-level data definition facilities (e.g., [32, 38]). An alternative way is as a general database tool that supports a wide variety of applications, in most cases, without full understanding of the needs of such applications. In either case, this results in less efficient processing capabilities.

In this paper we try to bridge this gap in the application domain of spatial databases. We describe an architecture for supporting spatial data applications efficiently. Our design is neutral in the sense that it is not biased towards either spatial or non-spatial data. Our approach is an extension of [2]. In particular, we describe the implications of augmenting spatial and non-spatial data in a database environment when spatial data is structurally separated from non-spatial data. We show how standard database operations (e.g., selection and join) are realized under this architecture. We introduce the notion of *extended operators* to integrate homogeneously both spatial and non-spatial operations. Spatial operations as well as standard database operations are redefined using extended operators. Although intended for spatial data, extended operators also provide an appropriate interface for integrating multi-media data into a database environment. In addition, we demonstrate how to use extended operators for query processing and optimization. We call our system SAND (denoting Spatial And Non-spatial Data).

The rest of this paper is organized as follows. Section 2 gives an example that will be used throughout the paper. Section 3 presents a classification of spatial operations into spatial selects and joins. It also discusses the overlay operation. Section 4 describes the SAND spatial database architecture where Section 4.1 discusses attributes in SAND and introduces the concept of a

spatial relation as a way of storing and linking spatial and non-spatial data together. The implications of this architecture are presented in Section 4.2 which also shows how traditional database operations are affected. In addition, it points out how dedicated spatial operations need to be adapted to fit in a database environment. Section 5 defines the concept of extended operators as well as their implementation. This is done in conjunction with the two operations: *spatial extract* and *relational extract*. They help synchronize the spatial and the relational structures into a consistent form. Section 6 is devoted to spatial query processing in the SAND architecture. It also shows the feasibility of spatial query optimization in SAND. Related work on spatial database architectures is discussed in Section 7. Section 8 contains concluding remarks and plans for future work.

2 A Land-use and Road Database Examples

In order to make our discussion concrete, we will often refer to the land-use database given in Figure 1, and the road database given in Figure 2. The land-use database models the different usages of pieces of land. The land is divided into non-overlapping sub-regions each of specific use. The roads database models roads of a given region. The schema `land-use` has one spatial attribute, `location` of type `REGION`, that represents the physical space that each piece of land occupies. The schema `roads` has one spatial attribute, `road_coords` of type `LINE_SEGMENT`, that specifies the location of the line segment with respect to other line segments in the database.

These databases are obviously too simple to be considered realistic; but they do serve the purposes of this paper. The schema of our database is described in an extended SQL-like syntax [2]. We use an SQL-like interface as a vehicle to present our ideas. Any other query language can be used with minor changes to the underlying design. Designing a more suitable interface will be among our concerns in a later stage in the development of our system. For now, we assume that the reader is familiar with standard SQL queries.

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

Figure 1: Land-use database schema

The user views the system as an extended relational database management system. In addition to the usual attributes in relations, the user can define spatial attributes in a homogeneous way. The system supports four types of spatial entities: points, line segments, polygons, and regions. The data types for them are `POINT`, `LINE_SEGMENT`, `POLYGON`, and `REGION`, respectively. The system can be extended easily to support other spatial entities in the same way.

In addition to the standard SQL commands, spatial operations are augmented in order to process and query spatial data. Spatial attribute data is stored in suitable spatial data structures. This feature along with other characteristics are discussed in more detail in Section 4. Below, we demonstrate some of these spatial operations along with some example queries.

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

Figure 2: Roads database schema

3 Operations

According to our model, we classify spatial operations into spatial select and spatial join operations. They resemble the relational select and join operations but in a spatial context. Spatial as well as relational select and join, in the context of the SAND system, are described in more detail in the following sub-sections.

3.1 Spatial and Relational Select

The following query retrieves all the regions inside a given query window:

```
select all
  from land_use
 where in_window(location,x,y,width,height)
```

The `in_window` operation in the `where` clause is termed a spatial-based selection. Spatial-based selection means that objects are selected by Boolean qualifications that refer only to the objects' spatial attributes. On the other hand, a relational-based selection means that objects are selected by Boolean qualifications that refer only to the objects' non-spatial attributes. Below, we give some examples of relational selections on the `land-use` schema of Figure 1.

```
usage = "University"
name = "University of Maryland"
importance > 9
```

Table 1 lists some spatial-based selections. Notice the use of r_i, l_i and p_i to specify instance values of region, line and point spatial attributes, respectively. s_i is more general and is used to include any instance value of a region, line or point attribute. Table 2 lists some spatial aggregate functions that can be composed in a Boolean condition to form spatial selections (e.g., see the last entry in Table 1).

