PROBE Spatial Data Modeling and Query Processing in an Image Database Application

JACK A. ORENSTEIN AND FRANK A. MANOLA, MEMBER, IEEE

Abstract—The PROBE research project has produced results in the areas of data modeling, spatial/temporal query processing, recursive query processing, and database system architecture for "nontraditional" application areas, many of which involve spatial data and data with complex structure. PROBE provides the *point set* as a construct for modeling spatial data. This abstraction is compatible with notions of spatial data found in a wide variety of applications. PROBE is extensible and supports a generalization hierarchy, so it is possible to incorporate application-specific implementations of the point set abstraction. PROBE's query processor supports point sets with the *geometry filter*, an optimizer of spatial queries. Spatial queries are processed by decomposing them into 1) a set-at-a-time portion that is evaluated efficiently by the geometry filter and 2) a portion that involves detailed manipulations of individual spatial objects by functions supplied with the application-specific representation. The output from the first step is an approximate answer, which is refined in the second step. The data model and the geometry filter are valid in all dimensions, and they are compatible with a wide variety of representations. PROBE's spatial data model and geometry filter are described, and it is shown how these facilities can be used to support image database applications.

Index Terms—Object-oriented database system, spatial data, spatial query.

I. Introduction

The numeric and string types available in most programming languages and database systems were designed to support applications that are much simpler than the computer-aided engineering, geographical, and image database applications being contemplated. In traditional business applications, fixed-point types are used. Several languages offer extensive built-in capabilities for describing the range, precision, and format of these numbers. Traditional engineering applications require floating-point numbers, which are well supported in virtually all programming languages, and standards for floating-point representation and precision are under development. Even though there is much specialization within each of these application areas, there seems to be no need for corresponding specializations of the data types mentioned. For example, the floating-point facilities used by mechanical engineers also serve the purposes of electrical and civil engineers.

The situation is much different in applications that manipulate spatial data. Notions of spatial data differ greatly from one application to another. The most obvious difference is in the dimensionality of the data. Temporal data can be viewed as one-dimensional (1-D) spatial data. Geographic applications and VLSI design involve two-dimensional data. Geological applications require three dimensions, and solid modeling sometimes requires four dimensions (to detect interference between moving 3-D objects). In some applications, spatial objects are continuous, while in others, space is best thought of as being discrete. Finally, spatial operations and representations of spatial objects differ greatly from one application to another. For example, the techniques of solid modeling (parametrically defined curves and surfaces and constructive solid geometry) are not used in geographical applications. For these reasons, it is unlikely that there will be a small collection of spatial data types that will become as widely accepted as the simpler types discussed above.

The term "image database" is a broad one, and again, there is no single view of spatial data that is adequate for all image database applications. For example, image databases often have to support both continuous and discrete representations [7], [23]. In addition to supporting a variety of image representations, an image database system has to be able to store images; provide access to special-purpose functions that process images, e.g., for feature extraction (which would be written in a conventional programming language); provide a query language to retrieve images based on content (once the features have been identified); and provide access mechanisms to evaluate these spatial queries efficiently. Unfortunately, conventional database systems provide no support for spatial data types and provide little support for the invocation of special-purpose functions. This paper is concerned with extending database system functionality to satisfy the requirements of spatial database applications in general, and we show how our approach can be used to support image database applications.

Attempts at supporting spatial applications using existing database systems (e.g., [9], [35]) have not been satisfactory. Common data models such as the relational model are not adequate for handling spatial data. While it is usually easy to develop a schema that will capture all

Manuscript received June 30, 1987; revised September 30, 1987. This work was supported in part by the Defense Advanced Research Projects Agency and in part by the Space and Naval Warfare Systems Command under Contract N00039-85-C-0263.

J. A. Orenstein is with the Computer Corporation of America, Cambridge, MA 02142.

F. A. Manola was with the Computer Corporation of America, Cambridge, MA 02142. He is now with GTE Labs, Waltham, MA 02254.

IEEE Log Number 8819884.

the data and relationships in an application, it is extremely difficult to specify even the simplest spatial operations using a query language. Furthermore, the access paths supported in existing database systems are unlikely to offer good performance since they were not designed for spatial data. For example, for containment and nearest-neighbor queries, clustering objects by proximity in space is likely to lead to better performance than clustering by a nonspatial attribute.

Attempts at extending database system capabilities (for spatial and other nontraditional applications) have been extremely application-specific. Extensions for text retrieval [33], [36], geographic information processing [22], image and pictorial databases [8], [15], and VLSI design [16], among others, have been proposed.

Recently, there has been much interest in a new approach to the problem of extending database system functionality. The key idea is to build in *extensibility* rather than a particular set of extensions. *Object-oriented* database systems permit the incorporation of *object classes* that provide a collection of specialized operations. The data model and query language of an object-oriented database system must provide a way to invoke these new operations, and the physical database must provide a way of storing the representations of objects that it does not "understand." These objects can be of variable size and may be arbitrarily large.

PROBE is a research project into object-oriented database systems (OODB's). Other OODB projects include POSTGRES [37], EXODUS [6], and STARBURST [34]. The goal of PROBE is to provide a general-purpose database system for applications involving spatial and temporal data and other kinds of data with complex structure.

In Section II, we provide an overview of PROBE, focusing on the facilities for dealing with spatial and temporal data. In Section III, we show how the PROBE database system and simple application-specific object classes combine to efficiently support PROBE's spatial data model. In Section IV, it is shown how an image database application can be supported using PROBE's data model and spatial query processor. Section V provides a summary and discusses the current status of the PROBE project and future plans.

II. PROBE's Approach to Spatial Data

In order to meet the needs of the application areas considered, we found it necessary to do research on data modeling, spatial and temporal query processing, recursive query processing, and database system architectures. In this section, the relevant results of this research are surveyed, and we focus on those results necessary for the further discussion of spatial data modeling and query processing.

A. The PROBE Data Model

The PROBE data model (PDM) has two basic types of data objects: *entities* and *functions*. An *entity* is a data object that denotes some individual thing. The basic characteristic of an entity that must be preserved in the model is its distinct indentity. Entities with similar characteristics are grouped into collections called *entity types*. For example, an image database might have an entity type **feature**, representing geographic features.

Properties of entities, relationships between entities, and operations on entities, are all uniformly represented in PDM by *functions*. Thus, in order to access properties of an entity or other entities related to an entity, or to perform operations on an entity, one must evaluate a function having the entity as an argument. For example,

- the single-argument function **population (CITY)** → **INTEGER** allows access to the value of the population attribute of a **CITY** entity;
- the function **location (POINT_FEATURE)** → **(LATITUDE, LONGITUDE)** allows access to the value of the location attribute of a point feature (note that a function can return a complex result);
- the multiargument function **altitude (LATITUDE, LONGITUDE, MODEL)** → **HEIGHT** allows access to the altitude values contained in a digital terrain model;
- the function **components (FEATURE)** → **set of FEATURE** allows access to the component features of a group feature (such as a city); and
- the function **overlay (LAYER-1, LAYER-2)** → **LAYER-3** provides access to an overlay operation defined for sets of polygons separated into different coverage layers.

Functions may also be defined that have no input arguments, or that have only boolean (truth-valued) results. For example,

- the zero-argument function **feature ( )** → **set of ENTITY** is implicitly defined for entity type **feature** and returns all entities of that type (such a function is implicitly defined for each entity type in the database) and
- the function **overlaps (POLYGON-1, POLYGON-2)** → **BOOLEAN** defines a predicate that is true if two polygons geometrically overlap. All predicates within PDM are defined as boolean-valued functions.

In PDM, a function is generically defined as a relationship between collections of entities and scalar values. The types of an entity serve to define what functions may be applied with the entity as a parameter value. There are two general classes of functions: *computed functions*, with output values computed by procedures, and *stored functions*, with output values determined by a conventional database search of a stored function *extent*. (Computed functions may involve the use of stored extents, in addition to computation.) References to all functions are treated syntactically as if they were references to computed functions, even when a stored extent exists, rather than treating the various classes of functions differently. However, particularly in the case of stored functions, functions can often be evaluated "in reverse," i.e., with "output" variables bound, to return "input" values (since both are available in a stored extent).

Entity types may be divided into *subtypes*, forming what are known as *generalization hierarchies*. For example, one might define **POINT_FEATURE** as a subtype of **FEATURE** and **ZOO** as a subtype of **POINT_FEATURE**. As another example, the declarations

type LAND_DIVISION is ENTITY
description(LAND_DIVISION) → STRING
area(LAND_DIVISION) → POLYGON

type OWNED_PARCEL is LAND_DIVISION
ownership(OWNED_PARCEL) → OWNER

define a **LAND_DIVISION** entity type having two functions and a subtype **OWNED_PARCEL** having an additional function. Because **OWNED_PARCEL** is a subtype of **LAND_DIVISION**, any entity of type **OWNED_PARCEL** is also an entity of the **LAND_DIVISION** supertype and automatically "inherits" the **description** and **area** functions. On the other hand, it is sometimes desirable that specialized versions of what appears to be the "same function" be available for different subtypes. For example, one might wish to provide a general **square_miles** function to compute the number of square miles in any 2-D shape, but have different specialized implementations for various representations of those shapes.

At the top of the generalization hierarchy, both entities and functions are members of the generic type **ENTITY**. In addition, the entity and function type definitions themselves are modeled as a collection of entities and functions, so that information in database definitions can be queried in the same way as database data.

Generic operations on objects (entities and functions), such as selection, function application, set operations, and formation of new derived function extends, have been defined in the form of an algebra [20] similar in some respects to the algebra defined for the relational data model. For the purposes of this paper, we discuss only two of the operations.

- **select(F, P)** works as in the relational algebra, selecting those tuples of **F** that satisfy the predicate **P**.
- **Apply-append(F, G)** is a function having all the arguments of **F** and **G**. **F** and **G** each have input arguments and output arguments. Semantically, the functions are composed by feeding the output of **F** into **G**. The result is similar to the natural join of **F** and **G** (natural join is an operation from the relational algebra).

Examples: The cities whose population exceeds 100 000 can be evaluated by **select(CITY**, population > 100 000). To obtain the area by each of these cities, apply-append can be used: **apply-append(apply-append(select(CITY**, population > 100 000), **shape)**, **square_miles)**.

B. PROBE's Architecture

We now describe the architectural ideas behind object-oriented database systems in general and then discuss PROBE's architecture in more detail. The architecture of an OODB has two main components:

- a *database system kernel*, a query processor designed to manipulate objects of arbitrary types, not just the numeric and string types of conventional database systems, and
- a collection of *object classes*, where an object class specifies the representation of objects of a new type and provides operations for manipulating objects of the type.

