MAP ALGEBRA: A DATA MODEL AND IMPLEMENTATION OF SPATIAL PARTITIONS FOR USE IN SPATIAL DATABASES AND GEOGRAPHIC INFORMATION SYSTEMS

By

MARK MCKENNEY

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2008

ACKNOWLEDGMENTS

I would like to acknowledge the members of my committee and thank them for the support they have given me throughout my time at the University of Florida.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS . . . 3

LIST OF TABLES . . . 6

LIST OF FIGURES . . . 7

ABSTRACT . . . 9

CHAPTER

1 INTRODUCTION . . . 11

1.1 Motivation . . . 11
1.2 Problem Statement . . . 14
1.3 Goals and Solutions . . . 16

2 RELATED WORK . . . 21

2.1 Traditional Spatial Types . . . 21
2.1.1 Spatial Data Models . . . 21
2.1.2 Operations . . . 23
2.1.3 Topological Relationships . . . 23
2.2 Spatial Data Types for Maps . . . 26
2.2.1 Spatial Data Models . . . 26
2.2.2 Operations . . . 31
2.2.3 Topological Relationships . . . 33

3 AN INFORMAL OVERVIEW OF SPATIAL PARTITIONS . . . 35

4 THE ABSTRACT MODEL OF SPATIAL PARTITIONS . . . 37

4.1 Spatial Partitions . . . 38
4.2 Operations . . . 40
4.3 Topological Relationships . . . 42

5 QUERYING SPATIAL PARTITIONS: THE USER VIEW OF MAPS IN SPATIAL DATABASES . . . 50

5.1 Types of Map Queries . . . 50
5.2 Map Query Language: A High-level Query Language for Maps . . . 52
5.2.1 Data Model . . . 52
5.2.2 MQL Querying Syntax . . . 55
5.2.3 Querying Maps . . . 57
5.2.3.1 Map queries . . . 57
5.2.3.2 Component attribute queries . . . 58
5.2.3.3 Component queries . . . 59

5.3 Querying Maps in Databases Using SQL . . . 60
5.3.1 Data Model . . . 60
5.3.2 Creating Maps . . . 61
5.3.3 Querying Maps . . . 62
5.3.3.1 Map queries . . . 63
5.3.3.2 Component attribute queries . . . 64
5.3.3.3 Component queries . . . 65

6 THE DISCRETE MODEL OF SPATIAL PARTITIONS . . . 67

6.1 Definitions from Graph Theory . . . 67
6.2 Representing Spatial Partitions as Graphs . . . 68
6.3 Properties of Spatial Partition Graphs . . . 76

7 THE IMPLEMENTATION MODEL OF SPATIAL PARTITIONS . . . 82

7.1 Map2D: an Implementation Model of Map Algebra . . . 82
7.1.1 A Data Model For Representing Spatial Partitions . . . 82
7.1.2 Algorithms for Map Operations . . . 86
7.1.2.1 Intersect . . . 87
7.1.2.2 Relabel . . . 92
7.1.2.3 Refine . . . 96
7.1.2.4 Combining operations to form new operations . . . 106
7.2 A Prototype Implementation of Map Algebra . . . 107
7.3 Performance Comparison of Map2D Algorithms with an Existing GIS . . . 108
7.3.1 Method of Comparison . . . 108
7.3.2 Results . . . 111

8 CONCLUSION . . . 116

REFERENCES . . . 119

BIOGRAPHICAL SKETCH . . . 126

LIST OF TABLES

Table page

2-1 A summary of spatial systems and their treatment of maps. 26

4-1 The first 42 valid matrices and protoypical drawings representing the possible topological relationships between maps. Each drawing shows the interaction of two maps, one map is medium-grey and has a dashed boundary, the other is light-grey and has a dotted boundary. Overlapping map interiors are dark-grey, and overlapping boundaries are drawn as a solid line. For reference, the figure for matrix 41 shows two disjoint maps and the figure for matrix 1 shows two equal maps. 48

4-2 The final 7 valid matrices and protoypical drawings representing the possible topological relationships between maps. 49

LIST OF FIGURES

Figure page

1-1 A depiction of current spatial system architecture. The data layer manages storage and retrieval of data from a file system or database. The middleware layer performs processing. The user interface layer provides user interaction mechanisms. . . . 13

1-2 Our proposed system architecture. . . . 18

2-1 Examples of complex spatial objects. A) A complex point object. B) A complex line object. C) A complex region object. . . . 21

2-2 Sample composite region with two components presenting a hole-like structure. . 22

2-3 A depiction of various spatial operations applied to two regions. A) The first argument region. B) The second argument region. C) The intersection of the regions. D) The union of the regions. E) The difference of the regions. . . . 23

2-4 The 9-intersection matrix for spatial objects A and B . . . 24

3-1 A sample spatial partition with two regions. A) The spatial partition annotated with region labels. B) The spatial partition with its region and boundary labels. Note that labels are modeled as sets of attributes in spatial partitions. . . . 36

4-1 A spatial partition π with two disconnected faces, one containing a hole. A) The interior (π°). B) The boundary ($\partial\pi$). D) The exterior (π^{-}). Note that the labels have been omitted in order to emphasize the components of the spatial partition. . . . 39

4-2 The application of the refine operation to a spatial partition. A) A spatial partition with two regions and its boundary and region labels. B) The result of the refine operation on Figure **A**. . . . 40

5-1 Two sample maps. A) A map with labels consisting of a single string named *crop*. B)A map with labels consisting of a pair of integers, indicating the average temperature and rainfall for each region, named *Avg_Temp* and *Avg_Rain*, respectively. . . . 54

5-2 A relation containing a map2D column and the associated label tables. The table Map1AttributeTable is associated with the map with ID equal to 1 in the table MapTable, and the table Map2AttributeTable is associated with the map with ID equal to 2 in the table MapTable. . . . 61

6-1 A spatial partition and its corresponding SSPG. A) The refinement of the partition in Figure 3-1A. B) The SSPG corresponding to Figure **A**. Nodeless edges are dashed. . . . 70

6-2 A labeled plane nodeless pseudograph for the partition in Figure 3-1. The edges and vertices are marked so that the sets of vertices, edges, nodeless edges, and faces can be expressed more easily. 75

7-1 A map showing an industrial and commercial zone. 83

7-2 A complex region object with each segment labeled. 85

7-3 Two different views of a map. A) The map represented in the implementation model of Map Algebra. B) The map with its segments labeled. 86

7-4 The result of the *intersect* operation applied to two maps. 87

7-5 Aggregating map labels in an intersect operation. A) The first argument map. B) The second argument map. C) The the result of aggregating the argument maps' labels during an intersect operation. 88

7-6 The result of the plane sweep portion of the intersect algorithm for the maps shown in Figure 7-5. A) The labeled geometry. B) The mapping. 90

7-7 The resulting label table from an intersect operation. A) The label table for the map shown in Figure 7-5A. B) The label table for the map shown in Figure 7-5B. C) The label table computed as the result of an intersect operation. . . . 91

7-8 Processing halfsegments in a region. A) Processing the smallest halfsegment h of the sequence. B) Processing halfsegment k of a cycle. 101

7-9 The result of the $overlay_2$ operation applied to two maps. 106

7-10 Images of the PMAI displaying the final result of the *overlay* algorithm test for each of the data sets used. 110

7-11 Running times for the intersect operation in PMAI and GISX. 113

7-12 Running times for the $overlay_2$ operation in PMAI and GISX. 114

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

MAP ALGEBRA: A DATA MODEL AND IMPLEMENTATION
OF SPATIAL PARTITIONS FOR USE IN SPATIAL DATABASES
AND GEOGRAPHIC INFORMATION SYSTEMS

By

Mark McKenney

August 2008

Chair: Markus Schneider
Major: Computer Engineering

The idea of a *map* is a fundamental metaphor in spatial systems. For instance, in the fields of geographical information systems (GIS), spatial databases, geography, robotics, computer assisted design (CAD), and spatial cognition, the arrangement of spatial data into map form plays a primary role in the composition, representation, and analysis of spatial data. However, no models of maps currently exist that provide a precise definition of a spatial data type for maps and operations over them. Instead, many informal definitions are provided, many of which are tied to specific implementation concepts. Furthermore, although the integration of spatial databases into spatial fields such as GIS has received much attention, the notion of integrating maps into databases has been overlooked. Thus, the idea of integrating maps into SQL and performing queries over them has not yet been explored. This thesis describes the design and implementation of *Map Algebra*, a type system and operations for maps in spatial databases. We provide a three level approach to defining Map Algebra. First, we provide an *abstract model* of maps. This is a mathematical description of maps and their operations and topological predicates. This model is defined on formal mathematical concepts that are not implementable in computer systems; therefore, we then provide a *discrete model* of maps based on discrete concepts that can be translated to computer systems. We then provide an *implementation model* of maps, complete with algorithms to implement map operations, that can be

directly implemented in a computer system. Furthermore, we develop a query language called *Map Query Language* that can be used to pose queries over maps in databases.

CHAPTER 1
INTRODUCTION

The idea of a *map* is a fundamental metaphor in spatial systems. For instance, in the fields of geographical information systems (GIS) [1–8], spatial databases [9–15], robotics [16–19], computer assisted design (CAD) [20–23], and spatial cognition [18, 24, 25], the arrangement of spatial data into map form plays a primary role in the composition, representation, and analysis of spatial data. Specifically, in applications such as spatial databases systems (SDS), GISs, and global positioning systems (GPS), the map itself plays a central role in that the map is the primary user interface tool. GISs in particular have adopted maps for geoprocessing, and a rich set of geoprocessing tools has been developed around them. Thus, maps in the context of GISs and SDSs and the functionality provided in these systems are of vital importance to industry, government, and the many scientific fields which make use of these systems.

The general goal of this dissertation is to examine the way maps are implemented and operated on in current spatial systems, and discover methods to improve the representation and storage of map data and the ways that users can interact with maps. We approach this goal by attempting to integrate maps as *first class citizens* in databases. In the next few sections we motivate our goal by providing a more detailed discussion of the state of the art with regard to maps in spatial systems, identifying problems related to the current state of the art in maps, and identifying some specific goals to improve the state of the art that the remainder of this thesis will address.

1.1 Motivation

In general, spatial systems can be conceptualized as consisting of three components arranged as shown in Figure 1-1: a data layer, a processing or middleware component, and a user interface. The data layer is typically implemented as a spatial database or file system that manages the physical storage of the spatial data. The middleware layer performs operations and computations that the data layer does not provide. Typical

examples of such computations are coordinate projections, triangulation of polygons, and the preparation of data for input to the user interface. The visualization layer displays data to the user and processes user input, which is passed back to the middleware layer and the data layer. In the following, we discuss each layer in more detail. Specific systems and their capabilities are discussed in Section 2.

We begin by examining the data layer in spatial systems. As was mentioned before, the data layer is used to store the actual spatial data. Spatial data itself is stored in the form of *spatial primitives*, which are the fundamental units of spatial objects. In early systems, spatial primitives consisted of *points* and *straight line segments*, which were used to construct spatial objects such as points, lines, and regions. The first generations of spatial data storage stored these primitives in files [26, 27]. Because file systems typically provide only input/output operations, all spatial operations were implemented in middleware, and the file system was used only for storage. When databases began to be used to store spatial data instead of file systems, the database was treated much like a file system in that it was only used for storage and spatial operations were still implemented in middleware [2, 14, 28–30]. For example, a region would be assigned a unique identifier, and all straight line segments that made up the region would exist in tuples in a database table along with the region identifier indicating which region they belonged to. Thus, a join operation was typically required between a region table and table containing straight line segments in order to construct a region. This proved to have poor performance, and, because the spatial objects were not known by the database, all operations were implemented in middleware.

The advent of extensible database systems [31–39] provided a solution to the problems with the original attempt to integrate spatial data into databases. With an extensible database, the database could be made aware of a *spatial data type* consisting of both data and operations over the data [10, 14, 15, 40]. Therefore, a database could manipulate a spatial data type, such as a region, as if it were any other type of which the

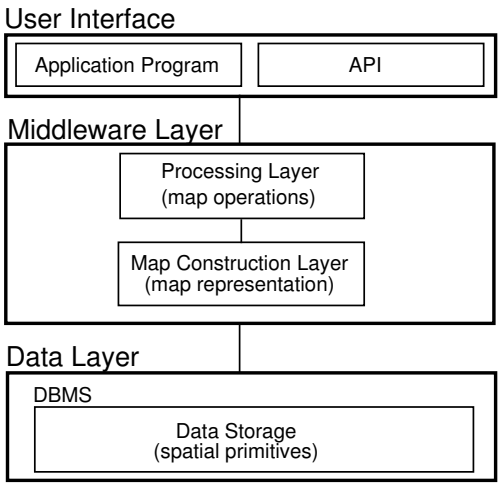


Figure 1-1. A depiction of current spatial system architecture. The data layer manages storage and retrieval of data from a file system or database. The middleware layer performs processing. The user interface layer provides user interaction mechanisms.

database was aware. Thus, operations like spatial intersection, union, and difference could be moved from middleware directly into the database. In other words, spatial objects became *first class citizens* in the database, allowing SQL queries over spatial objects that took advantage of spatial operations. The result was increased performance and flexibility of spatial systems. The current state of the art in spatial systems is to use a data layer consisting of a database that is aware of points, lines, and regions, together with operations such as spatial intersection, union, difference, and topological predicates.