3.2 Spatial and Relational Join

Join is a binary operator for combining two spatial relations. A theta-join is a join that handles comparisons between columns that are θ-comparable where θ is a comparator. If the comparator is based on spatial attributes, the operation is called a spatial join, while if the comparator is

Table 1: Some Spatial Selection Conditions

Spatial Selections	Description
in_window w	True if s_i lies inside the rectangular window w
in_circle c	True if s_i lies inside circle c
nearest_to p	True if s_i is nearest to point p
object_at p	True if s_i is located at point p
area $> a$	True if area of r_i is greater than the value a

Table 2: Some Spatial Aggregate Functions

Spatial Aggregate	Description
area	returns the area of a region r_i
perimeter	returns the perimeter of s_i
centroid	returns the centroid of s_i

based on a non-spatial attribute, the operation is called a non-spatial or relational join. Consider the following two examples. The first corresponds to a spatial join while the second corresponds to a relational join.

Example: Find all the adjacent regions to universities.

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

The spatial condition `adjacent_to` qualifies all the adjacent neighbors of a given set of regions. The result of this condition consists of a join relation which contains all the attributes of the two participating relations, including the two spatial attributes (i.e., the `location` attribute). It also generates two spatial data structures, corresponding to the two spatial attributes in the join relation. There is one data structure for the university regions and one map for their neighbors. A tuple in the resulting join relation corresponds to two spatial objects that are adjacent to each other.

Table 3 lists other spatial join conditions. Notice that we use the same notation presented in Section 3.1 to specify instance values of region, line, or point attributes.

It is interesting to note that in the literature, the term *spatial join* has several meanings. For example, in [22], Orenstein uses the term spatial join to mean, in the context of this paper, the spatial join intersection operation. Ooi and Sacks-Davis [28, 21] do not distinguish between spatial join and spatial selection. They use the term spatial join to mean both join and selection. They perform selections by fixing one of the arguments of the spatial join. However, no join action, in the conventional database sense, takes place in this case. In fact, it is more natural to view a window operation as a selection operation rather than a spatial join with a constant object (the window) that has no further non-spatial attributes. Güting [13] uses spatial joins

Table 3: Some Spatial Join Conditions

Spatial Join	Description
pass_through	True if l_i or r_i pass through s_j
adjacent_to	True if r_i is an adjacent neighbor of r_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

and spatial selections in almost the same way as we do. However, there are some differences in the interpretation and effect of spatial operations as well as in the way spatial data is stored and maintained. This will be discussed further in Section 7.

The relational join is demonstrated by the following example query, which retrieves all universities in the database that have the same zip code but one is less important than the other.

```
select all
  from land-use l1, land-use l2
 where l1.usage = "university"
   and l2.usage = "university"
   and l1.zip_code = l2.zip_code
   and l1.importance < l2.importance
```

The output relation resulting from the join will have two spatial attributes each referring to a region whose usage is university (e.g., one for the less important universities and one for the more important universities).

4 The SAND Spatial Database Architecture

4.1 Attributes and Spatial Relations

In a typical spatial database environment, spatial objects are described by spatial attributes as well as non-spatial attributes. As an example, a road is described in a database by a collection of line segments specified by the coordinates of their endpoints (i.e., spatial attributes); as well as the road name, road width, traffic directions, number of lanes, etc. (non-spatial attributes). We make the following assumptions about the way in which the non-spatial and spatial attributes describing an entity are linked.

1. Each data entity is represented by the most efficient form that suits its operational purpose. For example, spatial attributes are stored in spatial data structures and not flattened in database records.

2. Each set, say O, of homogeneous objects (i.e., of the same data type such as line data) that are spatially related to each other (e.g., are in proximity, or belong to a given region) is logically clustered in the database (e.g., stored together in database relations or in suitable spatial data structures). O is also described by a schema that has two sets of attributes: R and S. R is the set of non-spatial attributes while S is the set of spatial attributes. For

example, the schema definition shown in Figure 2 represents roads (homogeneous line data) in a given county (i.e., spatially related). The set of objects `roads` is described by the set of spatial attributes containing just one spatial attribute, namely `road_coords`, and the set of non-spatial attributes containing the other attributes in the schema. Different structures are used to store the data instances of the different sets of attributes.

All data instances of a spatial attribute over the set of homogeneous objects are stored in one spatial data structure suitable for handling the attribute's spatial type (e.g., region, line, or point). We refer to this data structure as a *map*. A map is associated with each spatial attribute in the schema. A map 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, point-in-region test, windowing, polygon intersection, area of a region, connected component retrieval, proximity queries, etc.).

The instances of a spatial attribute are merely some spatial indexes that point to the spatial description of the objects stored inside the spatial data structures. On the other hand, non-spatial data is stored in database relations. It is important to mention that a database relation is used in SAND only as a repository for clustering data instances of non-spatial attributes. Stated differently, the SAND spatial database architecture does not depend heavily on the fact that relations or a relational system is being used. It is how spatial and non-spatial data are linked together between the spatial and non-spatial repositories that is important. In SAND, the spatial and non-spatial description of an object are linked to each other by what we term a *spatial relation*. A spatial relation is composed of two main components: a spatial and a non-spatial repository. The spatial repository may be composed of one or more spatial data structures while the non-spatial repository is simply a relation. Figure 3 shows several forms of spatial relations.