The data model of an OODB must provide a construct that represents object class functions, and the query language must provide a way for these functions to be invoked. EXODUS can handle this task in an application-specific way since customization of the data model and query language is supported. POSTQUEL generalizes QUEL by allowing, for example, predicates from object classes to be used in place of a fixed set of predicates [38]. In PROBE, (computed) functions are used to represent object class functions. These functions are invoked when referred to in certain PDM algebra operations.

The process of adding an object class to an OODB is not completely trivial. The designers of EXODUS describe their system as a *database generator* and discuss the need for a *database implementer* (DBI), a person who creates a database system for an application by adding object classes to EXODUS [6] (the DBI has other duties also). This is an accurate view of how any OODB would be used, and the PROBE and POSTGRES systems would also benefit from the services of a DBI since this process of customization requires expertise beyond what could reasonably be expected of an application programmer. The other duties of a DBI must depend on the particular OODB being customized.

A more detailed specification of OODB architecture will focus on the database system kernel and its relationship to the object classes. It is therefore necessary to examine the "division of labor" between these components. Given a query involving specialized application objects, what part of the query should take place in the database system kernel, and what should be expected of the object classes? Different answers to this question correspond to different database system architectures.

There is a range of possibilities for splitting up the processing of a query involving specialized types. One extreme position is that the database system kernel should handle the entire query, including the parts dealing with the specialized data types. In spatial applications, this would require specific notions of spatial data to be "hardwired" into the database system. This option sacrifices generality, so we reject it. The other extreme is to pass to the object classes the parts of queries that involve specialized data. The specialized data could be stored in the database system and passed to the object class with the query, or the data could be stored in a storage device "owned" by the object class. The problem with this approach is that the object class must be concerned with database implementation issues relating to the manipulation of large collections of objects. For example, in order to handle queries over a collection of spatial objects, the object class will have to contain an implementation of an efficient spatial data structure appropriate for secondary

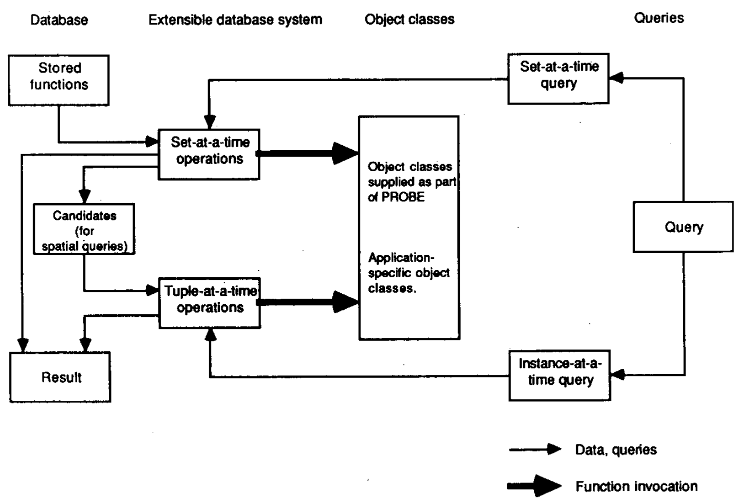

Database
Extensible database system
Object classes
Queries
Stored functions
Set-at-a-time operations
Candidates (for spatial queries)
Tuple-at-a-time operations
Result
Object classes supplied as part of PROBE
Application-specific object classes.
Set-at-a-time query
Query
Instance-at-a-time query
Data, queries
Function invocation

Fig. 1. Architecture of the PROBE database system.

storage, as well as concurrency control and recovery algorithms for the data structure.

We believe that an appropriate division of labor is for the database system to carry out suitably generalized basic database operations (e.g., joins, selection) on sets of generic objects, while each object class is responsible only for working on individual objects of more specialized types. (The operations described above, **select** and **apply-append**, are examples of the generalized database operations.) This division of labor corresponds well with a desirable division of labor among implementers—no implementer will be concerned with both database and application issues. For example, consider a proximity query: find all objects within a given distance of a given object. Instead of requiring an object class capable of processing this entire query, PROBE only requires the object class to provide a function that indicates whether a pair of objects satisfy the selection condition.

The decision to have the database system handle generic types and leave the more specialized types to the object classes raises the question of where the dividing line should be. The PROBE data model supports the notion of generalization. At the root of the hierarchy is the most generic type, ENTITY. Below this are types with more semantics, such as numeric types, string types, and spatial types. Application-specific types can be added. The dividing line was drawn so that PROBE supports only the most generally useful types: ENTITY and POINT-SET. POINT-SET is used to model all kinds of temporal and spatial objects (regardless of dimension and other application-specific concerns). POINT-SETs are described more fully in the following section.

An architecture based on this division of labor is shown in Fig. 1. It is an extensible architecture, and it supports a division of labor in which the database system kernel handles sets of generic objects, while the object classes handle individual objects of specialized types. The application program invokes database operations as usual. What is different under this architecture is that the invocation will refer to functions provided by the object classes. For example, an image database application built on top of PROBE might invoke a feature extraction function supplied by an IMAGE object class (see Section V).

C. Spatial and Temporal Data

Our approach to spatial and temporal data is highly compatible with the features of PDM presented above. Both spatial and temporal objects are modeled by mathematical abstractions called point sets. The *point set* of an object is the set of points in space occupied by that object. The discussion here is in terms of spatial data, but it applied equally well to temporal data. We view temporal data as one-dimensional spatial data [10].

A point set is a member of the POINT-SET type. This type is a specialization of the ENTITY type which introduces a collection of spatial operators such as union, intersection, and difference, as well as spatial predicates such as overlap, containment, and proximity. (These operations are discussed below.) Further specializations for POINT-SETs of specific dimensions, and for specific kinds of space (e.g., discrete or continuous), can be included as object classes. Since point sets are entities, functions that relate spatial/temporal and "conventional" entity types can be defined. Functions are also used to specify topological relationships between point sets, e.g., containment and adjacency. The modeling of spatial data has been described more fully in [19], [27].

In addition to dealing with POINT-SET entities as individual objects, there are many situations in which it is necessary to deal with POINT-SETs contained within other POINT-SETs. For example, a map feature might have a POINT-SET describing its shape. The POINT-SET for the containing map would have to contain all the POINT-SETs of features contained within the map. Similarly, POINT-SET entities that represent individual parts within an assembly may be grouped as components of the POINT-SET entity that represents the entire assembly. When we deal with a POINT-SET in its role as a container of other POINT-SETs, we refer to the container POINT-SET as a *space*. Since a POINT-SET contained in one space can itself contain other POINT-SETs, POINT-SETs naturally exhibit a hierarchical structure. We represent the hierarchical structure in the model by a set-valued **contains** function from the space to the spatial entities contained within the space. Multiple decompositions of the same set of points (such as a geographic area) can be defined using multiple POINT-SET entities denoting the same set of points.

Since entities of type POINT-SET are first-class PDM entities, they can be used as arguments of generic PDM functions in the same way as conventional PDM entities. In addition, specialized operations associated specifically with entities of type POINT-SET are defined. The operations provided for operating on generic POINT-SET entities fall into two categories: *point-set operations* and *structural operations*.

The point-set operations include set operations on POINT-SET entities, spatial selection, and overlay. The set operations intersection, union, and difference provide the primary means for combining POINT-SET entities into new POINT-SET entities. These operations are defined for entities $P1$ and $P2$ of type POINT-SET as follows. **POINT-SET_union**($P1$, $P2$) is an entity of type POINT-SET that denotes the set of points belonging to either $P1$ or $P2$ (or both). **POINT-SET_intersect** and **POINT-SET_diff** are defined analogously.

We introduce the following definitions to simplify notation in later sections. If S is the set of spatial objects $\{P_1, P_2, \cdots, P_{n-1}, P_n\}$ and X and Y are spatial objects, then

- $\cup S$ = **POINT-SET_union**(P_1, **POINT-SET_union**(P_2, $\cdots$, **POINT-SET_union**(P_{n-1}, P_n) $\cdots$)),
- $\cap S$ = **POINT-SET_intersect**(P_1, **POINT-SET_intersect**(P_2, $\cdots$, **POINT-SET_intersect**(P_{n-1}, P_n) $\cdots$)), and
- $X - Y$ = **POINT-SET_diff**(X, Y).

Also defined as point-set operations are special variants of generic PDM functions that are tailored to operate with POINT-SET entities. Specifically, predicates are added to functions such as selection that test various spatial conditions, such as whether a point set is empty, contains another point set, or intersects another point set. A whole range of other spatial relationships (e.g., "left-of," "above," and "adjacent-to") can be added in the same way.

Given a space containing objects that may overlap with one another, it is often useful to identify maximal subspaces that do not contain any object boundaries. For example, a crucial operation in geographic information systems is "polygon overlay." This operation superimposes two maps of the same area (e.g., land usage and political districts) and creates all the regions due to the intersection of regions from the input maps. PROBE's spatial data model includes an *overlay* operator to facilitate this kind of processing.

In discussing overlay, it is useful to have the concept of a uniform region. Let obj(p, S) be the set of objects in a space S that contain p, a point in S. Then a *uniform region* is a maximal subspace u in a space S such that, for every point p in u, obj(p, S) is the same. That is, u is a uniform region if, for all $p1$, $p2 \in u$, obj($p1$, S) = obj($p2$, S), and no subspace properly containing u has this property. To support operations such as polygon overlay, it is useful to be able to turn uniform regions into first-class objects. This is the finest partitioning that can be obtained given a set of objects (using only object boundaries to define partitions). Any coarser partitioning can be obtained by combining uniform regions (using **POINT-SET_union**). From the point of view of the data model, a space containing objects is indistinguishable from a space containing the uniform regions derived from a set of objects. They are both represented by a space containing spatial objects.

Based on this discussion of uniform regions, we can now define *overlay*: **overlay**(S) returns a space containing a spatial object for each uniform region of space S. The **overlay** operation can be used to compute polygon overlay as follows. Each input map is represented by a space containing a POINT-SET for each region of that map. The POINT-SETs from the two maps are placed in a single space by the **OB_union** operation (discussed below). The **overlay** operator is applied to the resulting space.

The point-set operations defined above form an algebra on point sets. As a result, given these operations, POINT-SET entities denoting complex shapes can be constructed by specifying algebraic expressions of point-set operations applied on POINT-SET entities.