The role of the middleware layer has typically been to provide operations that are not available in the data layer. Because current spatial databases are aware of concepts such as points, lines, and regions, the middleware layer in current systems combines these more simple spatial objects to form more complex spatial arrangements, typically *GIS maps*. Here we use the term GIS maps to mean a map as it is defined in the GIS literature [3–6]: a map is a collection of map layers such that each map layer is composed of points, lines, and regions that are optionally associated with attribute data. Therefore, a middleware layer collects points, lines, and regions from the database, and arranges them into map layers, which compose a GIS map [1]. Once a map is created, it is passed up to the

visualization layer, which handles issues such as clipping, displaying thematic information, and providing color and texture to maps for display.

In summary, Figure 1-1 illustrates the architecture of current spatial systems. The data layer of current systems utilizes a database which can store and manipulate the spatial primitives of points, lines, and regions. The middleware layer contains a component that can collect these spatial primitives and construct a map from them. Map operations, such as overlaying layers, are performed on constructed maps, and the results are presented to the user through a visualization or user interface layer.

It is clear that the concept of a map is important in spatial systems due to the fact that they are almost universally used in such systems. However, given the state of the art as discussed above, we note that the current middleware implementation of maps is strikingly similar to the middleware implementations of points, lines, and regions that existed in earlier systems. Specifically, we note that the spatial components of maps (i.e., the points, lines, and regions that compose the map) can be spread across multiple tables in a database. Therefore, we postulate that a middleware implementation of maps suffers similar problems to the middleware implementation of points, lines, and regions. However, the open problem exists that there is currently no method of integrating a map type into a database data layer of spatial systems. In other words, maps are not *first class citizens* in databases. In order to integrate maps as first class citizens in databases we must address the additional open problem that querying mechanisms for maps in databases have not been thoroughly studied. Currently, map attributes, such as names, populations, etc., can be queried and spatial queries can be performed over the points, lines, and regions that compose a map, but queries over maps themselves are unavailable. For example, it is not possible to find all maps in a database that form a submap of a query map.

1.2 Problem Statement

Based on the current state of the art of spatial systems and the previous discussion, the overall problem this thesis addresses is to discover methods for integrating maps

as first class citizens into spatial systems. We split this problem into five sub problems that can be addressed individually: (i) we must have a definition of maps that describes their properties, (ii) we must know the semantics of map operations, (iii) we must be able to determine relationships between maps based on their geometries (i.e., topological relationships), (iv) we must have a way to express queries over maps that provides access to all features of maps as well as map operations and predicates, (v) we must have efficient methods of implementing maps and operations in spatial systems.

A formal definition of maps that precisely describes their properties is essential for defining operations and predicates and ensuring closure properties of maps over them. In Chapter 2, we will see that many definitions for maps exist, but they are all based on informal descriptions or treat maps as collections of more primitive spatial types, and not as first class citizens. Therefore, in order to address this problem, we must provide a precise definition of maps based upon mathematical concepts instead of intuitive descriptions.

The existence of a formal model of maps based on mathematical concepts will allow the definition of precise semantics for map operations. Therefore, we must provide such semantic descriptions of map operations that currently exist based upon our model. This will allow us to ensure operational closure, guaranteeing that map properties are preserved through the application of such operations.

A formal model of maps based on mathematical concepts can also be leveraged to define the relationships of two maps in space. An important class of spatial queries over traditional spatial types is the class of topological queries. Topological queries are based upon topological predicates (Section 2) which indicate qualitative properties between a pair of spatial objects such as adjacency or inclusion. These predicates form the basis of spatial index structures as well as topological queries. Because of the importance of this class of queries, we need to define topological relationships between maps based on a formal definition of maps.

Because our overall goal is to include maps as first class citizens in databases, we must provide some method for users to interact with maps in databases. Therefore, we must provide some mechanism to interact with maps based on SQL, which is the standard database querying language. Because maps are currently not treated as first class citizens, no method of integrating map types into SQL currently exists. Therefore, we must show how maps can be represented in a database context, and develop a method for integrating map operations and map querying mechanisms into SQL.

Finally, the map data type and operations over it must be implemented. Therefore, we must provide an implementation for maps consisting of a data model and operations that preserves the properties of maps and the semantics of map operations. Furthermore, the operations over maps must be implemented efficiently, and the data model must support the operations adequately. Thus, we must provide algorithms for operations and implement them in order to verify their efficiency.

1.3 Goals and Solutions

In order to address the problems discussed in the previous section, we require a formal *type system* or *algebra* for maps which precisely defines the semantics of maps and operations over maps. Although maps and their operations are described in many works [3–7, 40], these definitions are all informal. In such cases, situations may arise in which the semantics of an operation are unclear; therefore, we will define a new *Map Algebra* that provides precise semantics. We approach the development of Map Algebra from three levels: the abstract level, the discrete level, and the implementation level. At the abstract level, we define a purely mathematical model of maps. This allows us to precisely describe the properties of maps. Using this model, we can then identify map operations, and precisely define their semantics. Map operations differ from traditional spatial operations because maps are thematic; therefore, map operations must take care of thematic data as well as geometric data. Furthermore, we can address the concerns of closure properties of map operations at this level. At this level, we can also leverage the formal definition of

maps to discover topological relationships between maps. Therefore, by completing the abstract definition of Map Algebra, we will provide solutions to problems (i), (ii), and (iii) mentioned above.

Given the abstract model of Map Algebra, we can consider constructs necessary for querying maps. We approach the problem of querying maps from two levels. First, we develop a new map query language, called MQL, that is able to express queries over maps. This is a special purpose language designed around maps, and is intended as a high level, user view of maps in databases. We then show how MQL constructs can be expressed in SQL. By creating a method to integrate maps into SQL, we can consider maps as first class citizens in a database setting as well as take advantage of the wide acceptance of SQL by database users.

Once the abstract model of Map Algebra, including querying mechanisms, is complete, we consider the implementation of maps in computer systems. This is achieved by first considering Map Algebra at the discrete level. A discrete model of maps translates the abstract model of maps into the discrete domain by defining discrete representations, such as graph representations, for the map data model defined at the abstract level. The discrete model should preserve the properties of the abstract model, but eliminate the need for concepts such as infinite point sets and continuous mappings. The result is a model that can represent maps based solely on discrete concepts. By defining a discrete representation of maps, we can ensure that the properties of maps defined at the abstract level are transferred into the discrete domain. This will allow us to ensure that maps represented in discrete methods are valid in the sense that they preserve the properties of maps as defined in the abstract, mathematical model.

At the implementation level, we focus on developing data structures and algorithms that can be used to implement a map data type. The properties of the implementation model of maps should be identical to the properties of the abstract and discrete models of maps, but be oriented towards implementation. The implementation model should provide

mechanisms for users to create, store, and manipulate maps. Once the implementation model is complete, we can directly implement it in a database system. The creation of an implementation model address problem (v) mentioned above.

Recall that the general goal of this proposal is to discover methods to improve the representation and storage of map data and the ways that users can interact with maps. Based on the state of the art of spatial systems and the problems identified above, we refine the goal of this proposal to discover a method by which maps can be integrated as first class citizens in database systems, including a method by which maps can be queried. Figure 1-2 provides a pictorial description of this goal. In essence, we aim to move the map functionality in spatial systems from the middleware layer to the data layer.

Once an implementation of Map Algebra is achieved, we hypothesize that it will be superior to current map implementations in several respects. We summarize the goals of our Map Algebra implementation in terms of the following three hypotheses.

Hypothesis 1. *By integrating maps as first class citizens in databases, constructing maps and performing operations over maps will be more efficient than current methods of implementing maps*

Because maps are currently implemented in middleware, GIS maps must be constructed from their component points, lines, and regions that are stored in the data layer. Therefore, a processing step must take place to perform this construction

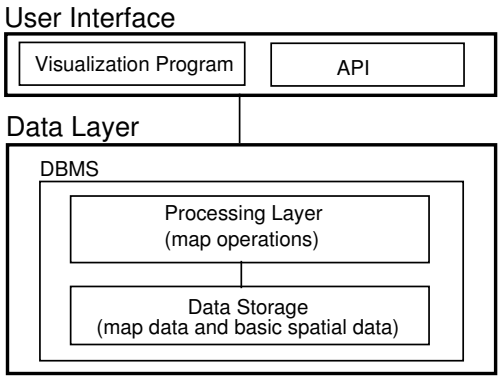


Figure 1-2. Our proposed system architecture.

in addition to retrieving the map components that are possibly scattered across multiple tables in a database. If a map is stored as a single database object, then it can simply be read directly from the database, and this construction step is no longer needed. Furthermore, the output of spatial operations, such as map overlay, is often *unstructured*, meaning that the algorithm computes the line segments that form the resulting geometry, but does not identify the spatial primitives in the resulting geometry (i.e., the points, lines, and regions). A separate algorithm must be used to identify the spatial primitives of the resulting map [41]. We hypothesize that the integration of maps as first class citizens in spatial databases will eliminate much of the overhead processing associated with middleware implementations of maps.

Hypothesis 2. *By integrating maps as first class citizens in databases, spatial systems that utilize maps will be less complex and easier to implement*

Modern databases provide a significant amount of functionality regarding data access and storage. Specifically, databases typically provide transactions, concurrency control, data recovery, an SQL interface, and multi-user access. If a map is implemented as a type in a database system, the database automatically provides these services. If a map is implemented in middleware, these concepts must be implemented explicitly, which typically results in duplication of database services in the middleware layer. Therefore, we hypothesize that a database implementation of a map type will allow application code to be less complex since applications do not have to implement these database services. We will investigate this hypothesis through implementing a prototype system based on our map concept. If we can successfully implement our map concept without the use of a middleware layer, then all map functionality will be provided by the database, which will also provide the traditional database services.

Hypothesis 3. *By integrating maps as first class citizens in databases, map functionality can be provided that is currently unavailable in spatial systems.*

Current spatial systems provide an immense amount of geoprocessing tools and operations for maps. However, we believe that integrating a map type into a database will provide additional functionality that does not exist in current systems. For example, a map type in a database can be directly used in SQL queries. Therefore, map queries can be issued from both within and externally to a GIS environment. External queries are possible because GIS middleware libraries are not required for the execution of map queries. Furthermore, complex queries involving maps, such as computing nested subqueries and aggregates, can be directly expressed in SQL and do not require a middleware layer or even a GIS intermediary. This provides more flexibility when accessing maps since any technology with database connectivity using SQL can issue queries over maps.

CHAPTER 2
RELATED WORK

We present the work related to Map Algebra in two general categories: work related to traditional spatial data types, and work related to data types for maps. In each category, we discuss the various data models that have been proposed, as well as operations and topological predicates that have been developed for the models.

2.1 Traditional Spatial Types

Traditional spatial data types provide models for representing individual *points*, *lines*, and *regions*. This is the approach that has typically been taken in spatial data modeling.

2.1.1 Spatial Data Models

We distinguish two *generations* of spatial data types. The types of the first generation have a simple structure, and are known as the *simple spatial types* [42–45]. A *simple point* describes an element of the Euclidean plane $\mathbb{R}^2$. A *simple line* is a one-dimensional, continuous geometric structure embedded in $\mathbb{R}^2$ with two end points. A *simple region* is a two-dimensional point set in $\mathbb{R}^2$ and topologically equivalent to a closed disk.

Additional requirements of applications as well as needed closure properties of operations led to the second generation of *complex spatial data types* [45–47] illustrated in Figure 2-1 ([10] for a survey). A *complex point* is a finite point collection (e.g., the positions of all lighthouses in Florida). A *complex line* is an arbitrary, finite collection of one-dimensional curves, i.e., a spatially embedded network possibly consisting of several disjoint connected components (e.g., the Nile Delta). A *complex region* permits multiple

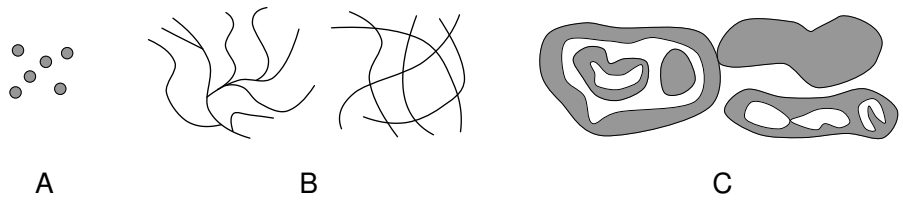


Figure 2-1. Examples of complex spatial objects. A) A complex point object. B) A complex line object. C) A complex region object.

areal components, called *faces*, and holes in faces (e.g., Italy with its mainland and offshore islands as components and with the Vatican as a hole).

Two additional spatial data types for regions have been proposed as intermediate steps between simple and complex regions. These are, *composite regions* [48] and *simple regions with holes* [45, 47, 49]. A composite region can contain multiple faces, but no holes. In other words, a composite region consists of finitely many simple regions that are either disjoint, or meet at single points. Although holes are not allowed in composite regions, "hole-like" configurations can exist if two components of one region touch at a single point of their boundaries at two different locations (Figure 2-2).

Figure 2-2. Sample composite region with two components presenting a hole-like structure.

A simple region with holes contains only a single face, with finitely many holes. The holes in a simple region with holes are allowed to meet at a point, but cannot form a configuration that causes the interior of the region to be disconnected. In other words, a hole-like structure cannot be formed by the holes in a simple region with holes. Intuitively, a simple region with holes is a complex region that has only a single face.

Each spatial type is defined such that it is made up of three parts: the *interior*, *boundary*, and *exterior*. Given a spatial object A, these components are indicated respectively as A°, ∂A, and A^-. For example, the boundary of a line is its endpoints, and its interior consists of the lines that connect the endpoints. The exterior of a line consists of all points in $\mathbb{R}^2$ that are not part of the interior or boundary. Similarly, the boundary of a region is the line that defines the region's border. The interior consists of all points that lie inside the region, and the exterior consists of all points that are not part of the boundary or interior. These concepts are required for the definition of topological relationships between spatial types (defined in the next section).