Figure 3: (a) A spatial relation with one spatial attribute A, (b) a spatial relation with two spatial attributes A1 and A2.

In particular, in a spatial relation, spatial and non-spatial attribute values of an object are linked by two types of links: *forward* and *backward* links. A forward link maps the non-spatial description of an object that is stored inside a tuple into the object's spatial description that is stored inside a spatial data structure. A backward link serves as a mapping in the opposite direction - i.e., from the spatial description of an object into its non-spatial description (see Figure 3). Notice that the description of a spatial object in the spatial data structure can point back, via backward links, into more than one tuple in the relation.

A forward link is stored inside a tuple as the instance value of a spatial attribute of the relation. Its purpose is to index and uniquely identify a spatial object from among a set of related spatial objects in the same spatial data structure. Regardless of the type of the spatial attribute being indexed, the forward link index value is of fixed length and does not depend on the size of the spatial object being referenced. Backward links are stored inside the spatial data structure along with each spatial object description, or in a separate look-up table that maps the spatial description of an object back to its non-spatial description (basically a tuple) inside a database relation. A backward link can be implemented by storing the tuple identifier of the object's corresponding tuple.

4.2 Some Implications of the SAND Architecture

The SAND architecture structurally separates the spatial from the non-spatial data. This has several implications on data manipulation and query processing. Here we demonstrate the implications that are related to insertion and deletion of objects, as well as relational-based and spatial-based selection and join. Section 5 shows how these operations, as well as others, are implemented in SAND.

We use the following notation. A set of homogeneous objects O is represented by a relation R and a data structure S. We assume that O's schema has only one spatial attribute in it. Extending the discussion to sets of more than one spatial attribute is straightforward but is not done here.

Insertion and Deletion: Performing insertion and deletion of spatial objects in SAND is simple. Basically, if any object o is to be inserted into (deleted from) the homogeneous set of objects O, then o's corresponding tuple in R has to be inserted (deleted). Also, o's representative entity in S storing the spatial aspect of o has to be inserted (deleted) as well. Finally, forward and backward links are established between both components of o.

Relational-based Selection: If some tuples are selected from relation R via a relational-based selection to form a new relation R', then a new data structure S' is also formed to store the spatial aspect of the selected objects in R'. This is achieved by the *sp_extract* operator, given in Figure 4 (*sp* denotes spatial). sp_extract extracts spatial data from S via the use of forward links from R to S. The extracted spatial data is inserted into S'. Backward links of the selected objects in S' have to be adjusted to point to their corresponding tuples in R' rather than R.

Spatial-based Selection: If some tuples are selected from data structure S via a spatial-based selection to form a new data structure S', then a new relation R' is also formed to store the corresponding tuples (the non-spatial aspect) of the selected objects in S'. This is achieved by the *db_extract* operator (*db* denotes database or relational). db_extract extracts tuples from R via the use of backward links from S to R and inserts them into R'. Forward links of the selected objects in R' have to be adjusted to point to their corresponding spatial entities in S' rather than

```
sp_extract(R1,S)
/* For all the tuples in R1, retrieve their spatial
   description from S and store it in a new spatial
   data structure U. */
begin
  initialize U;
  traverse  R1;
  for each tuple t in R1 do
    begin
      sid := get t's spatial-id;
      retrieve sid's spatial object o from S;
      insert o into U;
    end;
  return U;
end;
```

Figure 4: Retrieval of spatial description of objects given their non-spatial description

S.

Relational and Spatial Join: The standard relational join operation behaves slightly differently when performed on top of the SAND data architecture. When we join two relations where one or both of them have spatial attributes, we have to build new spatial data structures to store the spatial information of the tuples participating in the join operation. This involves the execution of the operation sp_extract twice; once for each spatial attribute to store the object regions that participated in the relational join.

On the other hand, a spatial join on two spatial relations, each having one spatial attribute, is performed as follows: the spatial operation corresponding to the spatial join is executed first. It identifies the matching pairs of spatial objects that will participate in the resulting spatial relation. Two spatial data structures are created where each of them stores the qualifying spatial objects that originate from one of the two input spatial relations. Two invocations of the operator db_extract are needed in order to build the resulting join relation. We need to retrieve the tuples corresponding to the qualifying objects from each of the two input spatial relations have to be retrieved. Then, tuples of matching spatial objects are merged and stored in the join relation. All the operations involved in performing a spatial or relational select and join are encompassed in what we term *extended operators*. They are described in more detail in the following section.

5 Extended Operators and Spatial Relations

We now address some implementation issues in SAND. In Section 4.2 we saw that standard database operations such as joins and selections as well as spatial operations need to be redefined when viewed in the context of the SAND data architecture. The reason is that we must maintain consistency among separate but related structures; namely the relation describing the non-spatial aspect of homogeneous objects (i.e., of the same type and in proximity) and the spatial data structures describing the spatial aspect of the objects.