The structural operations are concerned with the hierarchical structure of spaces described earlier. In general, these are convenient "macros," as they can be defined in terms of the nonspatial operators of the PDM algebra. The *object set operations* are defined for spaces $S1$ and $S2$ denoting the *same* point set, but having possibly different contained POINT-SETs (i.e., $S1$ and $S2$ register different information about a single point set). The definitions are as follows. **OB_union**($S1$, $S2$) is a space denoting the same point set as $S1$ and $S2$ that contains the set of objects contained in $S1$, $S2$, or both. (The objects in the result may not be spatially distinct, although their identities are retained.) **OB_intersect** and **OB_diff** are defined analogously.

The *expand* and *reduce* operators provide additional control over the CONTAINS relationship in spaces. If S is a space, X is in S's CONTAINS function, and Y is in X's CONTAINS function, **expand**(S) produces a space S' denoting the same point set, having moved Y into S's CONTAINS function without altering X's CONTAINS function. That is, for each immediate child X of S, **expand**(S) effectively copies each child of X so that it is also an immediate child of S. Note that placing an object in a new space (Y in the space of S in the above example) requires computation of the position of the object within the space. If the position is specified as a transformation, then a composition of transformations is necessary (e.g., multiplication of 4×4 matrices). **Reduce** is, in some sense, the inverse of **expand**. **Reduce**(S) produces a new space having no immediate children that are also contained in some other (immediate or indirect) child object of S.

Finally, since the substructure of a particular spatial representation is structured hierarchically, it is possible to use recursive processing techniques to search this hierarchical structure by traversing the **CONTAINS** relationship. PROBE's recursion operators (described in [30], [31]) can be used for this purpose.

III. Supporting the Spatial Data Model

Point-set entities and operations are useful in a wide variety of spatial applications. Point sets generalize spatial constructs found in many application areas, and the operations provided seem to be generally useful. However, in order to use PROBE in a spatial application, it is necessary to add object classes that provide specific representations for individual spatial objects and implement operations on instances of the representations.

Set-at-a-time spatial queries, e.g., selection with a spatial predicate, can be specified in PDM algebra. The processing of such queries is consistent with the division of labor discussed in Section II-B: PROBE processes sets of spatial objects and relies on the instance-at-a-time object class operations to manipulate individual spatial objects (e.g., to decide if a given object satisfies a spatial selection predicate). PROBE can support this approach to spatial data, but still provide good performance through the use of the *geometry filter*, a dimension- and representation-independent optimizer of spatial queries.

This section describes how the geometry filter and the object classes work together efficiently to support the point-set operations of PROBE's spatial data model. The structural operations do not require use of the filter. They can be supported using "ordinary" PDM algebra operations, and it is expected that more traditional query optimization techniques will be applicable. For spatial selection and overlay, the filter provides optimization of the query. For other geometric operations, e.g., the point-set set operations, the filter does not provide any optimization, but permits optimization of later operations. For example, the optimization of spatial selection on **POINT-SET_union**(A, B) is not possible unless the geometry filter's representation of **POINT-SET_union**(A, B) is available.

A similar approach to spatial query processing is taken in the PSQL project [32]. PSQL is an extension of SQL for "pictorial" data. The architecture of PSQL is very similar to PROBE's. Both systems advocate the division of labor discussed above. As a result, PSQL, like PROBE, can support a set of standard spatial operations that can be used with application-specific representations. (Abstract data types in PSQL are similar to object classes in PROBE.) The PSQL spatial query processor is based on the R-tree, a special-purpose spatial file organization [14]. The approach presented here is different. It is highly compatible with existing database system software and does not require the introduction of new file organizations.

A. Principles of the Geometry Filter

Many spatial queries (including the point-set operations of PROBE's spatial data model) can be expressed in terms of iteration over one or more collections of spatial objects and a function that considers each object or group of objects, invoked in the innermost loop. For example, in order to find all pairs of objects in a set S of spatial objects that overlap one another, the following algorithm can be used:

```
for each x in S
  for each y in S
    if overlap(x, y) then
      output(x, y)
```

(It is a simple matter to eliminate symmetric results (x overlaps y iff y overlaps x) and reflexive results (x always overlaps x).) This kind of algorithm is compatible with the proposed architecture since it embodies the desired division of labor. The database system implements the scans of S (this does not require any "understanding" of the objects belonging to S), while the spatial object class indicates whether individual objects overlap. This algorithm can be expressed in PDM algebra as follows:

```
result := select(cartesian-product(S, S), overlap)
```

(Here, and throughout this paper, details relating to the handling of argument names are suppressed. In this case, the spatial objects in the first appearance of S must be distinguished from those in the second through the renaming of function arguments.)

The problem with algorithms of this kind is one of performance. The nested loops lead to polynomial-time algorithms (whose degree is equal to the level of nesting). This will not be acceptable in practice, given that much more efficient special-purpose algorithms are often known.

PROBE's geometry filter is also compatible with the proposed architecture (refer again to Fig. 1) and provides much better performance, comparable to some of the best special-purpose algorithms [27]. The purpose of the filter is to optimize the nested loops that characterize brute-force geometric algorithms. The output from the filter is a set of *candidate* objects (or a set of groups of objects—one member of the group comes from each nested loop). Any object or group that is not included in the candidate set is certain not to satisfy the query. An object or group in the candidate set is *likely* to satisfy the query. The set of candidates will be refined to yield the precise answer by applying to each candidate a predicate from the supplied spatial object class (**overlap** in the example above).

The geometry filter contains algorithms that optimize spatial selection and overlay. As an example, consider spatial selection. Given two sets of spatial objects R and S, the *overlap query* returns all pairs of objects (r, s) such that r belongs to R, s belongs to S, and r and s overlap spatially. (This generalizes partial-match and range queries and can also be used to deal with a variety of containment and proximity queries.) This query would be processed by PROBE in two steps. First, a geometry filter algorithm, *spatial join*, would be invoked. This algorithm identifies candidates, pairs of objects (one from R and one from S) that are likely to overlap. Next, ordinary database operations would be used to refine the candidate set. These steps can be expressed in PDM algebra as follows:

```
candidates := spatial-join(R, S)
result := select(candidates, overlap)
```

This approach to spatial selection is consistent with the division of labor discussed in Section II-B. The database system kernel supplies the spatial join algorithm, which processes a collection of spatial objects (whose representation is not understood) to obtain candidate pairs. Each candidate is examined by the application-specific spatial object class. Of course, the spatial join algorithm must have *some* notion of spatial data in order to conclude that certain pairs of objects could not satisfy the query. Specifically, there must be a representation derivable from whatever representation is specified in the application-specific object class. The following subsections explain the principles behind the representation of spatial data used by the geometry filter. The construction of the representation is discussed in Section III-B. Sections III-C–III-E analyze each operation of the spatial data model. For each operation, the geometry filter algorithm is given, the interaction with the application-specific object class is described, and the requirements imposed on the object classes (for the geometry filter and application-specific representations) are described. Table I summarizes these results.

The requirements placed on the application-specific object class are minimal. As long as overlap, containment, union, intersection, and difference can be computed, the object class can be plugged in as a specialization of the POINT-SET object class, and spatial queries involving the object class can be optimized by the geometry filter.

1) Elements: The representation used by the geometry filter is conceptually a grid. A k-dimensional (k-D) point set in a k-D space is *approximated* by superimposing a k-D grid of cells on the space and noting which cells contain part of the point set (see Fig. 2). (In this section, we assume that the space contains a single point set. This assumption is relaxed in Section III-B). This approximation is *conservative* since the object being approximated is contained by the **POINT-SET_union** of the cells overlapping the object. While the presentation is in terms of 2-D data, all the ideas extend to higher dimensions (and to 1-D) without difficulty.

Algorithms for grid representations are often extremely

TABLE I
OBJECT CLASS REQUIREMENTS

Spatial operation	Application-specific object class operations
Creation of geometry filter representation	overlap, containment
Spatial selection	overlap, containment (for refining result)
Point-set set operations	union, intersection, difference
Overlay	union, intersection, difference

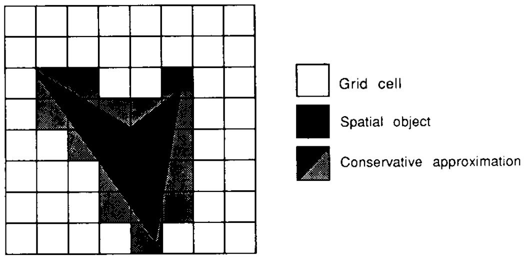


Fig. 2. Conservative approximation of a spatial object.

simple—the same action is repeated for each cell. However, the space and time requirements are very large for high-resolution grids. For this reason, the geometry filter uses an encoding of the grid. This encoding is obtained by recursively partitioning the space containing the object of interest. Partitioning continues until partitions that do not contain the object's boundary are obtained or until some maximum resolution is reached. (The resolution is that of the grid that is being encoded.)

In a k-D space, a partition is a $(k-1)$-D hyperplane perpendicular to the dimension being "split." The position and orientation of a partition are highly constrained, as described below. As a result of these constraints, the size, shape, and position of a partition can be described very concisely. Furthermore, the constraints impose a simplicity on the structure of the partitioning that can be exploited in geometry filter algorithms. (For example, if p and q are partitions, then p can contain q or q can contain p, but overlap other than containment can never occur.)

We will use the following notation: $< s_1 : n_1 | s_2 : n_2 | \cdots | s_m : n_m >$ denotes the string

$$\underbrace{s_1 s_1 \cdots s_1}_{n_1} \underbrace{s_2 s_2 \cdots s_2}_{n_2} \cdots \underbrace{s_m s_m \cdots s_m}_{n_m}.$$

If $n_i = 1$, then $s_i : n_i$ may be written as s_i. For example, $< 011 : 2 | 01 > = 01101101$.

The discussion will be simplified by assuming that the space is a grid of resolution $2^d \times 2^d$. Call the axes x and y. A cell in the grid is specified by providing two coordinates of d bits each: $(<x_0|x_1|\cdots|x_{d-1}>, <y_0|y_1|\cdots|y_{d-1}>)$.

A vertical split through the middle of this space amounts to discriminating on the value of x_0. In the left-hand half of the space, $x_0 = 0$; in the right-hand half, $x_0 = 1$. If the next split is horizontal, this corresponds to discrimination on the value of y_0. There are now four regions corresponding to the possible values of x_0 and y_0. Each additional split of a region creates two subregions characterized by the value of a specific bit from x or y. The sequence of splits defining a region can therefore be summarized by a corresponding sequence of these characteristic bits. If the direction of splitting alternates, then the sequence of bits corresponds to the interleaving of x and y bits. This interleaving creates a bit string that uniquely identifies a region. If r is a region obtained by splitting, then the corresponding bit string is the *z value* of r, denoted $z(r)$. (See Fig. 3.)