Figure 2-3. A depiction of various spatial operations applied to two regions. A) The first argument region. B) The second argument region. C) The intersection of the regions. D) The union of the regions. E) The difference of the regions.

2.1.2 Operations

Research into spatial operations over the traditional spatial types has traditionally centered around the point set operations of *intersection*, *union*, and *refine* [14, 15, 50–52]. If two spatial objects are considered as point sets, then the intersection, union, and difference operations correspond to set operations over those sets. For example, Figure 2-3 shows the result of these operations for a pair of regions.

Research into implementing these operations has built on concepts from computational geometry. Specifically, plane sweep algorithms from computational geometry [53, 54] have been adapted to compute geometric intersection, union, and difference over spatial objects given that the objects are represented as a sequence of straight line segments [55–58]. This has resulted in algorithms capable of computing such intersections in $O(n \lg n + k)$ time for geometric objects with n straight line segments and k intersecting segments.

2.1.3 Topological Relationships

The study of topological relationships between objects in space has been the subject of a vast amount of research [44, 45, 49, 59]. In the areas of databases and GIS, the motivation for formally defining topological relationships between spatial objects has been driven by a need for querying mechanisms that can filter these objects in spatial selections or spatial joins based on their topological relationships to each other, as well as a need for appropriate mechanisms to aid in spatial data analysis and spatial constraint specification.

$$\begin{pmatrix} A^\circ \cap B^\circ \neq \varnothing & A^\circ \cap \partial B \neq \varnothing & A^\circ \cap B^- \neq \varnothing \\ \partial A \cap B^\circ \neq \varnothing & \partial A \cap \partial B \neq \varnothing & \partial A \cap B^- \neq \varnothing \\ A^- \cap B^\circ \neq \varnothing & A^- \cap \partial B \neq \varnothing & A^- \cap B^- \neq \varnothing \end{pmatrix}$$

Figure 2-4. The 9-intersection matrix for spatial objects A and B

Topological relationships indicate qualitative properties of the relative positions of spatial objects that are preserved under continuous transformations such as translation, rotation, and scaling. Quantitative measures such as distance or size measurements are deliberately excluded in favor of modeling notions such as connectivity, adjacency, disjointedness, inclusion, and exclusion. Attempts to model and rigorously define the relationships between certain types of spatial objects have lead to the development of three popular approaches: the *9-intersection model* [44], which is developed based on point set theory and point set topology, the *calculus based method* [45], which is also based on point set topology, and the *RCC model* [59], which utilizes spatial logic. Because the definitions of spatial objects are based on topological principles, and the inability of the calculus based method to identify a complete set of topological relationships, the 9-intersection model is typically used to model topological relationships between spatial objects in the field of SDSs. The 9-intersection model characterizes the topological relationship between two spatial objects by evaluating the non-emptiness of the intersection between all combinations of the interior ($^\circ$), boundary (∂) and exterior ($^-$) of the objects involved. A unique 3×3 matrix, termed the *9-intersection matrix* (9IM), with values filled as illustrated in Figure 2-4 describes the topological relationships between a pair of objects:

Various models of topological predicates based on the 9-intersection model using both *component derivations*, in which relationships are derived based on the interactions of all components of spatial objects, and *composite derivations*, in which relationships model the global interaction of two objects, exist in the literature. Examples of component

derivations can be found in [48, 49]. In [49], the authors define topological relationships between regions with holes in which each of the relationships between all faces and holes are calculated. Given two regions, R and S, containing m and n holes respectively, a total of $(n + m + 2)^2$ topological predicates are possible. It is shown that this number can be reduced $mn + m + n + 1$; however, the total number of predicates between two objects depends on the number of holes the objects contain. Similarly, in [48], predicates between complex regions without holes are defined based on the topological relationship of each face within one region with all other faces of the same region, all faces of the other region, and the entire complex regions themselves. Given regions S and R with m and n faces respectively, a matrix is constructed with $(m+n+2)^2$ entries that represent the topological relationships between S and R and each of their faces.

The most basic example of a composite derivation model (in which the global interaction of two spatial objects is modeled) is the derivation of topological predicates between simple spatial objects in [44]. This model has been used as the basis for modeling topological relationships between object components in the component models discussed above. In [46], the authors apply an extended 9-intersection model to point sets belonging to complex points, lines, and regions. Based on this application, the authors are able to construct a composite derivation model for complex data types and derive a complete and finite set of topological predicates between them, thus resolving the main drawback of the component derivation model.

More recently, it has been observed that composite models of topological relationships between spatial objects are *global*, in that they characterize an entire scene by a single topological relationship that may hide *local* information about the object's relationship [60]. The hiding of local information is expressed in two ways in global topological relationship models: through the *dominance problem*, and the *composition problem*. The dominance problem indicates the property that the global view exhibits *dominance* properties among the topological relationships as defined by the 9-intersection model. For

Table 2-1. A summary of spatial systems and their treatment of maps.

Map Model	Middleware Maps	Thematic Map Model	Geometric Map Model	Operational Closure	Data Level Topological Constraint Enforcement
Raster		✓		✓	✓
Collections			✓		
Graph Models			✓		
Tigris	✓		✓		✓
ESRI GIS	✓		✓		
Map Algebra		✓		✓	✓

example, while building roads between two adjacent countries, one might be interested to know that there is a disjoint island in one of the countries for which a bridge to the other country is required. The disjointedness in this case is overshadowed or dominated by the existing *meet* (adjacent) situation between the countries' mainlands. The composition problem expresses the property that a global topological predicate may indicate a certain relationship between two objects that does not exist locally. For example, consider two complex regions that have individual faces that satisfy the *inside*,*covers*, and *meet* predicates. Globally, this configuration satisfies the overlap predicate even though no faces *overlap* locally. These properties have been addressed through the *local topological relationship* models between composite regions [60] and between complex regions [61]. These approaches model a topological relationship between two multicomponent objects based on the topological relationships that exist between the components of the objects. Furthermore, it is shown that these models are more expressive than the global 9IM models in that they can distinguish all the topological scenes that the 9IM models can distinguish, plus many more.

2.2 Spatial Data Types for Maps

2.2.1 Spatial Data Models

In this section, we examine the various approaches that have been taken to model maps in both the literature and in commercial GIS products. We present each approach to modeling maps, and then evaluate the approach with respect to four criteria. First,

we determine if the map model that is described or implemented is a middleware map model, or if it is actually a data level map model in spatial systems. We say that a map model is middleware if the map requires construction or processing outside of the database or file system in which it is stored. Second, we determine if the system provides thematic maps, meaning a map geometry annotated with thematic information, or non-thematic maps, consisting of only a map geometry. In the non-thematic case, it may be possible for a system to store thematic information separately from the map geometry, but it is not included in the map model itself. Third, we check if operational closure has been formally addressed. This is important since without formally addressing operational closure, we cannot be certain that the result of map operations will be valid in all circumstances. Finally, we find whether each model provides a mechanism to enforce topological constraints between map contents within the model, or if an external mechanism is required. External mechanisms of constraint enforcement are undesirable since they can be difficult to express outside of the relational data model and require computational overhead to enforce. For research models that do not provide an implementation, we evaluate the model based on its properties and description. We summarize the findings of this evaluation in Table 2-1.

The first approach to modelling maps that we discuss is the *raster*, or more generally, *tessellation* approach. Tesselation approaches [7, 62] impose a tessellation scheme on the underlying space and assign a value to each cell. A region in this approach consists of all adjacent cells with an identical label. Therefore, a map is considered to be a bounded tessellation such that each cell carries a label; i.e., maps in this scheme are thematic. However, tessellation maps generally allow only a single label for each cell, or sometimes a few labels. A disadvantage to the tessellation approaches is that they do not scale well to handling large numbers of labels in each cell due to the high storage requirement of these approaches. The advantage of the tessellation approach is that it aligns nicely with data collection mechanisms, such as sampling an area divided into a grid, or taking data

from a sensor network. Thus, tessellation approaches provide a thematic map type, and an algebra exists over tessellation maps with operational closure [7]. Furthermore, they implicitly enforce topological constraints over its contents since values are constrained to grid cells. However, tessellation approaches are limited in generality since they are geometrically constrained to the tessellation scheme in use, and tessellation representations of data can be very large, especially if high resolution cells are required.

The *collection* approach to modeling maps takes the view that a map is simply a collection of more basic spatial entities that may satisfy certain topological constraints. This is the approach taken in [12–15, 40, 51]. In general, collection types do not implicitly support thematic maps, but model maps as purely geometric structures. Some models provide mechanisms to associate thematic data with the individual components of the map, but the thematic data is not part of the map definition. Furthermore, no type closure is provided. In fact, the semantics of operations over collections, even in the specification provided by the Open Geospatial Consortium, are defined informally and are ambiguous. Additionally, no method of enforcing topological constraints over map contents is provided. However, it is possible to treat an entire geometric collection as a single object in the data level of spatial systems, meaning that they avoid a middleware implementation.

In [11, 63], the authors consider modeling maps as special types of plane graphs. This is an interesting approach to modelling maps because two-dimensional maps typically impose a plane graph on the embedding space. Problems in the proposed methods arise when different spatial objects in the map share coordinates. For example, given the method of deriving a plane graph from a collection of points, lines, and regions, it is unclear if a spatial point object that has the same coordinates as the endpoint of a spatial line object in the plane graph can be distinguished. Furthermore, the authors require a separate structure to model what they term the *combinatorial structure* of a plane graph, which includes the topological relationships between different spatial components of the

graph. Therefore, this model allows a geometric map type modeled as a single graph, but does not incorporate thematic information. Furthermore, operational closure over such maps is not discussed, nor is the specification and enforcement of topological constraints.

The Topologically Integrated Geographic Information System (TIGRIS) [2, 26, 27] is unique in that is an early system that incorporated maps into its spatial data storage model. In the TIGRIS system, the individual traditional spatial objects are stored in a map which contains no overlapping regions. For example, if a region representing Florida and a region representing a hurricane that partially overlaps Florida are stored, then three regions are actually stored by TIGRIS, the region representing the intersection of Florida and the hurricane, the region consisting of the part of Florida that is disjoint with the hurricane, and vice versa. These three regions are referred to as *topological primitives*, from which the original regions can be reconstructed. Although maps were integrated at a deep level in this system, a map type was not available to the user except through a middleware map implementation. Furthermore, the map storage was utilized to provide a spatial algebra for points, lines, and regions, but not for maps. Therefore, no operational closure was provided over maps. However, this system used the map storage to enforce topological constraints over regions at the data level.

We focus our discusion of commercial GIS map offerings on the software provided by Environmental Systems Research Institute (ESRI), since it is an industry standard and has had portions of its architecture published in the literature. In general, the ESRI software provides maps to users as a user interface that visualizes individual point, line, and region objects. Therefore, maps are implemented in a middleware layer that manipulates the more basic spatial types. The closest technology to a data level map data type that ESRI provides is the notion of *topologies* in [1]. A topology, in this sense, is a collection of spatial objects that satisfy certain topological constraints; specifically, spatial data objects are only allowed to *meet* or be *disjoint*. The drawback to topologies is that they do not have a formal model to handle thematic data, and thus maps are

implemented as geometric constructs. Furthermore, the description of the implementation of topologies reveals that they are implemented as a middleware type, and thus, the toplogical constraints over the contents of a topology must be expressed and enforced with an algorithm in a middleware layer.

From an implementation standpoint, research into maps has focused on data structures that are able to represent topological information about map components. Specifically, these structures represent adjacency information about the regions in maps. Furthermore, these models typically represent the boundary of a map as a collection of straight line segments. The earliest of these structures to be studied was the *winged edge structure* [23]. This structure is straightforward and can be implemented in memory or on disk. The winged edge structure associates the edges that define the boundary of a map with the regions they separate using a unique region identifier. Furthermore, an edge identifier is associated with each edge. In addition to carrying the region identifiers of the adjacent regions, each edge also carries an edge identifier indicating which edge would be encountered next if traversing the current region in a clockwise or counter-clockwise direction. Therefore, traversing all edges that bound a region is performed rather easily using these edge identifiers. The *doubly connected edge list* [57] is a similar structure that maintains less information at each edge, and is therefore more compact.

Another structure used for implementing maps is the *quad edge structure* [64]. This structure is unique in that it represents the boundary of a map, which turns out to be a plane graph when using most map models, and the dual of the plane graph imposed by the boundary of the map. Furthermore, an algebra is defined that allows a pointer to be moved around the map on either the boundary graph, or its dual. Therefore, this structure allows connectivity queries, in which a chain of regions can be identified that connect two argument regions, to be computed using existing graph algorithms. The drawback to this structure, as opposed to the winged edge structure and the doubly connected edge list, is

that it is more complicated to construct and maintain, and it is not as compact as more information is explicitly represented.