In this section, we introduce the notion of extended operators as an implementation tool that encapsulates spatial relations. We redefine spatial and non-spatial operations using extended operators. Extended operators provide a uniform interface that masks the differences between spatial and non-spatial operations. They also serve as a way to keep both the spatial and non-spatial components of a spatial relation consistent.

We assume that we have a collection of homogeneous objects O referred to by the pair $< R, S >$, where R is a relation that stores the non-spatial attribute information of O and S is a spatial data structure that stores the spatial attribute information of O. Again, 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 performed on U.

Extended operators are classified into relational- and spatial-based operators; x_op_r and x_op_s, respectively (r denotes $r$elational and s denotes $s$patial). They are defined using the operators sp_extract and db_extract, described in Section 4.2, as building blocks in addition to the corresponding un-extended operators. We first discuss the relational and spatial select operators. Then we discuss the relational and spatial join operators.

The Extended Select Operator: The extended select operator can be either spatial or relational. The extended relational select, referred to as x_op_r, takes as its arguments the pair $< R, S >$ and returns the pair $< T, U >$. Relation T is formed by applying the un-extended relational select operator on R, i.e., $T = op_r(R)$. Next, operator sp_extract (described in Section 4.2) is executed to form the resulting spatial data structure U. The spatial objects corresponding to the tuples in the resulting relation T are stored into U. The spatial and non-spatial components of each resulting object are then linked.

The extended spatial select operator x_op_s behaves analogously. First, the spatial select operator op_s is performed and the selected spatial objects are then stored into a resulting spatial data structure U. Operator db_extract (described in Section 4.2) is then used to extract the tuples, corresponding to the spatial objects stored in U, to build an output relation. In summary,

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

The Extended Join Operator: The extended relational join $< R_3, \{S'_1, S'_2\} >= x\_op_{rj}(< R_1, S_1 >, < R_2, S_2 >)$ (rj denotes $r$elational $j$oin) first applies the relational join operator op_{rj} on R_1 and R_2. This results in a joined relation R_3 which has two spatial attributes $R_3.s_1$ and $R_3.s_2$, where s_1 and s_2 are the spatial attributes that originally belonged to R_1 and R_2, respectively. Operator sp_extract is executed twice, once for each spatial attribute to build the corresponding data structures S'_1 and S'_2. In summary,

$$\begin{aligned} x\_op_{rj}(< R_1, S_1 >, < R_2, S_2 >) &= < R_3, \{S'_1, S'_2\} >, \text{where} \\ R_3 &= op_{rj}(R1, R2), \\ S'_1 &= sp\_extract(S_1, R_3.s_1), \text{ and} \\ S'_2 &= sp\_extract(S_2, R_3.s_2). \end{aligned}$$

The extended spatial join $< R_3, \{S'_1, S'_2\} >= x\_op_{sj}(< R_1, S_1 >, < R_2, S_2 >)$ (sj denotes $s$patial $j$oin) first applies the spatial join operator op_{sj} on S_1 and S_2. This results in two data structures,

$S'_1 \subseteq S_1$ and $S'_2 \subseteq S_2$. S'_1 and S'_2 represent the complete description of the spatial objects participating in the spatial join from each side. In some cases op_{sj} may result in just one data structure, say S'_3 that carries some combined spatial information. We also use an operator that we term *db_j_extract* which operates on the output spatial data structures (S'_1 and S'_2, or on S'_3) to produce the join relation, R_3. In summary,

$$\begin{aligned} x\_op_{sj}(< R_1, S_1 >, < R_2, S_2 >) &= < R_3, \{S'_1, S'_2\} >, \text{ or} \\ &= < R_3, S'_3 >, \text{where} \\ \{S'_1, S'_2\} &= op_{sj}(S_1, S_2), \text{ and} \\ R_3 &= db\_j\_extract(R_1, R_2, \{S'_1, S'_2\}), \text{or} \\ S'_3 &= op_{sj}(S_1, S_2), \text{ and} \\ R_3 &= db\_j\_extract(R_1, R_2, S'_3). \end{aligned}$$

6 Query Processing

Suppose we want to find the names of the parks within 10 miles from a university and that lie entirely inside the window `W`. The extended SQL query looks as follows:

```
select L.name L.usage
  from land-use L, land-use U
 where U.usage = "university"
   and L.usage = "park"
   and within(U.location,L.location,10)
   and in_window(L.location,W)
```