The z value of a region provides a concise description of the shape, size, and position of the region, as these attributes can be derived from the z value. In general, if the z value contains the first m bits of x and the first n bits of y, then the region described extends from $<x_0|\cdots|x_{m-1}|0:d-m>$ to $<x_0|\cdots|x_{m-1}|1:d-m>$ horizontally and from $<y_0|\cdots|y_{n-1}|0:d-n>$ to $<y_0|\cdots|y_{n-1}|1:d-n>$ vertically.

Partitions obtained by the splitting process described are called *elements*. These are the basic objects manipulated by the geometry filter. In general, a spatial object is represented by a collection of disjoint elements. The process of recursively partitioning a space to obtain elements is called *decomposition*. The decomposition algorithm appears below. Fig. 4 shows the elements generated in the decomposition of a spatial object.

```
function decompose(e: element, p: point-set,
                   result: list of element):
                   list of element;
begin
if contains(p, e) then
   Report-Element(e, result)
else if overlap(p, e) then
   if minimal(e) then
         Report-Element(e, result)
   else
         begin
         decompose(left(e), p, result);
         decompose(right(e), p, result);
         end;
(* else no overlap - nothing to do *)
return result;
end;
```

The algorithm is invoked with e bound to the single element representing the entire space. p is the object being decomposed, and *result* accumulates the elements generated. The **minimal** function returns true iff the input element is the smallest possible, i.e., if the resolution of the grid has been reached.

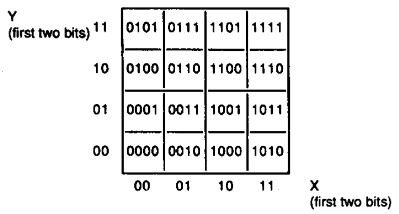


Fig. 3. The interleaving of bits from X and Y yields a bit string that characterizes regions obtained by recursive splitting.

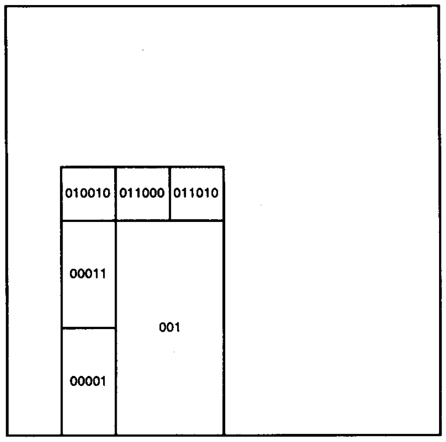


Fig. 4. Decomposition of a spatial object into elements. Each element is labeled with its z value. The z value is obtained by recording the relationship of the element to each split. If the element is to the left-hand side of or below a split, a 0 is generated; otherwise, a 1 is generated.

Recursive partitioning yields one-cell (i.e., minimal) elements if carried far enough. (The recursive calls of **decompose** may stop before minimal elements are reached. This occurs when a nonminimal element is contained in the object being decomposed.) The z values of the individual cells inside any element are consecutive. Furthermore, the lower left-hand and upper right-hand cells have the lowest and highest z values (respectively) inside the region. These z values are denoted $zlo(e)$ and $zhi(e)$ (see Fig. 5).

While the discussion so far has assumed that the direction of splitting alternates between x and y (corresponding to an $xyxy\cdots$ interleaving pattern), other patterns can be used. This issue is discussed in [27]. In the rest of this paper, we assume the $xyxy\cdots$ splitting pattern.

2) Z Order: Z values can be compared lexicographically, i.e., the bit strings are left-justified and then compared one bit at a time. Therefore, a collection of elements can be ordered by sorting lexicographically on their z values. Precedence in z order is denoted as $<_z$.

The spatial interpretation of this ordering is interesting. If each cell of a 2-D grid is treated as an element, then the z values of the elements trace out the path shown in Fig. 6. This ordering has been discovered many times [1], [5], [12], [17], [24], [28]. The oldest reference that we are aware of is to Peano in 1908 [29]. In [24], [25], we called it z ordering and we use that name here. The curve is recursive in that it consists of the same N shape (cov-

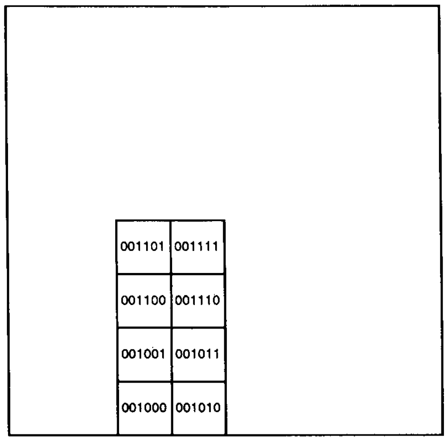


Fig. 5. The element whose z value is 001 contains all elements whose prefix is 001. The *zlo* and *zhi* values are in the low left-hand and upper right-hand corners, respectively.

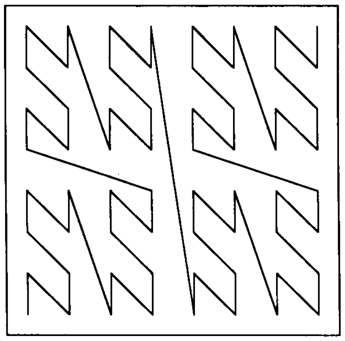


Fig. 6. Traversal of cells in z order. Compute the z value of each cell by interleaving the bits of the coordinates. For example, $(3, 5) = (011_2, 101_2) \rightarrow 011011_2$. Interpret the z values as integers and then "connect the dots."

ering four points) repeated throughout the space. Groups of four N's are connected in an N pattern, groups of four of these groups of four are connected in the same pattern, etc. (It is called z order because our first drawing of it had Z's instead of N's.)

Z order can be viewed as a mapping from multidimensional space to 1-D space which preserves proximity. That is, if two points are close in space, then they are likely to be close in z order. A practical benefit of preserving proximity is that elements are clustered for efficient access on secondary storage. Elements that are close in space are likely to be stored on the same disk page. (See [27] for more information on the preservation of proximity.) A curve related to z order can be obtained by a simple transformation of z values [11]. This alternative curve is compatible with the use of all algorithms to be described and improves on the preservation of proximity obtained with z order.

B. Creation of the Geometry Filter Representation

So far, we have seen how individual spatial objects (point-set entities) can be decomposed into z-ordered sequences of elements. However, PROBE's spatial data model allows spaces to contain multiple point sets. This section discusses the geometry filter representation of such a space and points out that this representation suggests algorithms based on familiar nonspatial data structures.

The geometry filter representation of a space is a z-ordered sequence of the elements from the decompositions of all the objects contained in the space. All that has to be done to obtain this representation is to decompose each object and merge the resulting z-ordered sequences into a single sequence. A *label* associated with each element identifies the object from which the element was obtained. In terms of PDM algebra, this construction can be achieved using the **apply-append** operation. For example, if the spatial objects representing geographical regions are provided by **shape(REGION) → POLYGON** and M is a map (a set of regions), then the geometry filter representation of M is obtained by **apply-append(apply-append(**M**, shape), decompose)**. By asking for an index to be constructed on the result, the required sorting (and indexing) will be performed.

During the decomposition of an object, the **decompose** function checks the relationship (overlap, containment, or disjointness) between elements and the object being decomposed. The **overlap** and **contains** functions invoked *directly* by **decompose** are generic. Each application-specific spatial object class provides implementations of these operations that are written specifically for that object class. PROBE senses the type of the object being decomposed and invokes the appropriate inplementation. (For this reason, the geometry filter is general enough to deal with multiple representations within the same query.) Since the type-specific **overlap** and **contains** predicates expect two arguments of the same type, the boxes corresponding to elements (or conservative approximations thereof) must be describable in the same representation as that of the object being decomposed. The derivations appear to be simple in practice. The code to derive the application-specific representation is not part of the geometry filter, as this would limit the generality of the filter.

With the ideas presented so far, it is possible to sketch the spatial join algorithm. Recall that the inputs to spatial join are two functions R and S storing spatial objects. Internally, stored functions would be represented using a B-tree (or other conventional file organization) keyed by z value. Besides the z value, each record carries a label identifying the spatial object from which the z value was obtained. A merge of the two z-ordered sequences can be performed to locate cases in which a z value from R contains, or is contained by, a z value from S. This indicates that the corresponding spatial objects are likely to overlap spatially, so a candidate pair has been identified. (See Fig. 7.)

The time required for the merge is proportional to the number of elements representing the objects in the two spaces. The number of elements arising from the decomposition of an object can be controlled in a number of

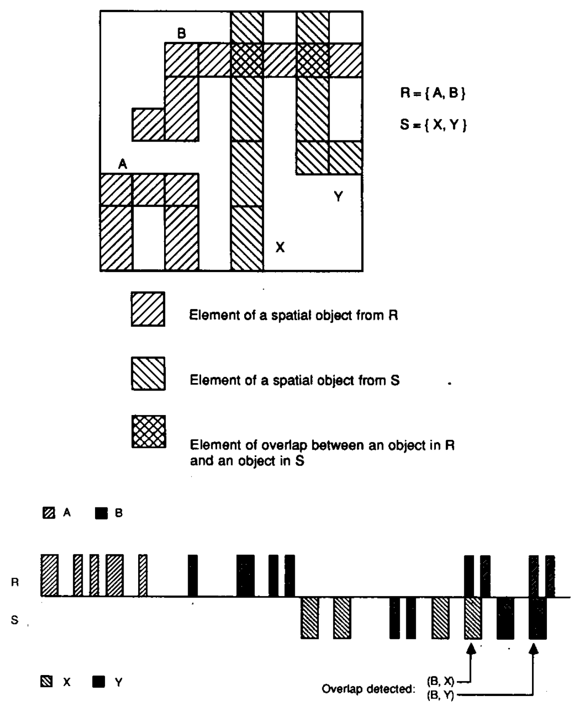

B
A
Y
X
R={A,B}
S={X,Y}
Element of a spatial object from R
Element of a spatial object from S
Element of overlap between an object in R and an object in S
A
B
R
S
X
Y
Overlap detected: (B, X) (B, Y)

Fig. 7. The overlap query can be processed by merging z-ordered sequences of elements. Overlap is detected where an element from R contains an elements from S or vice versa.

ways (e.g., by selecting the resolution of the space), so the time is actually linear in the number of objects (and the constant can be tuned to optimize the tradeoff between precision and time for the testing of candidates). This is clearly superior to the nested loops algorithm presented in Section III-A. Furthermore, it will be seen in the following section that *sublinear* performance can be obtained.