We base our Map Algebra on the model of maps presented in [65]. The authors of this paper define an abstract, mathematical data model that formally describes the type of *spatial partitions*. A spatial partition is the partitioning of the plane into regular, open point sets such that each point set is associated with a label. The use of labels to identify point sets allows thematic information to be modeled implicitly in spatial partitions. Furthermore, operations are defined over spatial partitions, and it is shown that the operations are closed over the type of spatial partitions. A detailed description of the type of spatial partitions is provided in Chapter 4. The main drawback to this model is that it is based on the concepts of infinite point sets and mappings that are not able to be represented discretely. This has been addressed in [66], in which a graph model of spatial partitions is defined. This model has the same properties of the spatial partition model, but is represented using discrete concepts. A detailed description of spatial partition graphs is provided in Section 6

2.2.2 Operations

In addition to modeling the geometric and thematic aspects of maps, operations over maps have been extensively investigated [3–10, 14, 15, 40, 51, 67–69]. Here we briefly describe some of the operations. Note that all operations known over maps can be expressed in terms of three *power operations*[65]. We present these power operations in detail in Section 4.2, and intuitively describe some of the more well known map operations here. The most well known map operation is the *map overlay*. An overlay takes two maps as arguments, and computes a resulting map that contains the spatial and thematic data of both argument maps. For instance, consider a map of the United States that only contains a single region representing the entire country, and a map containing a single region representing a high temperature zone that partially overlaps the map of the US. The overlay of these maps would contain three regions, one representing the portions

of both maps that overlap, and the other two representing the parts of the argument maps that do not overlap each other. Furthermore, the overlapping portion of the maps is labeled with the labels of both maps, and the non-overlapping portions carry the labels from their respective argument maps. A similar operation is the *superimposition* operation, in which the overlapping portions of the maps carry only the label from the first of the argument maps. Thus, one map is superimposed over the other such that the original of the first map remains in the result, and only the portion of the second map that does not intersect any part of the first remains. The *difference* operation is also similar to the overlay operation, except overlapping portions of the argument maps are removed from the result. This is similar to a difference operation between complex regions. Additionally, operations exist that operate solely on the attribute data of maps. For example, the *aggregate* operation computes aggregates of values of the neighborhood of regions. For instance, in a map of counties, an aggregate could be used to calculate the population of all counties that share a boundary with a specific county. Due to the large amount of map operations, we cannot present them all here and direct the reader to the referenced works for more information.

Implementing operations. In this section, we consider approaches to implementing the *map overlay* operation, since a variation of the overlay operation is required to compute nearly all spatial operations over maps. The implementation of operations over maps has developed in two distinct classes: operations based on a collection approach to modeling maps, and operations based on a data type approach to modeling maps. The collection approach to modeling maps considers a map as a collection of more simple data types. Therefore, operations based on this approach utilize techniques that help to manage collections of spatial data objects. The common factor in these implementations is that an operation is broken down into two steps: a *filter* step and a *refine* step. This idea is common in spatial index mechanisms used in spatial databases [70–75]. For example, to compute the spatial intersection of two collections of geometries, one must compute

the intersection of all pairs of geometries from the respective collections. However, we can reduce the amount of intersections that must be computed if we ignore pairs that obviously do not intersect. We can distinguish such pairs by maintaining a *minimum bounding rectangle* for each geometry in both collections. We can then do a simple test to see if the minimum bounding rectangles of two geometries intersect; if they do not, then the geometries do not intersect. Thus the filter step finds all pairs of geometries that may intersect based on a minimum bounding rectangle analysis. The refine step then performs an actual intersection algorithm on the pairs of geometries that pass through the filter step. This is the approach used in [76–78]. Although this method performs well for the collection approach to modeling maps, it does not apply to the data type approach to modeling maps in which the individual components of the map are not represented individually.

The data type approach to modeling maps represents a map as a single object. Therefore, algorithms to overlay maps in this approach cannot use the filter and refine steps that are utilized in the collection approach. Instead, the entire map geometry is considered as a whole. Computational geometry algorithms for this type of map overlay algorithm have been proposed [56, 79–82]. The main drawback to this type of algorithm is that regions whose minimum bounding boxes do not intersect are still included in the computation of spatial algorithms since the entire maps are considered as single objects. Although this approach is still asymptotically faster than the collection approaches (since all pairs of regions do not need to be computed, this approach has complexity identical to the type of plane sweep algorithm used), the collection approaches can be faster in situations when few map components intersect. This has lead to schemes such as partitioning maps to address this problem [81, 82].

2.2.3 Topological Relationships

The subject of topological relationships between maps has not yet been considered in the literature. Instead, models of topological relationships between the components of

maps, i.e., points, lines, and regions, have been studied extensively. These models were discussed previously.

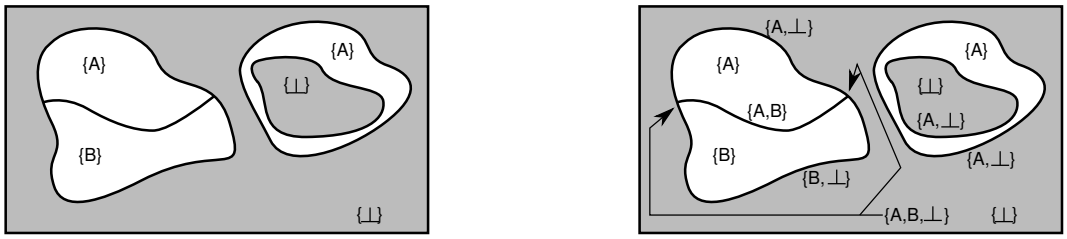
CHAPTER 3
AN INFORMAL OVERVIEW OF SPATIAL PARTITIONS

In this paper, we model maps as *spatial partitions*, as discussed in [65, 66, 83, 84]. The definition of spatial partitions is rather dense, so we begin by providing an intuitive description of them, and then present the formal definition in later sections.

A spatial partition, in two dimensions, is a subdivision of the plane into pairwise disjoint *regions* such that each region is associated with a *label* or *attribute* having simple or complex structure, and these regions are separated from each other by *boundaries*. The label of a region describes the thematic data associated with the region. All points within the spatial partition that have an identical label are part of the same region. Topological relationships are implicitly modeled among the regions in a spatial partition. For instance, neglecting common boundaries, the regions of a partition are always disjoint; this property causes maps to have a rather simple structure. Note that the *exterior* of a spatial partition (i.e., the unbounded face) is always labeled with the $\perp$ symbol. Figure 3-1A depicts an example spatial partition consisting of two regions.

We stated above that each region in a spatial partition is associated with a single attribute or label. A spatial partition is modeled by mapping Euclidean space to such labels. Labels themselves are modeled as sets of attributes. The regions of the spatial partition are then defined as consisting of all points which contain an identical label. Adjacent regions each have different labels in their interior, but their common boundary is assigned the label containing the labels of both adjacent regions. Figure 3-1B shows an example spatial partition complete with boundary labels.

In [65], operations over spatial partitions are defined based on known map operations in the literature. It is shown that all known operations over spatial partitions can be expressed in terms of three fundamental operations: intersection, relabel, and refine. Furthermore, the type of spatial partitions is shown to be closed under these operations,



A B

Figure 3-1. A sample spatial partition with two regions. A) The spatial partition annotated with region labels. B) The spatial partition with its region and boundary labels. Note that labels are modeled as sets of attributes in spatial partitions.

indicating that the type of spatial partitions is closed under all known operations over them.

CHAPTER 4
THE ABSTRACT MODEL OF SPATIAL PARTITIONS

Although the abstract model of spatial partitions has been presented in [65] and later refined in [83], these definitions required some modification in order to define both topological predicates over maps and the discrete model of spatial partitions. We present the modified definition here. We first introduce the mathematical notation and definitions required to formally define spatial partitions. Then, the formal mathematical type definition of spatial partitions is presented.

We now briefly summarize the mathematical notation used throughout the following sections. The application of a function $f : A \rightarrow B$ to a set of values $S \subseteq A$ is defined as $f(S) := \{f(x)|x \in S\} \subseteq B$. In some cases we know that $f(S)$ returns a singleton set, in which case we write $f[S]$ to denote the single element, i.e. $f(S) = \{y\} \Longrightarrow f[S] = y$. The inverse function $f^{-1} : B \rightarrow 2^A$ of f is defined as $f^{-1}(y) := \{x \in A|f(x) = y\}$. It is important to note that f^{-1} is a total function and that f^{-1} applied to a set yields a set of sets. We define the range function of a function $f : A \rightarrow B$ that returns the set of all elements that f returns for an input set A as $rng(f) := f(A)$.

Let (X, T) be a topological space [85] with topology $T \subseteq 2^x$, and let $S \subseteq X$. The *interior* of S, denoted by S°, is defined as the union of all open sets that are contained in S. The *closure* of S, denoted by $\overline{S}$ is defined as the intersection of all closed sets that contain S. The *exterior* of S is given by $S^- := (X - S)^\circ$, and the *boundary* or *frontier* of S is defined as $\partial S := \overline{S} \cap \overline{X - S}$. An open set is *regular* if $A = \overline{A}^\circ$ [86]. In this paper, we deal with the topological space $\mathbb{R}^2$.

A *partition* of a set S, in set theory, is a complete decomposition of the set S into non-empty, disjoint subsets $\{S_i|i \in I\}$, called blocks: (i) $\forall i \in I : S_i \neq \emptyset$, (ii) $\bigcup_{i \in I} S_i = S$, and (iii) $\forall i, j \in I, i \neq j : S_i \cap S_j = \emptyset$, where I is an index set used to name different blocks. A partition can equivalently be regarded as a total and surjective function $f : S \rightarrow I$. However, a spatial partition cannot be defined simply as a set-theoretic

partition of the plane, that is, as a partition of $\mathbb{R}^2$ or as a function $f : \mathbb{R}^2 \rightarrow I$, for two reasons: first, f cannot be assumed to be total in general, and second, f cannot be uniquely defined on the borders between adjacent subsets of $\mathbb{R}^2$.

4.1 Spatial Partitions

In [65], spatial partitions have been defined in several steps. First a *spatial mapping* of type A is a total function $\pi : \mathbb{R}^2 \rightarrow 2^A$. The existence of an undefined element $\perp_A$ is required to represent undefined labels (i.e., the exterior of a partition). Definition 1 identifies the different components of a partition within a spatial mapping. The labels on the borders of regions are modeled using the power set 2^A; a *border* of π (Definition 1(ii)) is a block that is mapped to a subset of A containing two or more elements, as opposed to a *region* of π (Definition 1(i)) which is a block mapped to a singleton set. The *interior* of π (Definition 1(iii)) is defined as the union of π's regions. The *boundary* of π (Definition 1(iv)) is defined as the union of π's borders. The *exterior* of π (Definition 1(v)) is the block mapped $\perp_A$. As an example, let π be the spatial partition in Figure 3-1 of type $X = \{A, B, \perp\}$. In this case, $rng(\pi) = \{\{A\}, \{B\}, \{\perp\}, \{A, B\}, \{A, \perp\}, \{B, \perp\}, \{A, B, \perp\}\}$. Therefore, the regions of π are the blocks labeled $\{A\}$, $\{B\}$, and $\{\perp\}$ and the boundaries are the blocks labeled $\{A, B\}$, $\{A, \perp\}$, $\{B, \perp\}$, and $\{A, B, \perp\}$. Figure 4-1 provides a pictorial example of the interior, exterior, and boundary of a more complex example map (note that the borders and boundary consist of the same points, but the boundary is a single point set whereas the borders are a set of point sets).

Definition 1. Let π be a spatial mapping of type A

(i) $\rho(\pi) := \pi^{-1}(rng(\pi) \cap \{X \in 2^A | \ |X| = 1\})$ (*regions*)

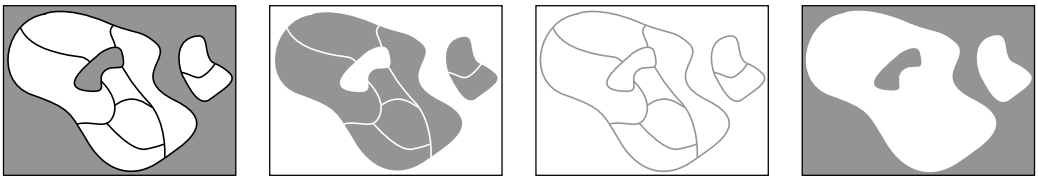
(ii) $\omega(\pi) := \pi^{-1}(rng(\pi) \cap \{X \in 2^A | \ |X| > 1\})$ (*borders*)

(iii) $\pi^\circ := \bigcup_{r \in \rho(\pi) | \pi[r] \neq \{\perp_A\}} r$ (*interior*)

(iv) $\partial\pi := \bigcup_{b \in \omega(\pi)} b$ (*boundary*)

(v) $\pi^- := \pi^{-1}(\{\perp_A\})$ (*exterior*)

A *spatial partition* of type A is then defined as a spatial mapping of type A whose regions are regular open sets [86] and whose borders are labeled with the union of labels



A B C D

Figure 4-1. A spatial partition π with two disconnected faces, one containing a hole. A) The interior (π°). B) The boundary ($\partial\pi$). D) The exterior (π^{-}). Note that the labels have been omitted in order to emphasize the components of the spatial partition.

of all adjacent regions. From this point forward, we use the term *partition* to refer to a spatial partition.

Definition 2. A *spatial partition* of type A is a spatial mapping π of type A with:

(i) $\forall r \in \rho(\pi) : r = \overline{r}^{\circ}$

(ii) $\forall b \in \omega(\pi) : \pi[b] = \{\pi[[r]] | r \in \rho(\pi) \wedge b \subseteq \partial r\}$

The remaining portion of the definition of spatial partitions requires the use of the refine operation over spatial partitions. This operation is formally defined in Section 4.2, and so we provide an intuitive definition here. The refine operation over spatial partitions uniquely identifies the connected components of a partition. Recall that two regions in a partition can share the same label if they are disjoint or meet at a point. Given a partition π containing multiple regions with the same label, the operation *refine*(π) returns a partition with identical structure to π, but with every region having a unique label. This is achieved by appending an integer to the label of each region that shares a label with another region. Figure 4-2 shows an example partition and the same partition after performing a refine operation. Note that the notation $(A, 1)$ indicates that the integer 1 has been appended to label A.