By using extended operators we have a homogeneous interface. In other words, each extended operator results in a relation and one or more spatial data structures (depending on the number of spatial attributes). This allows permuting the operations in any desired order. Also, it demonstrates the flexibility of the SAND architecture. Below, we describe several possible execution plans of the above query. Next, we demonstrate a few feasible optimizations that can take place. To ease the presentation, we use the following short-hand notation for the query predicates:

x_db_p : `select L.name, L.usage (db-projection)`

x_db_{s1} : `U.usage = "university" (db-selection)`

x_db_{s2} : `L.usage = "park" (db-selection)`

x_sp_{w1} : `within(U.location,L.location,10)`

x_sp_{w2} : `in_window(L.location,W)`

Consider the following different orders of executing the query, which we term execution plans:

query plan 1: x_db_{s1}, x_db_{s2}, x_sp_{w1}, x_sp_{w2}, x_db_p

query plan 2: x_db_{s1}, x_db_p, x_sp_{w2}, x_db_{s2}, x_sp_{w1}

query plan 3: x_sp_{w2}, x_db_{s2}, x_db_{s1}, x_sp_{w1}, x_db_p

query plan 4: x_db_{s2}, x_sp_{w2}, x_db_{s1}, x_sp_{w1}, x_db_p

We describe query plan 1 in more detail. Notice that we present the execution plan using the most general form of extended operators described in Section 5. This is only for clarification purposes. However, several optimizations can take place [3].

- x_db_{s1} : `U.usage = "university"`
  Relation `land-use` is searched (possibly through an index on attribute `usage`) and the qualified tuples are selected. The result of the selection operation is stored in a temporary relation T_{s1}. S-extract traverses T_{s1} and extracts from S, the spatial data structure storing instances of the attribute `land-use.location` of type region, the regions (universities) corresponding to the tuples in relation T_{s1}. The extracted regions are stored in a temporary spatial data structure S_{s1}. Notice that the database selection is performed by the DBMS and the spatial extract operation is performed by the spatial process.

- x_db_{s2} : `L.usage = "park"`
  This operation is executed in the same way as operation x_db_{s1}. It operates on the original relation `land-use` but outputs a temporary relation T_{s2} and a temporary spatial data structure S_{s2}.

- x_sp_{w1} : `within(`T_{s1}`.location,`T_{s2}`.location,10)`
  Given relations T_{s1} and T_{s2}, spatial data structures S_{s1} and S_{s2}, and a radius of expansion with value 10, the spatial operator `within` first operates on S_{s1} and S_{s2} to generate the regions in S_{s2} that are within 10 units of distance from regions in S_{s1}. The result is another temporary spatial data structure S_{s3}. By using the operator db_extract and S_{s3}, the spatial join relation is built consisting of tuples from T_{s1} and T_{s2} that are within the given distance from each other. The result of the spatial join is stored in temporary relation T_{s3} that has two spatial attributes `location1` and `location3` of type `region` corresponding to the data structures S_{s1} and S_{s3}, respectively. For referencing purposes only, all the attributes joined from relation T_{s1} are postfixed by the letter 1, (e.g., `usage1`), while the attributes joined from relation T_{s2} are postfixed by the letter 3, (e.g., `usage3`).

- x_sp_{w2} : `in_window(`T_{s3}`.location3,W)`
  The spatial selection operation `in_window` performs the window operation on S_{s3}, the data structure corresponding to spatial attribute `location3`. This results in yet another temporary data structure S_{s4}. The tuples in T_{s3} corresponding to the spatial objects in S_{s4} (i.e., inside the window W) are then extracted using operator db_extract into the temporary relation T_{s4}.

- x_db_p : `select` T_{s4}`.name3,` T_{s4}`.usage3`
  Attributes `name3` and `usage3` are projected from T_{s4} resulting in relation T_{s5}. All other attributes including their corresponding spatial data structures, if any, are deleted. Figure 5 shows a summary of the above execution plan.

Extended operators can be executed in any desirable order because of their uniform interface. This gives us flexibility in the sense that it enables us to to perform query optimization. In addition, once one candidate execution plan is selected for execution, more optimizations can take place. We start with a general and complete execution plan of the query using extended operators, then apply some transformation rules that enhance the performance of the execution plan while still producing the same correct results as the original execution plan. In other words, depending on the context, we can select less general and yet simpler forms of extended operators that are necessary to execute the given query with better performance. Below, we only list a

$$
\begin{aligned}
x\_db_{s1}(< U, S >) &\rightarrow < T_{s1}, S_{s1} > \\
x\_db_{s2}(< L, S >) &\rightarrow < T_{s2}, S_{s2} > \\
x\_sp_{w1}(< T_{s1}, S_{s1} >, < T_{s2}, S_{s2} >, 10) &\rightarrow < T_{s3}, S_{s3} > \\
x\_sp_{w2}(< T_{s3}, S_{s3} >) &\rightarrow < T_{s4}, S_{s4} > \\
x\_db_{p}(< T_{s4}, S_{s4} >) &\rightarrow < T_{s5}, S_{s5} >
\end{aligned}
$$

Figure 5: Summary of Query Plan 1

$$
\begin{aligned}
x\_db_{s1,p}(< U, S >) &\rightarrow < T'_{s1}, S_{s1} > \\
x\_db_{s2,p}(< L, S >) &\rightarrow < T'_{s2}, S_{s2} > \\
x\_sp_{w1}(< T'_{s1}, S_{s1} >, < T'_{s2}, S_{s2} >, 10) &\rightarrow < T'_{s3}, S_{s3} > \\
x\_sp_{w2}(< T'_{s3}, S_{s3} >) &\rightarrow < T_{s5}, \phi >
\end{aligned}
$$

Figure 6: The result of applying the projection optimization rule to query plan 1

range of these rules. The reader is referred to [3] for a detailed coverage of other feasible spatial optimization strategies.