C. Spatial Join

As discussed in the previous section, spatial join can be implemented by a merge of z-ordered sequences of elements. In this section, we discuss the implementation and optimization of this merge. Before getting into the details, it will be useful to summarize the results. First, with the optimization, sublinear performance is obtained. A special case of the overlap query is the range query (useful in traditional database applications). The spatial interpretation of the range query is to find all the data points in a given box. It is shown in [27] that, for the range query, the performance obtainable with the geometry filter is comparable to that of the best special-purpose data structures, such as the k-D tree [4]. Second, the data structure requirements are minimal: *any data structure that supports random and sequential accessing can be used by all the geometry filter algorithms*. This is important since it allows the use of existing file organizations such as B-trees (for secondary storage), and binary or AVL trees (for main memory), which are in common use in current database systems and elsewhere. There is no need to develop new special-purpose file organizations for spatial indexing. Other requirements are also compatible with existing database system software: z values can be represented by integers. Therefore, existing sort utilities can be used to create z-ordered sequences of elements. Also, the LRU buffering strategy is exactly what is called for in the implementation of the merge.

The following discussion refines the ideas presented in [24]–[27].

The merge logic of **spatial-join** is somewhat unconventional because elements represent intervals and because the elements within a sequence may exhibit containment. (When containment occurs, the elements involved are from different objects in the space.) Fig. 8 demonstrates some of the difficulties. In the example, the spatial join would report the candidate pairs (A, D), (A, E), (A, F), (B, E), (B, F), (C, H), and (C, I). The interaction between sets of nested elements ($\{A, B\}$ and $\{E, F\}$ in the example) is especially tricky to handle correctly.

The spatial join algorithm depends heavily on the mathematical properties of elements. First, the only possible relationships between elements are precedence in z order and containment. Overlap (other than containment) cannot occur. Second, in a sequence of elements ordered by *zlo*, nested elements are consecutive, with the larger elements appearing before the smaller elements. This is because of the relationship between containment and z values. If x and y are elements, then x contains y iff $z(x)$ is a prefix of $z(y)$. The only way an element could appear between x and y in the lexicographic ordering would be for it also to be part of the nesting, between x and y.

In order to keep track of a set of nested elements from a sequence, a stack is associated with each sequence. This stack is called a *nest*. The nest is updated as the input sequence is scanned. At any point in the merge, the top of a nest stores the "current" element from the corresponding sequence, and containing elements (from the same sequence) are stored deeper in the stack. The order of elements in the stack reflects the nesting of the elements. Previously examined elements that do not contain the current element are not present in the nest (see Fig. 9).

The state of each input sequence is defined by the sequence itself (including a cursor to indicate the next input element) and by the nest. The **state-of-input** data type is used to encapsulate this information. In each iteration of the merge, the state of one sequence is updated by advancing the cursor (by at least one position) and updating the nest.

Even though there are only two input sequences, the merge is conceptually four-way. As a sequence of elements is scanned during the merge, elements are "entered" and "exited." Element e is entered at $zlo(e)$ and exited at $zhi(e)$. Therefore, there are four kinds of events that occur (the input sequences are X and Y):

1) entering an element from sequence X,
2) exiting an element from sequence X,

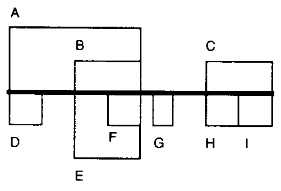


Fig. 8. In order to compute spatial join, the overlap between A and D, A and E, A and F, B and E, B and F, C and H, and C and I must be detected during the merge.

3) entering an element from sequency Y, and
4) exiting an element from sequence Y.

Each kind of event constitutes an input to the merge, and hence the merge is four-way. An entering element comes from a sequence (at the position indicated by the cursor), while an exiting element comes from a nest. When an element is entered, it is placed on the nest. Eventually, the element will be exited, and (for a set of nested elements) the order of exiting is opposite to the order of entering. Because of this last-in first-out behavior, it is appropriate to use stacks to represent nests.

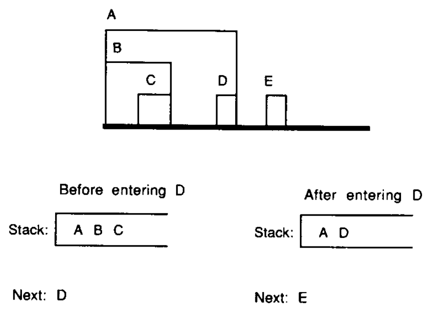


Fig. 9. State of the input sequence before and after entering an element.

The details of the algorithm are as follows:

```
function spatial-join(left-sequence, right-sequence: list of labeled-element): list of candidate;
var
    result: list of candidate;
    L, R: state-of-input;
    event: element;
begin
L := initial-state-of-input (left-sequence);
R := initial-state-of-input (right-sequence);

while not(eof(L.sequence) and empty(L.nest) and
        eof(R.sequence) and empty(R.nest))
    begin
    event := min(zlo(current(L)), zhi(top(L.nest)),
                zlo(current(R)), zhi(top(R.nest)))
    if event = zlo(current(L)) then
          enter-element(L, R, result)
    else if event = zhi(top(L.nest)) then
          exit-element(L, R, result)
    else if event = zlo(current(R)) then
          enter-element(R, L, result)
    else (* event = zhi(top(R.nest)) *)
          exit-element(R, L, result);
    end;
  return result;
end;

procedure enter-element(X, Y: state-of-input; result: list of candiate);
begin
while not contains(top(X.nest), current(X))
    pop(X.nest);
push(X.nest, current(X));
advance(X, Y);
end;

procedure exit-element(X, Y: state-of-input; result: list of candidate);
begin
report-pairs(top(X.nest), Y.nest, result);
pop(X.nest);
end;
```

Spatial-join begins by setting up an initial **state-of-input** for each sequence. This is done by setting the cursor of each sequence to the beginning of the sequence and by pushing an element representing the entire space onto an empty nest. Although it is not strictly necessary to push this element, it will simplify the algorithm later. The loop of **spatial-join** implements the merge logic. In each step, the four inputs to the merge are checked for the next "event." The *zlo* and *zhi* extract the start and end points of an element, respectively. The four-way "if" statement detects which event has occurred. The event is either an entry or an exit for an element from the left or the right sequence. The procedures invoked (**enter-element** and **exit-element**) take care of "advancing" the input.

In **enter-element,** X represents the sequence that contributed the element being entered (either L or R). Y represents the other sequence. First, *X.nest*, the stack associated with sequence X, is updated. In order to maintain the property that stack elements are nested, it may be necessary to pop some elements. For example, in Fig. 9, it is necessary to pop the stack twice before the current element, **current**(X), can be pushed. This loop is guaranteed to terminate before the nest is empty because of the element representing the entire space, placed at the bottom of the stack. The last step is to advance the input sequence of X. The procedure for doing this is discussed below.

In **exit-element**, X represents the sequence that contributed the element being exited (either L or R). Y represents the other sequence. **Report-pairs**(**top**(*X.nest*), *Y.nest*, *result*) appends to the list of results pairs of the form (u, v) where u = **top**(*X.nest*) and v is an element in *Y.nest*, for all elements of *Y.nest* (except the bottom element, which represents the whole space—that element was inserted "artificially"). Popping *X.nest* represents the advancing of the input sequence (recall that the nests provide "input" of exiting elements).

The advancing of the input sequence (in **enter-element**, using **advance**) must be handled carefully. A correct implementation is simply to advance to the next element of the sequence. This would mean that the spatial join requires every element of every sequence to be examined. This is normal for a merge, whose running time is proportional to the sum of the lengths of the input sequences. However, it is possible to do better. Spatial queries are often localized in one region of the space, and it should not be necessary to examine every element. Ideally, it would be possible to ignore parts of the space that are "obviously" not going to be fruitful. This can be done by advancing the input seqeunce by more than one element at a time.

In order to do this, we will require sequences to be stored in data structures that support both random and sequential accessing. This is a very modest requirement since, as noted above, many data structures have this property. We will assume the following operations:

- **randac(s, k)**: random access to sequence s, using search key k; if the search fails, then the record with the smallest key greater than k is located; and
- **seqac(s)**: sequential access on sequence s.

Advance(X, Y) finds the first labeled element of sequence X past **current**(X) that could possibly be relevant (i.e., overlap with an element from Y). The procedure begins by advancing to the next element in X following **current**(X). (The current and next elements are denoted by *Xcurrent* and *Xnext*, respectively.) The procedure ends here, not advancing further, if *Xnext* is contained by an element from Y that has already been examined. Such an element can be found in *Y.nest.* (See Fig. 10(a) and (b).) It would be incorrect to advance further because the overlap between *Xnext* and the relevant Y objects would be missed.

On the other hand, if *Xnext* is not covered by any Y element, then it is not relevant. The only thing that is certain at this point is that the next relevant X element ends past the beginning of *Ycurrent*, the current element of Y [see Fig. 10(c)]. *Xnext, and all subsequent elements until the next relevant X element, can be skipped.* Due to the fact that elements may exhibit containment, but never overlap, there are now two possibilities: 1) that a random access in *X.sequence* using *Ycurrent* will locate the first X element contained by *Ycurrent* [as in Fig. 10(c)] and 2) that the random access will locate the first X element that begins after the end of *Ycurrent* [as in Fig. 10(d)]. Remember that the position of an element within a sequence is based on the beginning of the interval *zlo*, not the end, so in the second case, the random access may actually miss a relevant element of X. If this occurs, the correct element can be located by stepping back one position in the X sequence.

Now, either an element containing or contained by *Ycurrent* has been found, or such an element does not exist. In the former case, it is still possible that there are larger containing elements [as in Fig. 10(d)]. The largest such element (that begins after *Xcurrent* begins) must be located (there are a variety of methods for doing this). In the latter case (in which *Ycurrent* does not interact with any X element), the X element located by the random access is the proper place to resume the X sequence [see Fig. 10(e)].