The boundary of a spatial partition implicitly imposes a graph on the plane. Specifically, the boundaries form an undirected planar graph. The edges of the graph are the points mapped to the boundaries between two regions. The vertices of the graph are the points mapped to boundaries between three or more regions. We identify edges

A B

Figure 4-2. The application of the refine operation to a spatial partition. A) A spatial partition with two regions and its boundary and region labels. B) The result of the refine operation on Figure **A**.

and vertices based on the cardinality of their labels. However, due to degenerate cases, we must use the refinement of a partition to identify these features. We define the set of edges and vertices imposed on the plane by a spatial partition as follows:

Definition 3. *Boundary points of a spatial partition* π *are classified as being a* vertex *or as being part of an* edge *by examining the refinement* $\sigma = refine(\pi)$ *as follows:*

(i) $\epsilon(\pi) = \{b \in \omega : |\sigma[b]| = 2\}$

(ii) $\nu(\pi) = \{b \in \omega : |\sigma[b]| > 2\}$

4.2 Operations

Three basic spatial partition operations have been defined that can be used to form the formal definitions of all other known partition operations: *intersection*, *relabel*, and *refine*. These are the only operations we present in this paper since the number of map operations is large and all other operations can be formulated using these three. Each of these operations is closed over the set of valid spatial partitions, meaning that if valid partitions are supplied as arguments to these operations, a valid partition will be returned [65]. The intersection of two partitions π and σ, of types A and B respectively, returns a spatial partition of type $A \times B$ such that each interior point p of the resulting partition is mapped to the pair of labels $(\pi[p], \sigma[p])$, and all border points are mapped to the set of labels of all adjacent regions. Formally, the definition of intersection of two partitions π and σ of types A and B can be described in several steps. First, the regions of the resulting partition must be known. This can be calculated by a simple set intersection of

all regions in both partitions, since $\cap$ is closed on regular open sets.

$$\rho_{\cap}(\pi, \sigma) := \{r \cap s | r \in \rho(\pi) \wedge s \in \rho(\sigma)\}$$

The union of all these regions gives the interior of the resulting partition: $\iota_{\cap}(\pi, \sigma) := \cup_{r \in \rho_{\cap}(\pi,\sigma)} r$. Next, the spatial mapping restricted just to the interior is calculated by mapping each interior point $p \in I := \iota_{\cap}(\pi, \sigma)$ to the pair of labels given by π and σ:

$$\pi_I := \lambda p : I.\{(\pi[p], \sigma[p])\}$$

Finally, the boundary labels are derived from the labels of all adjacent regions. Let $R := \rho_{\cap}(\pi, \sigma)$, $I := \iota_{\cap}(\pi, \sigma)$, and $F := \mathbb{R}^2 - I$. Then we have:

$$\mathit{intersection} : [A] \times [B] \rightarrow [A \times B]$$
$$\mathit{intersection}(\pi, \sigma) := \pi_I \cup \lambda p : F.\{\pi_I[[r]] | r \in R \wedge p \in \overline{r}\}$$

Relabeling a partition π of type A by a function $f : A \rightarrow B$ is defined as $f \circ \pi$, i.e., in the resulting partition of type B each point p is mapped to $f(\pi(p))$ (recall that $\pi(p)$ yields a singleton set, e.g. $\{a\}$, and that f applied to this yields the singleton set $\{f(a)\}$).

$$\mathit{relabel} : [A] \times (A \rightarrow B) \rightarrow [B]$$
$$\mathit{relabel}(\pi, f) := \lambda p : \mathbb{R}^2.f(\pi(p))$$

The refinement of a partition identifies the connected components of the partition. This is achieved by relabeling the connected components of a partition with consecutive numbers. A connected component of an open set S is a maximum subset $S \subseteq T$ such that any two points of T can be connected by a curve lying completely inside T [85]. Let $\gamma(r) = \{c_1, ...c_{n_r}\}$ denote the set of connected components in a region r. Then, *refine* can be defined in several steps. The regions of the resulting partition are the connected components of all regions of the original partition:

$$p_{\gamma}(\pi) := \bigcup_{r \in \rho(\pi)} \gamma(r)$$

The union of all these regions results in the interior of the resulting partition: $\iota_\gamma(\pi) := \cup_{r \in \rho_\gamma(\pi)} r$. This means that the set of interior and boundary points are not changed by refine.

We can now define the resulting partition on the interior:

$$\pi_I := \{(p, \{\pi[p], i)\}) | r \in \rho(\pi) \wedge \gamma(r) = \{c_1, ..., c_{n_r}\} \wedge i \in \{1, ..., n_r\} \wedge p \in c_i\}$$

Finally, we derive the labels for the boundary from the interior, much like the definition for intersection. Let $R := \rho_\gamma(\pi)$, $I := \iota_\gamma(\pi)$, and $F := \mathbb{R}^2 - I$. Then:

$$\textit{refine} : [A] \rightarrow [A \times \mathbb{N}]$$
$$\textit{refine}(\pi) := \pi_I \cup \lambda p : F.\{\pi_I[[r]] | r \in R \wedge p \in \overline{r}\}$$

4.3 Topological Relationships

In this section, we describe a method for deriving the topological relationships between a given pair of maps. We begin by describing various approaches to the problem, then outline our chosen method, and finally derive the actual relationships based on this method. Note that in this section, we refer to maps as *map geometries* because topological relationships consider only the spatial aspects of maps, and not their thematic attributes.

In order to define a complete, finite set of topological relationships between map geometries, we employ a method similar to that found in [46], in which the 9-intersection model is extended to describe complex points, lines, and regions. In Section 4.1, we defined a point set topological model of map geometries in which we identified the interior, exterior, and boundary point sets belonging to maps. Based on this model, we extend the 9-intersection model to apply to the point sets belonging to map objects. However, due to the spatial features of map geometries, the embedding space ($\mathbb{R}^2$), and the interaction of map geometries with the embedding space, some topological configurations are impossible and must be excluded. Therefore, we must identify topological constraints that must be satisfied in order for a given topological configuration to be valid. Furthermore, we must

identify these constraints such that all invalid topological configurations are excluded, and the complete set of valid configurations remains. We achieve this through a proof technique called *Proof-by-Constraint-and-Drawing*, in which we begin with the total set of 512 possible 9-intersection matrices, and determine the set of valid configurations by first providing a collection of topological constraint rules that invalidate impossible topological configurations, and second, validating all matrices that satisfy *all* constraint rules by providing a prototypical spatial configuration (i.e., the configurations can be drawn in the embedding space). Completeness is achieved because all topological configurations are either eliminated by constraint rules, or are proven to be possible through the drawing of a prototype. The remainder of this section contains the constraints, and the prototypical drawings of map geometries are shown in Table 4-1.

We identify eight constraint rules that 9IMs for map geometries must satisfy in order to be valid. Each constraint rule is first written in sentences and then expressed mathematically. Following each rule is the rationale explaining why the rule is correct. In the following, let π and σ be two spatial partitions.

Lemma 1. *Each component of a map geometry intersects at least one component of the other map geometry:*

$$(\forall C_\pi \in \{\pi^\circ, \partial\pi, \pi^-\} : \ C_\pi \cap \sigma^\circ \neq \varnothing \ \vee \ C_\pi \cap \partial\sigma \neq \varnothing \ \vee \ C_\pi \cap \sigma^- \neq \varnothing)$$
$$\wedge \ (\forall C_\sigma \in \{\sigma^\circ, \partial\sigma, \sigma^-\} : \ C_\sigma \cap \pi^\circ \neq \varnothing \ \vee \ C_\sigma \cap \partial\pi \neq \varnothing \ \vee \ C_\sigma \cap \pi^- \neq \varnothing)$$

Proof. Because spatial mappings are defined as total functions, it follows that $\pi^\circ \cup \partial\pi \cup \pi^- = \mathbb{R}^2$ and that $\sigma^\circ \cup \partial\sigma \cup \sigma^- = \mathbb{R}^2$. Thus, each part of π must intersect at least one part of σ, and vice versa. □

Lemma 2. *The exteriors of two map geometries always intersect:*

$$\pi^- \cap \sigma^- \neq \varnothing$$

Proof. The closure of each region in a map geomtry corresponds to a complex region as defined in [46]. Since complex regions are closed under the union operation, it follows that the union of all regions that compose a map geometry is a complex region,

whose boundary is defined by a Jordan curve. Therefore, every spatial partition has an exterior. Furthermore, in [65], the authors prove that spatial partitions are closed under intersection. Thus, the intersection of any two spatial partitions is a spatial partition that has an exterior. Therefore, the exteriors of any two spatial partitions intersect, since their intersection contains an exterior. □

Lemma 3. *If the boundary of a map geometry intersects the interior of another map geometry, then their interiors intersect:*

$$((\partial\pi \cap \sigma^\circ \neq \varnothing \Rightarrow \pi^\circ \cap \sigma^\circ \neq \varnothing) \ \wedge \ (\pi^\circ \cap \partial\sigma \neq \varnothing \Rightarrow \pi^\circ \cap \sigma^\circ \neq \varnothing))$$
$$\Leftrightarrow ((\partial\pi \cap \sigma^\circ = \varnothing \ \vee \ \pi^\circ \cap \sigma^\circ \neq \varnothing) \ \wedge \ (\pi^\circ \cap \partial\sigma = \varnothing \ \vee \ \pi^\circ \cap \sigma^\circ \neq \varnothing))$$

Proof. Assume that a boundary b of partition π intersects the interior of partition σ but their interiors do not intersect. In order for this to be true, the label of the regions on either side of b must be labeled with the empty label. According to the definition of spatial partitions, a boundary separates two regions with different labels; thus, this is impossible and we have a proof by contradiction. □

Lemma 4. *If the boundary of a map geometry intersects the exterior of a second map geometry, then the interior of the first map geometry intersects the exterior of the second:*

$$((\partial\pi \cap \sigma^- \neq \varnothing \Rightarrow \pi^\circ \cap \sigma^- \neq \varnothing) \ \wedge \ (\pi^- \cap \partial\sigma \neq \varnothing \Rightarrow \pi^- \cap \sigma^\circ \neq \varnothing))$$
$$\Leftrightarrow ((\partial\pi \cap \sigma^- = \varnothing \ \vee \ \pi^\circ \cap \sigma^- \neq \varnothing) \ \wedge \ (\pi^- \cap \partial\sigma = \varnothing \ \vee \ \pi^- \cap \sigma^\circ \neq \varnothing))$$

Proof. This proof is similar to the previous proof. Assume that the boundary b of partition π intersects the exterior of partition σ but the interior of π does not intersect the exterior of σ. In order for this to be true, the label of the regions on either side of b must be labeled with the empty label. According to the definition of spatial partitions, a boundary separates two regions with different labels; thus, this is impossible and we have a proof by contradiction. □

Lemma 5. *If the boundaries of two map geometries are equivalent, then their interiors intersect:*

$$(\partial\pi = \partial\sigma \Rightarrow \pi^\circ \cap \sigma^\circ \neq \varnothing) \Leftrightarrow (c \Rightarrow d) \Leftrightarrow (\neg c \ \vee \ d) \text{ where}$$
$$\begin{aligned} c = \ & \partial\pi \cap \partial\sigma \neq \varnothing \ \wedge \ \pi^\circ \cap \partial\sigma = \varnothing \ \wedge \ \partial\pi \cap \sigma^\circ = \varnothing \\ & \wedge \ \partial\pi \cap \sigma^- = \varnothing \ \wedge \ \pi^- \cap \partial\sigma = \varnothing \\ d = \ & \pi^\circ \cap \sigma^\circ \neq \varnothing \end{aligned}$$

Proof. Assume that two spatial partitions have an identical boundary, but their interiors do not intersect. The only configuration which can accommodate this situation is if one spatial partition's interior is equivalent to the exterior of the other spatial partition. However, according to Lemma 2, the exteriors of two partitions always intersect. If a partition's interior is equivalent to another partition's exterior, then their exteriors would not intersect. Therefore, this configuration is not possible, and the interiors of two partitions with equivalent boundaries must intersect. □

Lemma 6. *If the boundary of a map geometry is completely contained in the interior of a second map geometry, then the boundary and interior of the second map geometry must intersect the exterior of the first, and vice versa:*

$$(\partial\pi \subset \sigma^\circ \Rightarrow \pi^- \cap \partial\sigma \neq \varnothing \ \wedge \ \pi^- \cap \sigma^\circ \neq \varnothing) \Leftrightarrow (\neg c \ \vee \ d) \text{ where}$$
$$\begin{aligned} c = \ & \partial\pi \cap \sigma^\circ \neq \varnothing \ \wedge \ \partial\pi \cap \partial\sigma = \varnothing \ \wedge \ \partial\pi \cap \sigma^- = \varnothing \\ d = \ & \pi^- \cap \partial\sigma \neq \varnothing \ \wedge \ \pi^- \cap \sigma^\circ \neq \varnothing \end{aligned}$$

Proof. If the boundary of spatial partition π is completely contained in the interior of spatial partition σ, it follows from the Jordan Curve Theorem that the boundary of σ is completely contained in the exterior of π. By Lemma 4, it then follows that the interior of σ intersects the exterior of π. □