Rearranging the order of the above operations may result in better performance. For example, in the above plan, performing the spatial selection `in_window` before the spatial join `within` could have resulted in better performance. Also, performing the database projection earlier reduces the I/O overhead. Query plan 2, given above, reflects both of these enhancements. In this case, the spatial query optimizer stops maintaining the spatial data structure of the dropped attribute as early as possible. This is also applicable when the user does not select relational attributes in the `select clause` (e.g., `select road_coords from` ..). Applying this rule to query plan 1 results in the query plan summarized in Figure 6. Notice that relations U and L are projected at the same time that the selection operations x_db_{s1} and x_db_{s2} are performed on U and L, respectively.

In addition, several optimization steps can be performed within each operation sequence (application plan). As an example, we can avoid the creation, writing, and subsequent reading of several temporary relations and spatial data structures (e.g., S_{s4} is built and stored but was deleted after the projection step).

As another example, if we cascade two spatial operations, we can defer the db_extract operator until both spatial operators are performed. This means that we can build the relational part at the end using one db_extract operator. This will save one execution of the db_extract operator as illustrated by the following example. Suppose we wish to perform the operation $x\_op_{s2}(x\_op_{s1}(< R, S >))$ where x_op_{s1} and x_op_{s2} are extended spatial operators. One way to perform this operation is as follows. Let S_1 be $op_{s1}(S)$ and R_1 be db_extract(R, S_1). Then,

$$x\_op_{s2}(x\_op_{s1}(< R, S >)) = < db\_extract(R_1, op_{s2}(S1)), op_{s2}(S1) >.$$

The above formula uses the standard definition of extended operators. However, a better way to perform the same operation is to use R instead of R_1 in the above formula. The formula then is:

$$x\_op_{s2}(x\_op_{s1}(< R, S >)) = < db\_extract(R, op_{s2}(S1)), op_{s2}(S1) >.$$

The new formula is functionally equivalent to the former one but is more efficient. It uses just one execution of the db_extract operator. This optimization saves execution time since it avoids creating, writing, and then reading an intermediate temporary relation. Other possible optimization strategies (e.g., intersecting spatial or tuple pointers, and simplified versions of spatial operations) are presented in [3]. We expect more domain-specific rules to emerge as we begin the implementation phase of SAND.

7 Related Work on Spatial Database Architectures

Supporting bi-directional links between database objects emerged as early as hierarchical and network database management systems (e.g., IMS [20] and CODASYL DBTG [8]). They were also proposed in [18] as an enhancement to System R [4] to support complex objects. In [19] bi-directional links were used in the context of geographical databases where a geographical object is represented by a tuple having some unique identifier and a long field to store the geographic representation of the object in addition to other non-spatial attributes. Links were used to express explicit relationships between geographical objects (e.g., parent-child, or reference relationships).

The idea of introducing abstract data types (ADTs) as new attribute domains into a relational database system and supporting user-defined index structures appears in [34, 37]. However, only backward links are supported by this architecture. Our work can be viewed as a merge of the ADT work in [34] and the complex object support in [18]. Note that we also support ADTs, user-defined data structures, and set-oriented operations while maintaining bi-directional links between corresponding components of the ADT for efficient access. In addition, we consider a full-spectrum system for handling spatial data where we address spatial query processing and optimization techniques, and a cost model for combined spatial and non-spatial operations. More specifically, the following are the main features of the SAND architecture:

- SAND features a dual architecture where spatial data is stored separate from non-spatial data. This may not permit merging of spatial and non-spatial data into common multiple attribute indexes. However, we believe that this feature is unnecessary and of less practical use.
- Spatial data is stored in disk-based spatial data structures that apply some buffering mechanism and are suitable for the type of data and required operations. This matches with the fact that spatial data is voluminous. It also avoids the need for down- or up-loading of spatial data from an off-line representation into an on-line representation (as in the case of the systems Gral [13] and Geo-Kernel [41]). Considering the large size of spatial data, the down-/up-loading approach seems impractical (notice that if the on-line representation is disk-based in order to accommodate the large number of spatial data items, then down-loading spatial data from a relation (disk-based) to another on-line representation (also disk-based) would be even more impractical).
- We maintain bi-directional links between the spatial and non-spatial descriptions of a spatial object. Although maintaining these links imposes some overhead, it provides flexibility in browsing between the spatial and non-spatial sides of the system, thereby permitting operations to be performed in any arbitrary order chosen by the optimizer. Maintaining only uni-directional links imposes some restrictions on the optimizer so that the optimizer is forced to favor one aspect of the data (whether spatial or non-spatial) over the other. An example illustrating this is given below.