D. Point-Set Set Operations

Set operations are commonly implemented by merging, and this is how the point-set set operations are implemented in the filter. Many of the peculiarities of the merge used for **spatial-join** are applicable here, and for the same reason: sequences of elements are being merged. The merge logic is, in fact, identical. The merge is four-way, and the events detected are entry to and exit from elements. All that differs are the actions taken for each event. Therefore, it is possible to describe the implementation of **POINT-SET_union** and **POINT-SET_intersect** by specifying the **enter-element** and **exit-element** algorithms, using the spatial join algorithm without modifi-

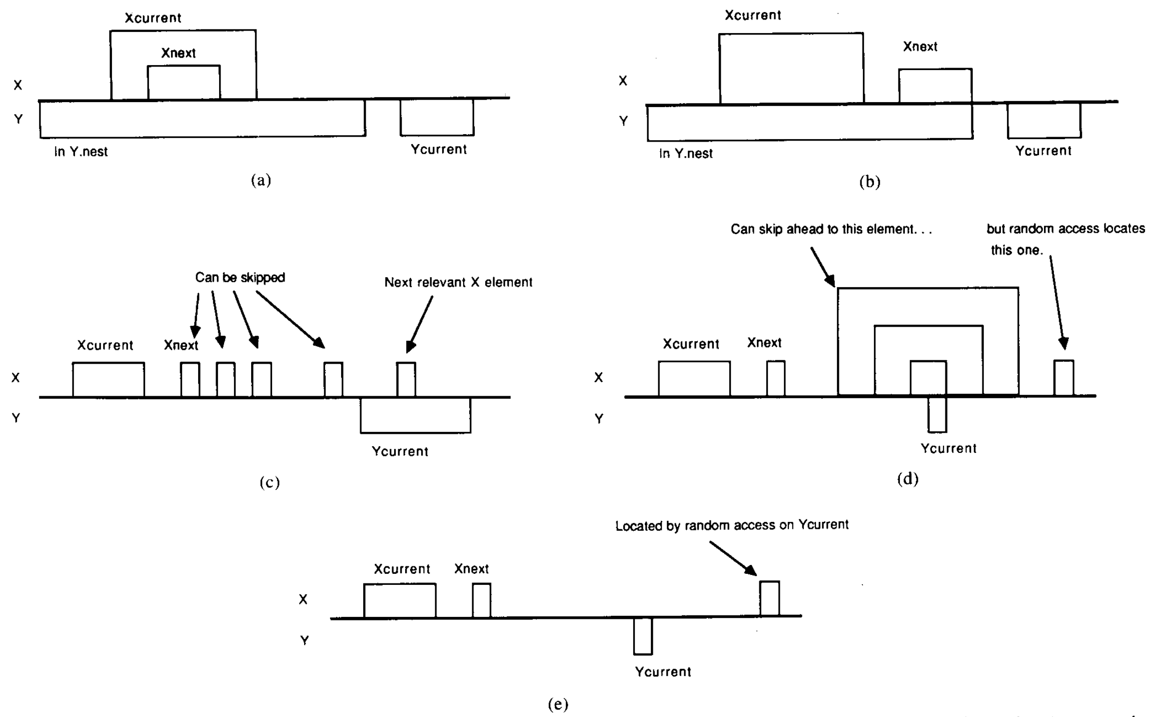


Fig. 10. (a) *Xnext* is contained by an element in *Y.nest*. (b) *Xnext* is contained by an element in *Y.nest*. This is the same situation as in (a), except the *Xnext* is after *Xcurrent*, not inside it. (c) Random access in *X.sequence* using *Ycurrent* locates next relevant element in *X*. (d) Random access misses elements that cover *Ycurrent* but begin before *Ycurrent*. Stepping back by one position in *X* locates an element containing *Ycurrent*, if there is one. (e) If there is no element containing or contained by *Ycurrent*, advance to the element located by the random access.

cation. **POINT-SET_diff** can be implemented on top of **POINT-SET_intersect**.

For the set operations, each input sequence contains elements comprising a *single* point set. For this reason, elements within one input sequence exhibit the precedence relationship only. Containment arises only between elements from different point sets. This allows some simplification of the algorithms. In particular, because containment within a sequence cannot occur, a stack is no longer needed to keep track of nested elements. However, for ease of presentation, we describe the algorithms as derivatives of **spatial-join**.

These algorithms show how set operations can be computed directly from geometry filter representations. An alternative is to first carry out the set operation on the application-specific representation and then to decompose the result. Performance considerations will dictate which approach is preferable.

1) POINT-SET_union: The problem in computing **POINT-SET_union** is to combine adjacent elements from opposite sequences into maximal intervals. A global variable *I* is used to keep track of the endpoints of the interval under construction. When an element from either sequence is entered, either the element is adjacent to *I* or it is not. In the former case, *I* is extended. In the latter case, *I* is maximal, so the elements of its decomposition are added to the result, and a new interval is started. The endpoints of the new interval are those of the current element. Following the update of *I*, the next relevant element in *each* input sequence is the one starting after the end of *I*. These can be located using random access.

It is simplest to detect adjacent and maximal intervals in the **enter-element** algorithm. The code appears below. Nothing needs to be done by **exit-element**.

```
procedure enter-element(X, Y: state-of-input,
        result: list of element);
var
        X, Y: state-of-input;
        result: list of element;
begin
(* I is an interval with components lo and hi *)
if adjacent(I, current(X)) then
        I.hi := zhi(current(X))
else
        begin
        report-elements(decompose(I), result);
        I.lo := zlo(current(X));
        I.hi := zhi(current(X));
        end;
randac(X.sequence, I.hi + 1);
randac(Y.sequence, I.hi + 1);
end;
```

2) POINT-SET_intersect: This algorithm is very similar to spatial join. As with spatial join, and unlike **POINT-SET_union**, the only cases of interest are those in which an element from one sequence contains an element from the other sequence. However, instead of reporting the pair of elements, only their intersection is of interest, i.e., the smaller of the two (since overlap other than containment cannot occur). Because of the resemblance to **spatial-join**, the call to **advance** is preserved. Since the modifications are so minor, the algorithms for **enter-element** and **exit-element** are omitted.

3) POINT-SET_diff: There is a subtlety involved in the computing of **POINT-SET_diff** relating to the correctness of the result. In order to retain the precision of a filter (i.e., that positive results are not lost), each operation must return a conservative approximation of the exact result. To preserve this property through a **POINT-SET-_diff**, some care must be taken with the boundary elements, those elements containing the boundary of the decomposed spatial object.

If P and Q are point sets, P' is the **POINT-SET_union** of the elements of **decompose**(P), and Q' is the **POINT-SET_union** of the elements of **decompose**(Q), then P' is a conservative approximation of P and Q' is a conservative approximation of Q. Furthermore, **POINT-SET_union**(P', Q') and **POINT-SET_intersect**(P', Q') are conservative approximations of **POINT-SET_union**(P, Q) and **POINT-SET_intersect**(P, Q), respectively (see Fig. 11). This means that sequences of operations consisting of invocations of **POINT-SET_union** and **POINT-SET_intersect**, carried out on both the application-specific and the geometry filter representations, will yield a geometry filter representation that retains the filtering property. However, **POINT-SET_diff**(P', Q') is not, in general, a conservative approximation of **POINT-SET_diff**(P, Q). The difficulty is that **POINT-SET_diff**(P, Q) = **POINT-SET_intersect**(P, **complement**(Q)) (where the complement is with respect to the containing space) and that **complement**(Q') is not a conservative approximation of **complement**(Q) (see Fig. 12). Further analysis shows that the problem is due to boundary elements. The decomposition of Q and the decomposition of **complement**(Q) have some common boundary elements, yet the complement of the decomposition of Q' does not include these elements. Stated more concisely, decomposition (followed by the **POINT-SET_union** of the elements) and **complement** do not commute: **POINT-SET_union**(**decompose**(**complement**(x))) $\neq$ **complement**(**POINT-SET_union**(**decompose**(x))).

To compute the conservative approximation of **POINT-SET_diff**(P, Q), compute **POINT-SET_intersect**(P', **complement***(Q')) where **complement***(Q') is **complement**(Q') plus sufficient elements to guarantee that the boundary of Q is contained in the result. (See Fig. 12.)

To compute **complement***(Q'), first compute **complement**(Q'). Then, for each element of **complement**(Q'), generate the decomposition of the point-set obtained by expanding the element in all directions by one cell. Sort this collection of elements by the z value and then, in one pass, discard duplicates and merge smaller elements into larger ones (this is analogous to what is done in **POINT-SET_union**). It is unfortunate that this implementation is superlinear (due to the sort), but we know of no other solution.

Expansion by one cell is correct, but is likely to generate a large number of elements in the following decomposition. For performance reasons, a greater expansion can be used (e.g., using two or four cells), but as usual, there is a tradeoff with precision. This expansion is expensive (requiring a sorting step), and it may be cheaper to decompose the complement of the application-specific representation.

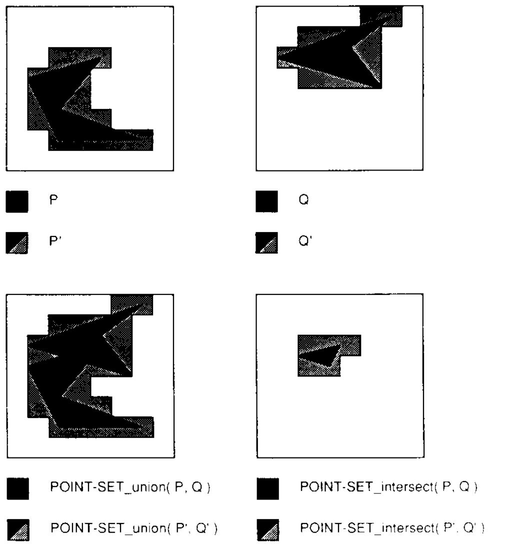

■ P
▨ P'
■ Q
▨ Q'
■ POINT-SET_union(P, Q)
▨ POINT-SET_union(P', Q')
■ POINT-SET_intersect(P, Q)
▨ POINT-SET_intersect(P', Q')

Fig. 11. P' and Q' are conservative approximations of P and Q, respectively. **POINT-SET_union**(P', Q') is a conservative approximation of **POINT-SET_union**(P, Q), and **POINT-SET_intersect**(P', Q') is a conservative approximation of **POINT-SET_intersect**(P, Q).

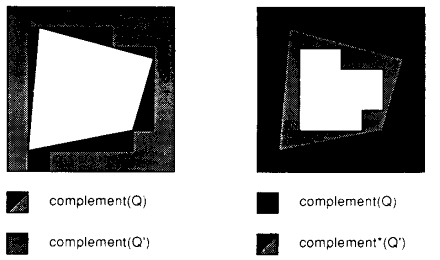


Fig. 12. **Complement**(Q') is not a conservative approximation for **complement**(Q), but **complement***(Q') is.