Lemma 7. *If the boundary of one map geometry is completely contained in the interior of a second map geometry, and the boundary of the second map geometry is completely contained in the exterior of the first, then the interior of the first map geometry cannot intersect the exterior of the second and the interior of the second map geometry must intersect the exterior of the first and vice versa:*

$$((\partial\pi \subset \sigma^\circ \ \wedge \ \pi^- \supset \partial\sigma) \Rightarrow (\pi^\circ \cap \sigma^- = \varnothing \ \wedge \ \pi^- \cap \sigma^\circ \neq \varnothing))$$
$$\Leftrightarrow (c \Rightarrow d) \Leftrightarrow (\neg c \ \vee \ d) \text{ where}$$
$$c = \ \partial\pi \cap \sigma^\circ \neq \varnothing \ \wedge \ \partial\pi \cap \partial\sigma = \varnothing \ \wedge \ \partial\pi \cap \sigma^- = \varnothing$$
$$\wedge \ \pi^\circ \cap \partial\sigma = \varnothing \ \wedge \ \pi^- \cap \partial\sigma \neq \varnothing$$
$$d = \ \pi^\circ \cap \sigma^- = \varnothing \ \wedge \ \pi^- \cap \sigma^\circ \neq \varnothing$$

Proof. We construct this proof in two parts. According to Lemma 6, if $\partial\pi \subseteq \sigma^\circ$, then $\pi^- \cap \sigma^\circ \neq \varnothing$. Now we must prove that π° cannot intersect σ^-. Since $\partial\pi \subset \sigma^\circ$, it follows that π° intersects σ°. Therefore, the only configuration where $\pi^\circ \cap \sigma^- \neq \varnothing$ can occur is if σ contains a hole that is contained by π. However, in order for this configuration to exist, the $\partial\sigma$ would have to intersect the interior or the boundary of π. Since the lemma specifies the situation where $\pi^- \supset \partial\sigma$, this configuration cannot exist; thus, the interior of π cannot intersect the exterior of σ. □

Lemma 8. *If the boundary of a map geometry is completely contained in the exterior of a second map geometry and the boundary of the second map geometry is completely contained in the exterior of the first, then the interiors of the map geometries cannot intersect:*

$$((\partial\pi \subset \sigma^- \ \wedge \ \pi^- \supset \partial\sigma) \Rightarrow (\pi^\circ \cap \sigma^\circ = \varnothing))$$
$$\Leftrightarrow (c \Rightarrow d) \Leftrightarrow (\neg c \ \vee \ d) \text{ where}$$
$$c = \ \partial\pi \cap \sigma^\circ = \varnothing \ \wedge \ \partial\pi \cap \partial\sigma = \varnothing \ \wedge \ \partial\pi \cap \sigma^- \neq \varnothing$$
$$\wedge \ \pi^\circ \cap \partial\sigma = \varnothing \ \wedge \ \pi^- \cap \partial\sigma \neq \varnothing$$
$$d = \ \pi^\circ \cap \sigma^\circ = \varnothing$$

Proof. The lemma states that the interiors of two disjoint maps do not intersect. Without loss of generality, consider two map geometries that each consist of a single region. We can consider these map geometries as complex region objects. If two complex regions are disjoint, then their interiors do not intersect. We can reduce any arbitrary map to a complex region by computing the spatial union of its regions. It follows that because the

interiors of two disjoint regions do not intersect, the interiors of two disjoint maps do not intersect. □

Using a simple program to apply these eight constraint rules reduces the 512 possible matrices to 49 valid matrices that represent topological relationships between two maps geometries. The matrices and their validating prototypes are depicted in Table 4-1. Finally, we summarize our results as follows:

Theorem 1. *Based on the 9-intersection model for spatial partitions, 49 different topological relationships exist between two map geometries.*

Proof. The argumentation is based on the *Proof-by-Constraint-and-Drawing* method. The constraint rules, whose correctness has been proven in Lemmas 1 to 8, reduce the 512 possible 9-intersection matrices to 49 matrices. The ability to draw prototypes of the corresponding 49 topological configurations in Table 4-1 proves that the constraint rules are complete. □

Table 4-1. The first 42 valid matrices and protoypical drawings representing the possible topological relationships between maps. Each drawing shows the interaction of two maps, one map is medium-grey and has a dashed boundary, the other is light-grey and has a dotted boundary. Overlapping map interiors are dark-grey, and overlapping boundaries are drawn as a solid line. For reference, the figure for matrix 41 shows two disjoint maps and the figure for matrix 1 shows two equal maps.

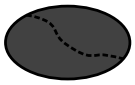
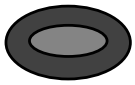
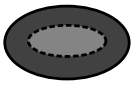
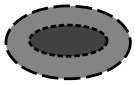
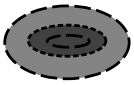
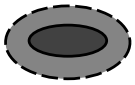
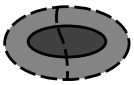
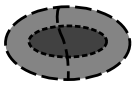
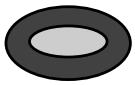
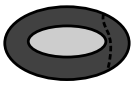
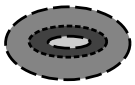
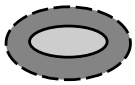
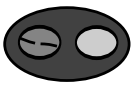
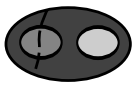
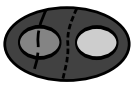
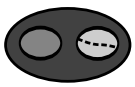
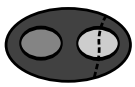
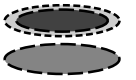
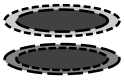
Matrix 1	Matrix 2	Matrix 3	Matrix 4	Matrix 5	Matrix 6
					
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Matrix 7	Matrix 8	Matrix 9	Matrix 10	Matrix 11	Matrix 12
					
$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$
Matrix 13	Matrix 14	Matrix 15	Matrix 16	Matrix 17	Matrix 18
					
$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$
Matrix 19	Matrix 20	Matrix 21	Matrix 22	Matrix 23	Matrix 24
					
$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$
Matrix 25	Matrix 26	Matrix 27	Matrix 28	Matrix 29	Matrix 30
					
$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
Matrix 31	Matrix 32	Matrix 33	Matrix 34	Matrix 35	Matrix 36
					
$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
Matrix 37	Matrix 38	Matrix 39	Matrix 40	Matrix 41	Matrix 42
					
$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

Table 4-2. The final 7 valid matrices and protoypical drawings representing the possible topological relationships between maps.

Matrix 43	Matrix 44	Matrix 45	Matrix 46
			
$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
Matrix 47	**Matrix 48**	**Matrix 49**	
			
$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	

CHAPTER 5
QUERYING SPATIAL PARTITIONS: THE USER VIEW OF MAPS IN SPATIAL DATABASES

The previous chapter defined an abstract, mathematical model of maps in the form of spatial partitions. This model defines the type of spatial partitions as well as the semantics of operations over spatial partitions. In this chapter, we show how the type of spatial partitions and the operations over them can be queried and manipulated by a user. We approach this problem at two levels: first, we define a method to express map queries that is independent of database concepts such as SQL, and second, we show how SQL constructs can be extended to provide map querying capabilities, and implement the constructs provided in the first step. We assume the existence of a spatial database that is aware of a type called *map2D*, which represents the type of spatial partitions. Furthermore, we assume that the database is aware of operations and topological predicates over spatial partitions. In following sections, we discuss the types of queries that can be performed over the database type of map2D, provide a suitable, high-level database representation for the type map2D, and show how map2D instances can be created and queried in such a database.

5.1 Types of Map Queries

In Section 2, we discussed traditional spatial data types and their operations. Recall that these traditional spatial types only represent a geometry, and do not integrate thematic information. Furthermore, the spatial operations over them are purely geometric in nature and do not take into account thematic attributes (instead, thematic attributes are associated with a spatial object, but are considered separately from the object's geometry). This has lead to a relatively straightforward implementation of the traditional spatial types into spatial databases since queries over these types are restricted to involving only the geometry of a spatial object. For example, traditional spatial queries answer questions such as "return all regions that intersect a given query region", or "return all regions with an area greater than one thousand square miles". Furthermore,

attribute queries over traditional spatial types are straightforward since they deal only with attributes stored along with a spatial object (i.e., the attributes are separate from the spatial object. The spatial object itself consists solely of a geometry). Therefore, queries such as "return all regions that have a population greater than one thousand" do not need to access a spatial object, but only an attribute stored separately from a spatial object.

As we have seen, spatial partitions differ significantly from the traditional spatial types due to their integration of both spatial and thematic information. Thus, maps can be queried in different ways than the traditional spatial types. For example, a map query may ask to return maps based on the attributes associated with the regions that make up the map. The traditional spatial types have no method to associate attributes with individual components of spatial objects. An example map query would be to "return all maps that have a region named Florida".

Traditional spatial types have two types of queries associated with them, spatial queries and attribute queries. As we have seen, maps allow for new types of queries due their more complex structure and the integration of attribute data in the data type definition. Thus, instead of being defined merely as a geometry, a map has four components which may be involved in queries: (i) a map geometry, (ii) components in the form of *regions* that compose the geometry, (iii) attributes associated with each region, and (iv) possibly attributes associated with the entire map as well. Therefore, we classify map queries into four classes of queries: *map queries*, which look at the map as a whole; *map attribute queries*, which are similar to attribute queries over traditional spatial objects in that they deal with attributes stored separately from a map object; *component attribute queries*, which deal with the attributes associated with regions in a map; and *component queries*, which deal with the regions that compose a map. In the following sections we discuss each query type in more detail and show how queries of each type can be expressed. Note that map attribute queries are identical to thematic queries involving

the traditional spatial types; therefore, they are simply traditional queries that do not interact with a map or its components, and we do not discus them further.

5.2 Map Query Language: A High-level Query Language for Maps

In this section, we approach the concept of querying maps from the perspective of a user who is not familiar with databases or SQL. Therefore, we treat a map as an abstract data type that hides the details of its implementation from the user. This has the consequence that such a language is independent of the underlying implementation, and can be implemented over any type of storage medium, such as a database, file system, or computer memory. In the following, we develop the Map Query Language (MQL) incrementally by first examining the data model of maps used in MQL, and then showing how the various classes of map queries can be expressed in MQL.

5.2.1 Data Model

As mentioned above, in this section we treat a map as an abstract data type and make no assumptions as to the implementation or storage medium associated with a map implementation. However, we cannot simply use the definition of spatial partitions as given in Chapter 4 since it is too general for creating a query language meant for users with little technical knowledge of querying mechanisms. Specifically, we must pose some constraints on map labels. Therefore, for the purposes of MQL, we define a new data type called *label* which enforces constraints over the structure and contents of spatial partition labels for querying with MQL.

The first constraint for labels that we require is that labels can only contain data that corresponds to a set of defined types. In general, such a set can consist of any defined type, but for illustration purposes, we will assume that the attributes contained in labels must be of type *string* or *integer*, where a string is a sequence of characters, and an integer is a number belonging to the set of integers. The second constraint is that all labels in a map must be defined by a single *label component type*, which is defined as a list such that

each item in the list consists of a type associated with a *label component attribute* (i.e., an identifier that uniquely identifies the attribute in the label). More formally:

Definition 4. *Let* $T = \{string, integer\}$ *be the set of valid label component types for attributes contained in a label such that a string is a sequence of* characters *and an integer is a number* $x \in \mathbb{Z}$.

Let IDS be the set of all possible values for a label component attribute A Label Type *is a tuple of pairs* $LS = (a_1 : b_1, \ldots, a_n : b_n)$ *where* $a_i \in T$ *and* $b_i \in IDS$.

The MQL *create label type* statement that allows a user to create label types and associate them with an identifier:

Definition 5. *Let IDS be the set of valid label component attributes and T be the set of valid label component types. The* create label statement *can then be used to create labels as follows:*

CREATE LABEL TYPE l $(a_1 : b_1, \ldots, a_n : b_n)$ *where* $a_i \in T$ *and* $l, b_i \in IDS$

Given a label type, a label is then a tuple containing a value of the appropriate label component type for each pair in the associated label type. Furthermore, each value can be identified by the label component attribute defined in the label type:

Definition 6. *Let LT be a* label type *defined by CREATE LABEL TYPE LT* $(a_1 : b_1, \ldots, a_n : b_n)$ *where* $a_i \in T$ *and* $l, b_i \in IDS$.

A label of type LT is then defined as a tuple $L = (c_1 \in a_1, \ldots, c_n \in a_n)$

Figure 5-1 depicts two sample maps that satisfies these label constraints. The label type for the map shown in Figure 5-1A is $(string : Crop)$, and the label structure for the map shown in Figure 5-1B is $(integer : Avg\_Temp, integer : Avg\_Rain)$. These label types can be created by a user with the following statements, respectively: *CREATE LABEL TYPE field_crop* $(string : Crop)$ and *CREATE LABEL TYPE climate_data* $(integer : Avg\_Temp, integer : Avg\_Rain)$. These maps will be used throughout this section to illustrate MQL concepts.

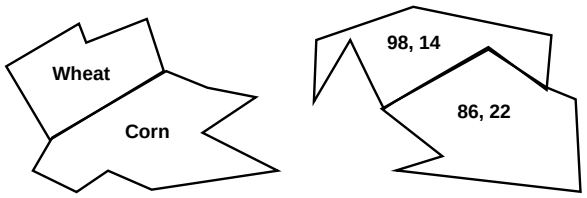


Figure 5-1. Two sample maps. A) A map with labels consisting of a single string named *crop*. B)A map with labels consisting of a pair of integers, indicating the average temperature and rainfall for each region, named *Avg_Temp* and *Avg_Rain*, respectively.