- Since spatial data is stored in separate structures we avoid the duplication of spatial data when the objects are referenced more than once. Consider the relation resulting from joining two relations each having one spatial attribute. If data were stored in textual form inside a tuple (e.g., as in Gral [13]), every time this tuple participates in the join the textual description of the spatial object needs to be duplicated as well. However, in SAND, we just duplicate the spatial index referring to the spatial description and not the whole description.
- An emphasis is placed on the estimation of the cost of spatial operations as well as the cost of non-spatial operations for the purpose of query processing and optimization since in SAND spatial operations are given equal weight.

On the other hand, dual architectures such as SAND need further study in the following issues:

- Sharing and concurrency control for the different varieties of spatial data structures. Efforts along this line is un-avoidable in most architectures that support special-purpose data structures.
- Alternative and efficient ways of establishing bi-directional links (e.g., through global object-ids, relocatable tuple-ids, etc.).

We now relate our research to some of the recent and interesting research prototypes that were precursors to our work. Due to space limitations, we do not consider some other very interesting prototypes (among them are GEOVIEW [40], Csiro [1], and PSQL [26]).

Geo-Kernel [31, 41] and Gral [13] provide efficient support for spatial data. Gral uses long attributes to store spatial data while Geo-Kernel uses relation-valued attributes [30]. During query evaluation, both systems use appropriate spatial data structures to query spatial data. As mentioned above, there is a need for conversion routines back and forth between the textual and operational (or on-line) representations. A drawback of this approach is the considerable time spent in data conversion and downloading. In addition, the on-line representation has to be disk-based in order to accommodate the large volumes of the downloaded spatial data. In this case, conversion between two disk-based representations of the data (relations and data-structures as off-line and on-line representations of data, respectively) may not be efficient. Finally, it also appears that both systems maintain only one type of link, namely backward links.

The drawback of using only backward links is illustrated by the following example. Consider a query that performs just two selections; one spatial (sp_select) and one non-spatial (db_select) on a relation, say R, where R has only one spatial attribute, say A, where instances are stored in textual form inside tuples of R. Using backward links only, the following two plans can be applied ($sids$ denotes a set of spatial-ids):

$$
\begin{aligned}
&Plan1 \\
&\quad T1 \leftarrow db\_select(R) \\
&\quad S1 \leftarrow download(T1.A) \\
&\quad sids1 \leftarrow sp\_select(S1) \\
&\quad T2 \leftarrow db\_extract(T1, sids1)
\end{aligned}
$$

$$
\begin{aligned}
&Plan2 \\
&\quad S \leftarrow download(R.A) \\
&\quad sids \leftarrow sp\_select(S)
\end{aligned}
$$

$$T1 \leftarrow db\_extract(R, sids)$$
$$T2 \leftarrow db\_select(T1)$$

Both plans assume the availability of only backward links. In addition, notice that the plans are not symmetric. In dealing with such a query, Geo-Kernel always selects Plan 1 to avoid the extensive downloading of the whole relation. Plan 2 is almost always more expensive than Plan 1 for the following reason: operations download, sp_select and db_extract operate on a larger number of records in Plan 2 than in Plan 1 (because Plan 1 narrows down the scope by performing db_select first). Usually, the overhead of downloading spatial data is high relative to db_select (which is deferred in Plan 2).

The Probe system [10, 23, 24] provides another alternative, yet interesting, architecture. Probe treats space and time as generic types with the same status as integers, floating point numbers, strings, etc. In this respect, Probe differs from most extensible systems in that it augments its kernel to support spatial processing (through the point-set concept [24]) while all other systems consider space as a special type which makes their kernels application-independent and everything related to geometry application-specific.

Probe provides a general method for indexing the space by linearizing the spatial index (i.e., by mapping it to one-dimension) via bit-interleaving techniques and storing the resulting values into an attribute that is indexed by a B-tree. The application program can index the underlying space using this general spatial attribute. One disadvantage is that Probe limits itself to space-filling curve representations of spatial data when there are other interesting spatial data structures that can be used.

GEOQL [21, 28] extends SQL to support spatial applications. The underlying architecture is composed of an SQL back-end, a spatial processor, and an extended optimizer. In GEOQL, each relation is assumed to have only one spatial attribute. Multiple spatial representation of an object is difficult under this model (e.g., to model a city once as a point in a point map and once as a region by defining its bounding perimeter). GEOQL is biased towards the relational component in several aspects. In particular, even though spatial data structures are maintained, spatial operations cannot be composed directly without building intermediate database relations. This limits the efficiency of spatial query processing. Moreover, GEOQL's optimizer does not take into account the cost of spatial operations. It only optimizes the cost of non-spatial operations.