E. Overlay

A space of n objects can have as many as 2^n uniform regions, one for every member of the power set of the n

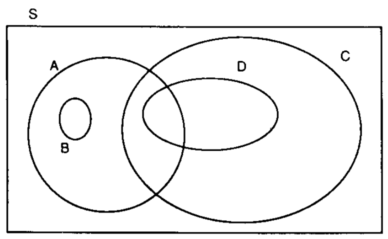


Fig. 13. Overlay of a space. The candidate sets found by the geometry filter are $\{A, B\}$ and $\{A, C, D\}$.

objects. (Uniform regions were defined in Section II-C.) An obvious brute-force algorithm is to enumerate the power set of the objects in the space and to compute the mutual intersection of each combination of objects. In geographical applications, overlays involving hundreds and even thousands of 2-D objects are computed routinely, so it is clearly the case that, in practice, the vast majority of the combinations yield regions that are empty or not uniform (and therefore not of interest) and are not even considered. (Otherwise, the algorithm would still be running.)

Compared to spatial join, support for overlay is more complicated, but conceptually there is nothing different: the filter returns candidate sets of objects. The set of candidates is a subset of the power set of the space. The objects within a candidate set are likely to interact with one another to produce (nonempty) uniform regions. If no candidate set contains a given set of objects G, then the objects in G do not interact to produce nonempty uniform regions.

Computationally, it is (almost always) far more efficient to enumerate the power set of each candidate than it would be to enumerate the power set of the entire space. This is not the most useful comparison since enumeration of the power set of the space is not feasible anyway. A performance comparison of our overlay algorithm with algorithms in use has not yet been attempted. One clear advantage of our algorithm is its generality. It is valid for all dimensions and for all representations. The application-specific spatial object class only has to supply a function that computes the overlay of two objects. This is simple once the point-set set operations have been supplied: the overlay of A and B comprises the three objects $A \cap B$, $A - B$, and $B - A$.

The geometry filter's overlay algorithm has two steps.

1) Identify candidate sets, groups of objects that are likely to overlap spatially.
2) Compute the uniform regions due to the overlap of objects within each candidate.

The following two subsections discuss these steps in detail. Overlay takes as input a set of objects within a single space. The geometry filter representation of this space is a z-ordered sequence containing all the labeled elements from the decompositions of all the objects in the space. The first step can then be accomplished in a single pass over the z-ordered sequence as described in Section III-E1. The output is a collection of candidate sets. In Fig. 13, B and C do not appear together in any candidate, indicating that B and C do not overlap anywhere in the space. Certain candidates are subsumed by others and so do not have to be considered explicitly. For example, $\{A, C\}$ does not have to be considered because all interactions between A and C will be examined when candidate $\{A, C, D\}$ is considered. The second step, described in Section III-E2, discusses the generation of uniform regions from the candidate set.

1) Finding Candidate Sets: The **overlay** algorithm takes as input a space S and returns a set of candidates. A candidate is a subset of objects in S that is likely to generate nonempty uniform regions.

S is represented by a z-ordered sequence of labeled elements. To simplify termination of the algorithm, assume that S is terminated by an element whose z value is larger than any other in S. As with spatial join, it will be useful to record the status of the scan of S using the **state-of-input** type. During the scan, the *nest* field of **state-of-input** stores elements that have been entered but not exited. The spatial objects corresponding to these elements (which can be located via the labels) are likely to overlap. Therefore, the candidates are located by sampling the stack at "appropriate" times.

One possibility is to sample the stack upon entering each element. However, this is more frequenct than is necessary. Suppose that the stack has the labeled elements $[A, B, C]$ (these are the labels). When the next element, with label D, is entered, there are two possibilities. If C's element contains D's element, then the stack becomes $[A, B, C, D]$. Since $[A, B, C, D]$ subsumes $[A, B, C]$ (as discussed above), it is not necessary to report $[A, B, C]$. If, on the other hand, D's element is not contained by C's element, then it is necessary to pop the stack at least once before pushing D. The result is $[A, B, D]$ (assuming that B's element contains D's). In this case, the candidate $[A, B, C]$ is relevant since C is missing from the updated stack.

The **overlay** algorithm operates by sampling the stack when its size reaches a "local maximum," i.e., just before it is popped to accommodate a new element. At this moment, the top of the stack corresponds to some region of the space that is maximal in terms of occupancy. If the new element can be accommodated without popping the stack, it must be true that the current stack has not reached a local maximum; i.e., "moving" into the new element does not result in the leaving of any objects already represented in the stack.

The complete algorithm is as follows:

```
function overlay(S: state-of-input):
set of candidate;
begin
    result: set of candidate;
    e:  labeled-element;

reset(S.sequence);   (*Start at the beginning*)
                     (*of the sequence*)
clear(S.nest);  (*Start with an empty nest*)
e := next(S.sequence);   (*Get next*)
                         (*element and*)
                         (*advance the*)
                         (*sequence*)
push(S.nest, e);

while not end-of-file(S.sequence) do
    begin
    e := next(S.sequence);
    if not(top(S.nest) contains e) then
        (*At a local maximum*)
        begin
        add(result, nest(S));
        while not(top(S.nest) contains e) do
                pop(S.nest);
        end;
    push(S.nest, e);
    end;

return result;
end;
```

2) Generating the Uniform Regions: The generation of uniform regions from candidates is implemented jointly by the geometry filter and the application-specific spatial object class. The geometry filter provides control and management of intermediate results, as will be described. The object class provides the intersection and difference operations (which are also required to support the set operations). Suppose, in the example of Fig. 13, that candidate $\{A, C, D\}$ is processed first. The geometry filter uses the object class to compute $A \cap C$, $A - C$, and $C - A$. Next, each of these is combined with D to obtain $A \cap C \cap D$, $(A \cap C) - D$, $(A \cap D) - C$ (which is empty and therefore discarded), $A - (C \cup D)$, $(C \cap D) - A$, and $C - (A \cup D)$.

When candidate $\{A, B\}$ is processed, the uniform regions B and $A - (B \cup C \cup D)$ will be computed. The overlay algorithm (running in the kernel) understands that $A - B$ is not uniform because pieces of A have already been "removed" to contribute to other uniform regions. Therefore, it is appropriate to substitute $A - (C \cup D)$ for A in $A - B$, yielding the correct result.

In general, a uniform region can always be described by an expression of the form $\cap X - \cup Y$ where X and Y have no objects in common. The resulting uniform region is characterized by X. In a given space S, the uniform region corresponding to X is $\mathrm{UR}(X) = \cap X - \cup$**OB_diff**$(S, X)$. Other expressions may produce equivalent results. For example, in Fig. 13, $\mathrm{UR}(A)$ is described by $\cap \{A\} - \cup \{B, C, D\}$, but also by $\cap \{A\} - \cup \{B, C\}$. (Note that the characteristic part $\{A\}$ is the same in both cases.)

Unfortunately, it is not always possible to compute a uniform region by processing a single candidate. In Fig. 13, $\mathrm{UR}(A)$ requires the processing of both candidates. We therefore take the following approach. *$UR(X)$ is obtained by computing a series of approximations, the last of which is exact: $UR_0(X), UR_1(X), \cdots, UR_n(X) = UR(X)$.* Each new approximation reflects the processing of a candidate, and $\mathrm{UR}_i(X) \supseteq \mathrm{UR}_{i+1}(X)$, $0 \le i < n$. It is not necessary to keep track of each approximation; only the most recent approximation of each uniform region is needed. For this reason, we introduce a set of variables: $\mathrm{UR}^*(X)$ is the most recent approximation of $\mathrm{UR}(X)$.

The complete algorithm appears below. It is consistent with PROBE's architecture—the object class never has to deal with more than a fixed number of objects at a time (two in this case).

```
function uniform-regions(candidates:
set of set of point-set): set of point-set;
var
  c, u: set of point-set;
  include, exclude: point-set;
  UR*: table of point-set; (*indexed by*)
                           (*set of point-set*)
  result: set of point-set;
begin
clear(UR*);

for each c ∈ candidates
  for each u ∈ power-set(c)
    begin

      if UR*[u] is undefined then
      (*This combination of objects has not*)
      (*been encountered in the processing*)
      (*of previous candidates. *)
      (*Form the initial approximation.*)
      UR*[u] := ∩ u;

    include := UR*[u];

    if not empty(include)
      begin
      exclude := ∪ OB_diff(c, u);
      UR*[u] := include - exclude;
      end;
    end;

  result := {};
```

```
    for each u ∈ UR*
        if not empty(u)
            insert(result, u);

    return result;
end;
```

The evaluations of ∩ and ∪ can be optimized by storing intermediate and final results, reusing them whenever possible. For example, if $\cap\{A, B, C\}$ is computed as **POINT-SET_intersect**(**POINT-SET_intersect**(A, B), C), then $\cap\{A, B, D\}$ can be optimized by reusing $\cap\{A, B\}$.

The size of an input space occurring in practice is such that it is reasonable to represent c and u, using bit strings. This simplifies and optimizes many aspects of the algorithm: iteration through the power set of c is particularly simple, the computation of **OB_diff**(c, u) just requires pairwise bit operations on the bit string, and finally, the bit string is amenable for use as an index into the UR* table.

IV. Example of an Image Database Application

The PROBE data model, the spatial operations supported, and the spatial query processor provide general support for applications involving spatial data. This section shows how these tools can be used to support an image database application. This example has not been implemented, but a geographic application has been implemented on top of a "breadboard" implementation of PROBE [13].