Recall that a spatial partition is defined by a spatial mapping $\pi := \mathbb{R}^2 \rightarrow 2^A$, where A is the set of all region labels, with some restrictions (Definition 2). Due to the restrictions placed on labels above, we only consider the subset of spatial partitions whose spatial mapping restricts the set A to labels that satisfy a given label type. Thus, given label type LT, we consider spatial partitions defined by the spatial mapping $\pi_\lambda := \mathbb{R}^2 \rightarrow 2^{LT}$. For example, the map shown in Figure 5-1A has a label type $field\_crop = (string : Crop)$ and a spatial mapping $\pi_\lambda : \mathbb{R}^2 \rightarrow 2^{field\_crop}$. We refer to the type of spatial partitions whose set of region labels is constrained by a label type LT as $map2D_{LT}$. We refer to the type of spatial partitions consisting of maps with any valid label type as *map2D*. MQL is a mechanism to pose queries over maps of the type map2D.

In order to show how maps can be created in MQL, we introduce some notation and new operations. These operations and notations facilitate the expression of queries in MQL. Specifically, these operations provide the ability to create new map2D instances and add regions to map2D instances:

Definition 7.

Given a label type LT and an identifier M, the MQL map create statement *returns a new map with no regions:*

CREATE MAP M (LT)

The add operation takes a map2D instance defined by a label type LT, a region to be added the map2D instance, and a label and adds the argument region and label to the map2D instance:

$add := map2D_{LT} \times region \times LT$

Furthermore, we assume an assignment operator exists for the type map2D in the form of: '='.

Therefore, in order to create the maps in Figure 5-1, we would use the following sequence of MQL statements:

CREATE LABEL TYPE field_crop(*string* : *Crop*)

CREATE LABEL TYPE climate_data(*integer* : *Avg_Temp*, *integer* : *Avg_Rain*)

CREATE MAP M1(*field_crop*)

CREATE MAP M2(*climate_data*)

Once we have created maps, we can then add regions to them. Let r, s, t, and u be regions, each corresponding to a region in the maps shown in Figure 5-1. We can then add these regions to their respective maps as follows:

$$add(M1, r, (\text{`}wheat\text{'}))$$
$$add(M1, s, (\text{`}corn\text{'}))$$
$$add(M2, t, (98, 14))$$
$$add(M2, u, (86, 22))$$

5.2.2 MQL Querying Syntax

Based on the type of map2D, we now define the constructs necessary to pose queries in MQL. A MQL statement consists of three components called *clauses*: the *FIND* clause, the *IN* clause, and the *THAT* clause. The *FIND* clause indicates what will be returned by the query. MQL statements can return either map2D instances or labels. Therefore, the *FIND* clause contains either the keyword *MAP*, to indicate that a map will be returned by

the query, or a label type that indicates the structure of labels that will be returned by the query.

The *IN* clause contains a list of map2D instances that query a will be performed over. In MQL implementations, this list will correspond to a database relation or file system structure, but for the purposes of defining a user concept that is not tied to a specific implementation, we will represent it simply as a list.

Finally, the *THAT* clause contains a boolean expression that is evaluated over each of the map objects in the *IN* clause, their labels, and/or their component regions. In order to achieve this, we introduce two functions: *CONTAINS_LABEL()* and *CONTAINS_REGION()* that evaluate a Boolean expression and return the result. The *CONTAINS_LABEL()* operation contains a boolean expression involving labels of the maps in the *IN* clause. The *CONTAINS_REGION()* operation contains a boolean expression involving the component regions of the maps in the *IN* clause. The signatures for these functions are:

Definition 8. *The boolean expression given as an argument to the* CONTAINS_LABEL *function will be applied to the label associated with each region in a map. The boolean expression given as an argument to the* CONTAINS_REGION *function will be applied to each component region of a given map.*

$CONTAINS\_LABEL := boolean\_expression \rightarrow \mathbb{B}$

$CONTAINS\_REGION := boolean\_expression \rightarrow \mathbb{B}$

The *IN* clause of an MQL expression can contain a boolean expression consisting of predicates over maps, *CONTAINS_LABEL()* statements, and *CONTAINS_REGION()* statements. We refer to such a boolean expression as a *criteria*. Therefore, a MQL statement returns maps or labels from a list of maps that satisfy a given criteria

Therefore, the structure of an MQL query is as follows:

Definition 9. *Let LT be a label type. Let the notation $x|y$ mean that x or y may be used, but not both*

FIND *MAP | LT*

IN $M_1, \ldots M_n \in$ *map2D*

THAT *criteria*

It is straightforward to consider MQL statements where maps are returned, since any map in the *IN* clause that satisfies the given criteria will be returned. However MQL statements that return labels are not as straightforward. If a label structure is given in the *FIND* clause, then the query will return all the labels from every map in the *IN* clause that satisfies the criteria. However, labels will only be included in the result that strictly conform to the label structure given (i.e., the labels have the same type and identifier as given in the label structure in the *FIND* clause of the MQL query). In the following subsections, we show how the different classes of map queries can be expressed in MQL.

5.2.3 Querying Maps

At this point, the user has the ability to create maps and add regions and their corresponding labels to maps. In the following sections, we examine how a user can pose queries over maps and execute map operations using MQL constructs. We provide a discussion of each class of map queries identified previously, providing sample queries and showing how they can be expressed in MQL. New operations are introduced as they are needed.

5.2.3.1 Map queries

Map queries, as stated previously, are queries that consider a map as a whole, often involving map operations. For example, queries that return all maps whose area is larger than one thousand square feet or that calculate the overlay of a pair of maps, or that return maps based on a topological relationship fall into this category. In general, queries of this type can be expressed in a straightforward manner as long as the appropriate map

operation is defined. For example, in order to find all maps in a table with an area greater than one thousand, the user would use an area operation:

Query 5.2.1. *FIND MAP IN M1, M2 THAT area( ) > 1000*

Queries involving geometric operations, such as overlay, can be expressed by simply calling the appropriate operations:

Query 5.2.2. *M3 = Overlay(M1, M2)*

Finally, if we assume that we have a topological predicate between maps named *intersects* available, which returns a value of *true* if two maps intersect, we can discover all maps that intersect a query map as follows:

Query 5.2.3. *FIND MAP IN M1, M2 THAT intersects(M3)*

5.2.3.2 Component attribute queries

The class of component attribute queries does not correlate with any type of queries over the traditional spatial types. This is because the ability to associate attribute data with the components of spatial objects is unique to spatial partitions. For example a user may want to find the names of all maps that contain a region that contains the crop wheat. Such a query makes use of the *CONTAINS_LABEL* function in its criteria to look at the label of each region in each map in the *IN* clause:

Query 5.2.4. *FIND MAP IN M1, M2 THAT CONTAINS_LABEL(Crop = 'wheat')*

Because the map M1 contains a region with a label containing an identifier "Crop" and the value "wheat", the *CONTAINS_LABEL* with the supplied argument will evaluate to *true* for map M1, and it will be returned.

A more complex query would return attributes from regions in a map that satisfy some criteria. For example, a user may wish to find the average temperature for all regions in maps that have an average rainfall greater than 20 inches. Recall that if a label type is specified in the *FIND* clause of a MQL query, all labels will be returned for any map

that satisfies the criteria in the MQL statement. Therefore, we require a mechanism to isolate only the labels that satisfy the given criteria in a map. We achieve this by making the observation that every region in a map, when considered separately from the map, forms a *singleton map*. A singleton map is map that contains only a single region, which means it has only a single region label. If each region of every map in the *IN* clause can be considered separately, then we can return the labels only for the singleton maps that satisfy the criteria. In order to achieve this, we assume the existence of a map operation $Expand := map2D \rightarrow 2^{map2D}$, which takes a map2D instance, and returns a set of singleton maps such that each map in the result set corresponds to a region in the original map. This function can then be used in the *IN* clause of MQL. Therefore, we can express the desired query as:

Query 5.2.5. *FIND ((integer: Avg_Temp)) IN Expand(M1), Expand(M2) THAT CONTAINS_LABEL(Avg_Rain > 20)*

5.2.3.3 Component queries

Component queries allow users to query maps based on their components, i.e. regions. These are similar in concept to component attribute queries, but deal with the region geometries contained in a map instead of the region attributes. For example, if a user wishes to return all maps that contain a region with an area greater than one thousand square miles, they could use an *area* operator defined for regions as follows:

Query 5.2.6. *FIND MAP IN M1, M2 THAT CONTAINS_REGION(area() > 1000)*

Another useful query is to extract regions from a map that satisfy some predicate. For example, to find all regions in a map with area greater than one thousand square miles, a user could pose the following query:

Query 5.2.7. *FIND MAP IN Expand(M1) THAT area() > 1000*

5.3 Querying Maps in Databases Using SQL

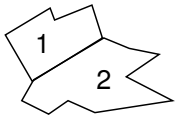
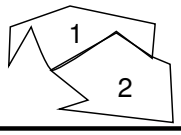
In the previous section, we showed how maps can be queried using the MQL language. In this section, we consider querying maps in SQL and show how the concepts from MQL can be expressed in SQL. We assume the existence of a database that is aware of the type *map2D*, as defined previously, and the operations and predicates over it.

5.3.1 Data Model

The data model for maps in spatial databases should support the four classes of queries discussed above. Therefore, we propose a data model for spatial partitions in a database to be composed of two components: the geometric component and the label component. The geometric component consists of the map geometry, and is implemented as a column type named *map2D*. This is similar in concept to the implementation of traditional spatial types in databases. The difference between map2D and the implementation of traditional types is that each region in the map is associated with a region identifier which will be used to associate each region with its attribute data.

The second component to the map2D type consists of the attributes for each region. We assume that each instance of a map2D in a column is associated with a *label table*, which contains the attributes for each region in the map. The only constraint we place on the label table is that it must contain a column of type integer named *region_id* that is used to associate each tuple in the label table with a region in its corresponding map. By storing the region attributes in a separate database table, instead of within the map2D data type itself, we allow the user to perform queries over the attributes associated with regions using standard SQL. This facilitates the the class of component attribute queries described above. Finally, we assume that a map instance stores the information necessary to identify its corresponding label table. Figure 5-2 shows an example instance of a table containing a column of type map2D which contains two tuples, and the label tables associated with the map stored in each tuple.

MapTable

ID: int	Name: str	Geom: map2D
1	Map_A	
2	Map_B	

Map1AttributeTable

region_id: int	Crop: str
1	Wheat
2	Corn

Map2AttributeTable

region_id: int	Avg Temp: int	Avg Rain: int
1	98	14
2	86	22

Figure 5-2. A relation containing a map2D column and the associated label tables. The table Map1AttributeTable is associated with the map with ID equal to 1 in the table MapTable, and the table Map2AttributeTable is associated with the map with ID equal to 2 in the table MapTable.

5.3.2 Creating Maps

In this section we show how we can extend the SQL *CREATE* and *INSERT* constructs to accommodate the creation of tables containing map columns and tuples containing maps. We assume that the database is aware of the data model of maps as discussed in the previous section, specifically the type *map2D*. We now show how a user can create the tables depicted in Figure 5-2.

Consider a user who wishes to create a map in a database. We assume that this user begins with an empty database (i.e.., a database containing no tables) and performs the necessary steps to create a map2D instance within the database. The first step the user must take in order to create a map is to create a table in which the map will be stored. Since the database is aware of the map2D type, we can simply use a create table command as shown below:

Statement 5.3.1. *CREATE TABLE MapTable(ID: int, Name: str, Geom: map2D)*

Note that the above statement creates only the table named "MapTable" and does not create any tuples. In order to associate labels with the regions in a map, a map must be associated with a label table. Therefore, the user must create a label table for a map

in order to associate the map with the table. The creation of label table is analogous to defining a label type for a map in MQL. A label table will be associated with a particular map, and its schema will define the structure and types of the attributes stored in it (i.e., each tuple is analogous to a label). Recall that we impose the restriction on a label table that it must contain a column of type integer with the name *region_id* that will be used to associate each entry in the label table with a region in the map. For example, to create the table named "Map1AttributeTable" in Figure 5-2, a user would use the following statement:

Statement 5.3.2. *CREATE TABLE Map1AttributeTable(region_id: int, crop: str)*

Given a label table, a user can then create a map whose regions are associated with the labels in the label table through an *insert* statement. We assume the existence of a map constructor, which we indicate as *map2D(string labelTable)*, that takes the name of a label table as an argument and returns an empty map object that is associated with the label table. In other words, the map object stores the name of the label table internally. Thus, a map can be inserted into the "MapTable" by:

Statement 5.3.3. *INSERT INTO MapTable VALUES(1, 'Map_A', map2D('Map1AttributeTable'))*

5.3.3 Querying Maps

In this section, we show how the classes of queries given above can be expressed over the given data model for maps in spatial databases. For each class of queries, we show how SQL can be extended to express the MQL queries that have been given previously.

In order to express MQL queries in SQL, we make some general observations. First, the three clauses of an MQL statement, *FIND*, *IN*, *THAT*, correspond to the SQL statement clauses of *SELECT*, *FROM*, and *WHERE*, respectively. In the SELECT clause, a user can return a map by indicating that values of a column of type map2D are returned. The list of maps found in the *IN* clause of MQL will be expressed as a relation

in the *FROM* clause of SQL. Finally, the boolean expression in the *THAT* clause of MQL will be expressed in the *WHERE* clause of SQL. In later sections, we will show how the functions introduced in MQL can be expressed in SQL.

Note that when a query that returns labels is expressed in MQL, a label structure is given in the *FIND* clause. This is necessary since maps in the *IN* clause may have different label structures, and two maps may have labels containing attributes with the same identifier, but different types. By expressing the desired label structure in the query, such cases can be resolved since the type of the desired attribute is clearly expressed. Therefore, we introduce a notation to achieve the same effect in SQL. If there is some ambiguity about the type of an attribute in the *SELECT* clause of an SQL statement because it refers to an attribute found in a map label, then the type must explicitly provided in the query in the form of (*type* : *identifier*).