8 Conclusion, and Future Plans

The SAND architecture makes use of bi-directional links for combining spatial and non-spatial attributes efficiently. SAND's goal is to allow operations (whether spatial or non-spatial) to be performed in their most natural environment. Flexibility in the interaction between spatial and non-spatial attributes is achieved through the use of forward and backward links. Spatial relations and extended operators were introduced to provide a uniform way for interfacing with spatial as well as multi-media applications. They hide the implementation details of spatial data structures and help synchronize various related data items. Although not addressed in this paper, spatial and non-spatial integrity constraints can be attached to these extended operators as well. We also compared a number of alternative architectures for supporting spatial data in terms of the type of links that they use and the means in which the spatial data is stored (both on- and off-line).

Directions for future research include implementing SAND on top of an existing extensible system, and defining a cost model for evaluating query plans. In addition, we plan to study

the overhead of alternative spatial database architectures, analyze and compare them using the abstraction of this paper. This can be performed by considering the cost of the feasible query evaluation plans that can be offered by each architecture.

References

[1] D. J. Abel. SIRO-DBMS: A database tool-kit for geographical information systems. *International Journal of Geographical Information Systems*, 3(2):103–116, April–June 1989.

[2] 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*, pages 589–598, Zurich, Switzerland, July 1990.

[3] W. G. Aref and H. Samet. Optimization strategies for spatial query processing. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pages 81–90, Barcelona, Spain, September 1991.

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[5] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems*, 13(3):231–262, September 1988.

[6] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. Technical Report 808, University of Wisconsin, Madison, WI, November 1988.

[7] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, volume 17, pages 413–423, Chicago, IL, June 1988.

[8] CODASYL. CODASYL Data Base Task Group. *ACM*, April 1971.

[9] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, MA, June 1984.

[10] U. Dayal and J. M. Smith. A knowledge-oriented database management system. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, pages 227–257. Springer-Verlag, New York, 1986.

[11] O. Deux et. al. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[12] R. H. Guting. Geo-relational algebra: A model and query language for geometric database systems. In *Advances in Database Technology - International Conference on Extending Database Technology (EDBT 88)*, pages 506–527, Venice, Italy, March 1988.

[13] 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.

[14] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[15] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[16] W. Kim, J. Banerjee, H.Chou, J. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proceedings of the 2nd ACM OOPSLA Conference*, pages 118–125, Orlando, FL, October 1987.

[17] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[18] R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier. Supporting complex objects in a relational system for engineering databases. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 145–155. Springer-Verlag, New York, 1984.

[19] R. Lorie and A. Meier. Using a relational DBMS for geographical databases. *Geo-Processing*, 2:243–257, 1984.

[20] W. C. McGee. The IMS/VS system. *IBM Systems Journal*, 16(2):84–168, June 1977.

[21] 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).

[22] J. A. Orenstein. An object-oriented approach to spatial data processing. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 669–678, Zurich, Switzerland, July 1990.

[23] J. A. Orenstein and F. A. Manola. Spatial data modeling and query processing in PROBE. Technical Report CCA-86-05, Computer Corporation of America, Cambridge, MA, October 1986.

[24] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

[25] S. L. Osborn and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Transactions on Database Systems*, 11:357–373, 1986.

[26] N. Roussopoulos, C. Faloutsos, and T. Sellis. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639–650, May 1988.

[27] L. Rowe and M. Stonebraker. The POSTGRES data model. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 83–96, Brighton, England, September 1987.

[28] 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.

[29] H. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[30] 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. Also Lecture Notes in Computer Science 361*, pages 50–68. Springer-Verlag, Berlin, 1989.

[31] 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.

[32] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: A geographic information system based on quadtrees. *International Journal of Geographical Information Systems*, 4(2):103–131, April–June 1990.

[33] D. Shipman. The functional data model and the data language of DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[34] M. Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 262–269, Los Angeles, CA, February 1986.

[35] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, September 1987.

[36] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, Washington, DC, May 1986.

[37] M. Stonebraker, B. Rubenstein, and A. Guttmann. Application of abstract data types and abstract indices to CAD databases. In *Proceedings of the ACM/IEEE Conference on Engineering Design Applications*, pages 107–113, San Jose, CA, 1983.

[38] C. D. Tomlin. *Geographic Information Systems and Cartographic Modeling*. Prentice Hall, Englewood Cliffs, N.J., 1990.

[39] P. van Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, University, Leiden, The Netherlands, December 1990.

[40] T. C. Waugh and R. G. Healey. The GEOVIEW design: A relational data base approach to geographical data handling. *International Journal of Geographical Information Systems*, 1(2):101–118, April–June 1987.

[41] A. Wolf. The DASDBS GEO-Kernel: Concepts, experiences, and the second step. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases, Proceedings of the First Symposium SSD'89. Also Lecture Notes in Computer Science 409*, pages 67–88. Springer-Verlag, Berlin, 1990.

[42] C. Zaniolo. The database language GEM. In *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*, volume 12, pages 207–218, San Jose, CA, May 1983.