The schema for this example describes images and features derived from those images. The extraction of features is accomplished automatically or semiautomatically by an IMAGE object class. Regardless of how this is accomplished, the set of features in an image can be accessed by a function of the schema. Once the features are extracted and classified, queries involving these features can be evaluated.

```
type IMAGE is ENTITY
    pixels(IMAGE, X, Y) → PIXEL
    place(IMAGE) → BOX(*Bounding box*)
                    (*giving*)
                    (*bounding latitudes*)
                    (*and longitudes.*)
    time(IMAGE) → TIME(*When the image*)
                    (*was taken*)
    frequency(IMAGE) → FLOAT (*spectral*)
                        (*band*)
    feature (IMAGE) → set of FEATURE
                    (*Set of notable*)
                    (*features,*)
                    (*extracted*)
                    (*by an image*)
                    (*interpreter*)
    .
    .
    .

type FEATURE is ENTITY
    entity-type(FEATURE) → FEATURE-TYPE
    location(FEATURE) → (LATITUDE,*)
                        (*LONGITUDE*)
                    (*Real-world coordinates*)
    occurrences(FEATURE) → set of
    (IMAGE, X, Y)
                    (*Describes occurrence of*)
                    (*a feature in each image*)
                    (*containing the feature,*)
                    (*and gives the position *)
                    (*of the feature within *)
                    (*the image.*)
    near(FEATURE) → set of FEATURE
                    (*A set of nearby features*)
    .
    .
    .

type ROAD is FEATURE
    name(ROAD) → STRING
    crosses(ROAD) → set of ROAD
    length(ROAD) → REAL
    .
    .
    .

type BUS-STOP is FEATURE
    buses(BUS-STOP) → BUS-LINE
    .
    .
    .
```

The database stores images in whatever representation is specified by the IMAGE object class, presumably an array of pixels. The individual pixels can be accessed via the database system (as is done in [19]); however, it is expected that most of the processing of images would be carried out by IMAGE object class functions. Using **apply-append**, a fully or partially automated image interpreter is invoked through the **features** function. Alternatively, **features** could be a stored function that is populated when images are brought into the database system. The image interpreter analyzes an image and returns a set of FEATURE entities. Classification of features (e.g., placing a ROAD feature in the extent of the **road** function) is also performed by **features**. Other PDM algebra operations permit comparisons of the features found to features already in the database. Features not seen before will get new surrogates. For all features, old or new, the association between the image and the feature is noted in the **occurrences** function of the FEATURE object class.

A database described by this schema could be used to answer a query such as the following: What bus lines have stops within 100 yds. of Massachusetts Avenue and Tremont Street? One strategy for evaluating this query is the following.

• Using the **occurrences** function, the set of images containing Massachusetts Avenue and the set of images containing Tremont Street are located. Pick one of the images containing both of these roads (e.g., the most recent).

- Using **POINT-SET_intersect**, find the position of the intersection of the two roads. A 100 yd "buffer" around the intersection is then constructed.
- Find an image that (spatially) contains the buffer, and in which bus stops are visible. This can be done using spatial join and spatial selection. The selection predicate has both spatial and nonspatial parts.
- Spatial join and spatial selection can now be used to locate the bus stops that fall within the buffer.

V. Summary and Future Plans

PROBE is an object-oriented database system. Its capabilities can be extended through the addition of object classes. The PROBE data model offers two constructs, entities and functions (stored and computed), and an algebraic query language. Object class instances can be stored in the database. An object class function is modeled by a (computed) function and invoked by referring to the function in a PDM algebra operation.

PROBE supports a spatial data model in which the basic spatial objects are point sets. Point sets have a well-defined set of operations that are useful in many applications. Further specialization of the model, as determined by the application, is accomplished by the addition of spatial object classes.

Efficient evaluation of spatial queries is achieved through a geometry filter. An application-specific spatial object class to be included in PROBE must support only the most basic unary and binary operations. Set-at-a-time processing of spatial objects is the responsibility of the geometry filter. The geometry filter also relies on the extensibility feature to incorporate its own dimension-independent representation of spatial objects.

A "breadboard" implementation of PROBE has been completed. It includes most of the PDM algebra, parts of the geometry filter (enough to include spatial join), object classes to support a geographic application, and application code. The system has a graphical interface, which was also incorporated as an object class.

Future PROBE plans include testing the data model and query processor in a real application and research on extensible query optimization. Much work remains to be done on the geometry filter. Experience with the implementation of spatial join in the breadboard has shown that even the simplest spatial query can be solved in a variety of ways. There is a choice as to what spatial objects should be decomposed (e.g., polygons or polygon edges), what dimension to work in (most k-D problems can also be solved by a simple transformation to $2k$ dimensions), how precise the decomposition process should be, etc. Even less is known about the performances of overlay and the set operations.

Acknowledgment

The authors gratefully acknowledge the contribution of A. Buchmann, U. Chakravarthy, U. Dayal, M. DeWitt, D. Goldhirsch, S. Heiler, and A. Rosenthal to the ideas described here. M. DeWitt also helped to improve the presentation by contributing several extraordinarily thorough reviews of earlier versions of this paper.

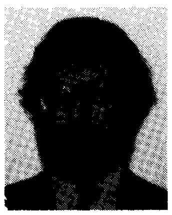
References

[1] D. J. Abel and J. L. Smith, "A data structure and algorithm based on a linear key for a rectangle retrieval problem," *Comput. Vision, Graphics, Image Processing*, vol. 27, no. 1, pp. 19-31, 1983.
[2] D. S. Batory and A. P. Buchmann, "Molecular objects, abstract data types and data models: A framework," in *Proc. 10th Int. Conf. Very Large Databases*, 1984.
[3] D. S. Batory and W. Kim, "Modeling concepts for VLSI CAD objects," *ACM Trans. Database Syst.*, vol. 10, no. 3, 1985.
[4] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509-517, 1975.
[5] W. A. Burkhard, "Interpolation-based index maintenance," in *Proc. 2nd ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, pp. 76-89, 1983.
[6] M. J. Carey *et al.*, "The architecture of the EXODUS extensible DBMS," in *Proc. Int. Workshop Object-Oriented Database Syst.*, pp. 52-65, 1986.
[7] S.-K. Chang *et al.*, "A relational database system for pictures," in *Proc. IEEE Workshop Picture Data Description Management*, 1977.
[8] S.-K. Chang, Ed., *IEEE Comput.*, Special Issue on Pictorial Information Systems, vol. 14, no. 11, 1981.
[9] M. Chock *et al.*, "Database structure and manipulation capabilities of a picture database management system (PICDBMS)," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-6, no. 4, 1984.
[10] U. Dayal *et al.*, "PROBE—A research project in knowledge-oriented database systems: Preliminary analysis," Computer Corporation of America, Tech. Rep. CCA-85-03, 1985.
[11] C. Faloutsos, "Multiattribute hashing using gray codes," in *Proc. ACM SIGMOD*, 1986.
[12] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, no. 12, pp. 905-910, 1982.
[13] D. Goldhirsch and J. A. Orenstein, "Extensibility in PROBE," *Database Eng.*, June 1987.
[14] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD*, 1984.
[15] *Proceedings of the IEEE Workshop on Picture Data Description and Management*, 1977.
[16] R. H. Katz, *Information Management for Engineering Design*. New York: Springer-Verlag, 1985.
[17] R. Laurini, "Graphics databases built on Peano space-filling curves," in *Proc. EUROGRAPHICS Conf.*, pp. 327-338, 1985.
[18] R. A. Lorie and A. Meier, "Using a relational DBMS for geographical databases," IBM Res. Rep. RJ 3848 (43915) 4/6/83, 1983.
[19] F. Manola and J. Orenstein, "Toward a general spatial data model for an object-oriented DBMS," in *Proc. 12th Int. Conf. Very Large Databases*, 1986.
[20] F. Manola and U. Dayal, "PDM: An object-oriented data model," in *Proc. Int. Workshop Object-Oriented Database Syst.*, 1986.
[21] F. Manola, "PDM: An object-oriented data model for PROBE," Computer Corporation of America, Tech. Rep., to appear.
[22] S. Morehouse, "ARC/INFO: A geo-relational model for spatial information," in *Proc. Seventh Int. Symp. Comput. Assisted Cartography*, American Congress on Surveying and Mapping, 1985.
[23] G. Nagy and S. Wagle, "Geographic data processing," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 139-181, 1979.
[24] J. A. Orenstein, "Algorithms and data structures for the implementation of a relational database system," School Comput. Sci., McGill Univ., Montreal, P.Q., Canada, Tech. Rep. SOCS-82-17, 1983.
[25] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *Proc. 3rd ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, pp. 181-190, 1984.
[26] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *Proc. ACM SIGMOD*, 1986.
[27] J. A. Orenstein and F. A. Manola, "Spatial data modeling and query processing in PROBE," Computer Corporation of America, Tech. Rep. CCA-86-05, 1986.
[28] M. Ouksel and P. Scheuermann, "Storage mappings for multidimensional linear dynamic hashing," in *Proc. 2nd ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, pp. 90-105, 1983.

[29] G. Peano, "La curva di Peano nel 'formulario mathematico,'" in *Selected Works of G. Peano*, H. C. Kennedy, Ed. (transl.). Allen and Unwin.
[30] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal recursion: A practical approach to supporting recursive applications," in *Proc. ACM SIGMOD*, 1986.
[31] —, "Traversal recursion: A practical approach to supporting recursive applications," Computer Corporation of America, Tech. Rep. CCA-86-06, 1986.
[32] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed *R*-trees," in *Proc. ACM SIGMOD*, 1985.
[33] H.-J. Schek and P. Pistor, "Data structures for an integrated data base management and information retrieval system," in *Proc. VLDB 8*, pp. 197-207, 1982.
[34] P. Schwarz *et al.*, "Extensibility in the Starburst database system," in *Proc. Int. Workshop Object-Oriented Database Syst.*, pp. 85-93, 1986.
[35] J. D. Smith, "The application of data base management systems to spatial data handling," Dep. Landscape Architecture Regional Planning, Univ. Massachusetts, Amherst, MA, Project Rep., 1984.
[36] M. Stonebraker *et al.*, "Document processing in a relational data base system," *ACM TOOIS*, vol. 1, no. 2, pp. 143-158, 1983.
[37] M. Stonebraker, "Object management in POSTGRES using procedures," in *Proc. Int. Workshop Object-Oriented Database Syst.*, 1986, pp. 66-72.
[38] M. Stonebraker and L. A. Rowe, "The design of POSTGRES," in *Proc. ACM-SIGMOD*, 1986, pp. 340-355.

Jack A. Orenstein received the Ph.D. degree in computer science from McGill University, Montreal, P.Q., Canada, in 1983.

He is a Computer Scientist at the Computer Corporation of America. Before that he was an Assistant Professor in the Department of Computer and Information Science at the University of Massachusetts, Amherst. His research interests include the modeling and processing of spatial data and object-oriented database systems.



Frank A. Manola (S'67-M'71) received the B.S. degree in civil engineering from Duke University, Durham, NC, in 1966 and the M.S.E. degree in computer science from the University of Pennsylvania, Philadelphia, in 1971.

In 1971 he joined the Naval Research Laboratory, Washington, DC. From 1978 to 1987 he was a Senior Computer Scientist at the Computer Corporation of America. He is currently a Principal Member of the Technical Staff in the Department of Intelligent Database Systems, GTE Laboratories, Inc., Waltham, MA. His research interests include object-oriented database technology, multimedia database applications, and data security.

Mr. Manola is a member of the Association for Computing Machinery and the IEEE Computer Society.