5.3.3.1 Map queries

Map queries consider a map as a whole, often involving map operations. For instance, discovering maps with a total area greater than a given amount, or computing map operations such as overlay or intersection fall into this category. In Queries 5.2.1 - 5.2.3, we showed how various map queries can be expressed in MQL. Expressing the same queries in SQL is relatively straightforward since we are treating maps as column types. For example, Query 5.2.1 can be expressed as:

Query 5.3.1. *SELECT Geom FROM MapTable WHERE Geom.area() > 1000*

Queries involving map operations are slightly more complex in SQL than MQL since the argument maps must be identified explicitly in the relations in which they reside. Query 5.2.2 might be expressed as:

Query 5.3.2. *SELECT Overlay(M1.Geom, M2.Geom) FROM MapTable M1, MapTable M2 WHERE M1.ID = 1 AND M2.ID = 2*

Similarly, topological predicates between maps, as expressed in MQL in Query 5.2.3 can be expressed as:

Query 5.3.3. *SELECT M1.Geom FROM MapTable M1, MapTable M2 WHERE M2.ID = 3 AND Intersects(M1.Geom, M2.Geom)*

Note that these queries can be expressed in a relatively straightforward manner, since they are similar in concept to traditional spatial queries over the traditional spatial types. In other words, we are querying the spatial object as a whole, and simply using operations and predicates that take instances of the spatial types as arguments. This is the approach taken in querying the traditional spatial types.

5.3.3.2 Component attribute queries

Expressing component attribute queries in SQL is conceptually more simple than expressing them in MQL since the label tables are implemented as relations. This means that we can use standard SQL to query them. For example, a user may wish to know the average temperature of regions whose average rainfall is greater than 20 inches per year for regions in Map_B in Figure 5-2. Such a query could be expressed as:

Query 5.3.4. *SELECT Avg_Temp FROM Map2AttributeTable WHERE Avg_Rain > 20*

This works if the user is aware of the name of a label table for a given map. If the user does not know this information, we must provide a means of accessing a map's label table through the map type itself. In MQL, this is achieved through the use of the *CONTAINS_LABEL* operation. Therefore, we must extend SQL to include an operation that achieves the same functionality. We introduce an operation called *GetLabelTable* which takes a map2D instance as an argument and returns the relation consisting of the label table associated with the given map. Given such an operation, we can then express component attribute queries given instances of the map2D type:

Query 5.3.5. *SELECT Avg_Temp FROM GetLabelTable(SELECT GEOM FROM MapTable WHERE ID = 2) WHERE Avg_Rain > 20*

However, querying a label table directly does not allow us to express queries such as Query 5.2.4 and Query 5.2.5. The query above only queries the label table of a single map, whereas Query 5.2.4 and Query 5.2.5 execute over multiple maps. In order to achieve this in SQL, we require the use of a *correlated attribute query*. A correlated attribute query is similar in concept to a correlated subquery in that a subquery is run repeatedly for multiple values specified in its containing query. Correlated attribute queries typically require that attribute types be specified in a query, as was discussed previously. For example to return all maps that contain a region with an average rainfall greater than 20 inches, we could express the query as:

Query 5.3.6. *SELECT Geom FROM MapTable M1 WHERE EXISTS (SELECT * FROM GetLabelTable(M1.Geom) WHERE (integer:Avg_Rain) > 20)*

Thus, the subquery is executed for each map geometry stored in 'MapTable'. Using this concept of correlated attribute queries, we can respectively express MQL Query 5.2.4 and MQL Query 5.2.5 as:

Query 5.3.7. *SELECT Geom FROM MapTable M1 WHERE EXISTS (SELECT * FROM GetLabelTable(M1.Geom) WHERE (string:Crop) = 'Wheat')*

Query 5.3.8. *SELECT (integer:Avg_Temp) FROM MapTable M1 WHERE EXISTS (SELECT * FROM GetLabelTable(M1.Geom) WHERE (integer:Avg_Rain) > 20)*

Note that because the label tables of maps stored in the same column can have different label structures, it is possible that some label tables will not contain a column named "Avg_Rain" with type integer. Thus, if any error occurs due to schema elements not existing, the subquery simply returns no values and the query continues executing.

5.3.3.3 Component queries

Component queries are similar in concept to component attribute queries, but deal with the region geometries contained in a map instead of the region attributes. Because the map geometry is stored in the column type *map2D*, we cannot use a correlated

subquery concept similar to the approach used in component attribute queries. Instead, we must provide access to the regions in a map in a manner which corresponds to SQL and relational constructs. In MQL we were able to express component queries using the *Expand* operator. We can express this class of queries in SQL by defining the *Expand* function in terms of relations. Therefore, we define the *Expand* function to take a map geometry and return a relation consisting of singleton maps such that each singleton map represents a region in the original map. Furthermore, each singleton map will have a label table associated with it containing a single region label. For example, if a user wishes to return all regions from a map that have an area greater than one thousand square miles (as in MQL Query 5.2.6), they could use the *Expand* operator in conjunction with an *area* operator defined for regions as follows:

Query 5.3.9. *SELECT M1.Geom FROM Expand(SELECT Geom from MapTable) as M1 WHERE area(M1.Geom) > 1000*

We can also use the Expand operator in correlated subqueries to identify maps that contain regions that satisfy some predicate. For example, to return all maps that contain a region with an area greater than one thousand square miles (as in MQL Query 5.2.7) , a user could write:

Query 5.3.10. *SELECT M1.Geom FROM MapTable M1 WHERE EXISTS (SELECT M2.Geom FROM Expand(M1.Geom) as M2 WHERE area(M2.Geom) > 1000)*

CHAPTER 6
THE DISCRETE MODEL OF SPATIAL PARTITIONS

The abstract model of spatial partitions maps each point in the plane to a specific label. However, computers provide only a finite resolution for the representation of data which is not adequate for the explicit representation of abstract spatial partitions. In order to represent maps in computers, a *discrete map model* is required that preserves the properties of spatial partitions while providing a finite representation that is suitable for storage and manipulation in computers. In this section, we provide a *graph model of spatial partitions*, which is a graph theoretic, discrete model of spatial partitions. Note that there is some ambiguity among graph terms in the literature, especially concerning terms indicating graphs that are allowed to contain loops and multiple edges between pairs of vertices. We begin this section by first providing an overview of graph terms and definitions that we use to develop our model.

6.1 Definitions from Graph Theory

In graph theory, a graph is a pair $G = (V, E)$ of disjoint sets such that V is a set of vertices and $E \subseteq V \times V$ is a set of vertex pairs indicating edges between vertices. We denote the sets of vertices and edges for a given graph g as $V(g)$ and $E(g)$, respectively. A *multigraph* is a pair $G_M = (V, E)$ of disjoint sets with a mapping $E \to V \times V$ allowing a multigraph to have multiple edges between a given pair of vertices. A *loop* in a graph is an edge that has a single vertex as both of its endpoints. A multigraph with loops is a *pseudograph*, and is defined as a pair $G_P = (V, E)$ with a mapping $E \to V \times V$. Finally, a *nodeless pseudograph* is a pseudograph that (possibly) contains edges that form loops that connect no vertices and that intersect no other edges or vertices. We define a nodeless pseudograph as a triple $G_N = (V, E, N)$ (where N is the set of nodeless edges) with mapping $E \to V \times V$.

A *path* is a non-empty graph $P = (V, E)$ such that $V = \{v_1, \ldots, v_n\}$, $E = \{v_1v_2, \ldots, v_{n-1}v_n\}$ and all v_i are distinct. Given a graph $g = (V, E)$, a path of g is a

graph $p = (V_p, E_p)$ where $V_p \subseteq V(g) \ \wedge \ E_p \subseteq E(g)$ where p satisfies the definition of a path. A *cycle* is a path whose first and last vertices are identical, defined by a non-empty graph $C = (V, E)$ such that $V = \{v_1, \ldots, v_n\}$, $E = \{v_1v_2, \ldots, v_nv_1\}$ where all v_i are distinct and $n \geq 3$. Given a graph $g = (V, E)$, a cycle of g is a graph $c = (V_c, E_c)$ where $V_c \subseteq V(g) \ \wedge \ E_c \subseteq E(G)$ where c satisfies the definition of a cycle. The *length* of a cycle is the number of its edges. A *polygonal arc* is the union of finitely many straight line segments embedded into the plane and is homeomorphic to the closed unit interval $[0, 1]$. We use the terms *arc* and polygonal arc interchangeably.

A graph is *planar* if it can be embedded in the plane such that no two edges intersect. A particular drawing of a graph is an *embedding* of the graph. A particular planar graph can have multiple embeddings in the plane such that edges in each embedding are drawn differently. A particular embedding of a planar graph is a *plane graph*. Formally, a plane graph is a pair (V, E) such that $V \subseteq \mathbb{R}^2$, every edge is an arc between two vertices, the interior of an edge contains no vertex, and no two edges intersect except at their vertices. A plane graph g may contain cycles. We say a cycle c in plane graph g is *minimal* if there does not exist a path in g that splits the polygon induced by c into two pieces. We use the notation $C(g)$ to indicate the set of minimal cycles in the graph g.

6.2 Representing Spatial Partitions as Graphs

In this section, we define the type of *spatial partition graphs* that is able to model both the structural and labelling properties of spatial partitions in a graph. We first attempt to define a graph based on the vertex and edge structure of a partition, but show that this is not sufficient because the labels of the partition are not explicitly represented in such a graph. We then define a new type of graph that is capable of representing both the structural and labelling properties of a spatial partition and that is defined based on discrete concepts. We then show how such a graph can be obtained from a given spatial partition.

Recall that in the abstract model of spatial partitions, we can identify a graph structure based on the boundary of a partition π. Specifically, we observe that we can identify $\nu(\pi)$, the set of points that are vertices, and $\epsilon(\pi)$, the set of point sets belonging to edges of a partition. However, we cannot simply assign the vertices and edges to a pair (V, E) in order to achieve a graph representation of π since it is possible for nodeless edges to be present in $\epsilon(\pi)$. For example, the boundary of the rightmost face of region A in Figure 3-1 is composed of two nodeless edges; these edges are nodeless because the label of every point they contain is a set containing two attributes (i.e., no point in the edge satisfies the definition of a vertex in a partition). Therefore, we must use a triple (V, E, N) consisting of the sets of vertices, edges, and nodeless edges to represent a partition as a graph. Deriving the set V from a partition π is trivial because we can directly identify the set of vertex points $\nu(\pi)$. Definition 3 defines the set of all edges of π as $\epsilon(\pi)$; thus, it does not differentiate between edges and nodeless edges. It follows that the set of nodeless edges of a partition, N, is a subset of $\epsilon(\pi)$, but we cannot identify N by simply examining labels. Intuitively, the label of a nodeless edge should not form a subset of the label of any vertex. However, in Figure 3-1, the label of the boundary of the upper border of the left face of region A and the labels of both borders of the right face of region A are the same. Therefore, we cannot necessarily differentiate nodeless edges from edges by comparing labels. We can circumvent this problem if we can differentiate identically labeled edges from different faces of the same region. This can be achieved through the *refine* operation. Each nodeless edge in π can be identified as an edge in $\sigma = \mathit{refine}(\pi)$ whose label does not form a subset of any vertex label (a vertex lies on an edge in the refinement of a partition iff the edge label is a subset of the vertex label). Note that the refine operation does not alter the edge structure of a partition, only its labels. Therefore, we can use $\sigma = \mathit{refine}(\pi)$ to identify the set of nodeless edges N in π by saying that each edge in σ whose label does not form a subset of any vertex label in σ is in the set N. The edges of π can then be calculated as $E = \epsilon(\pi) - N$. Figure 6-1A shows the refinement of the partition in

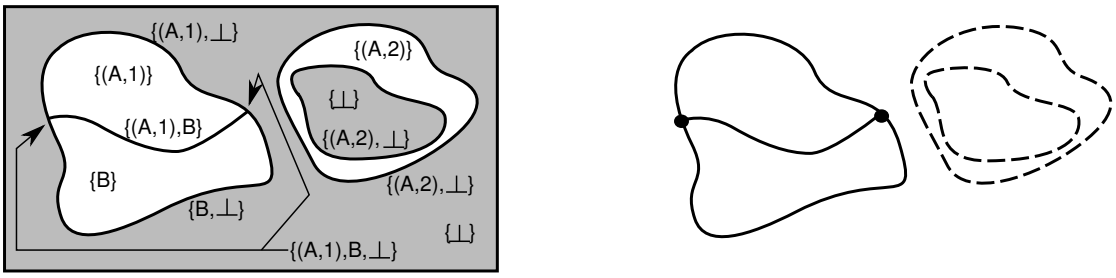


Figure 6-1. A spatial partition and its corresponding SSPG. A) The refinement of the partition in Figure 3-1A. B) The SSPG corresponding to Figure **A**. Nodeless edges are dashed.

Figure 3-1. Figure 6-1B shows the graph representation of the partition in Figure 6-1A obtained by the method described above (nodeless edges are dashed). Because deriving a graph from a partition in this fashion results in a graph that exactly represents the edge structure of the partition from which it is derived, we call this type of graph a *structural spatial partition graph* (SSPG), and define it formally as follows:

Definition 10. *Given a spatial partition π of type A and its refinement $\sigma = refine(\pi)$, we construct a structural spatial partition graph $SSPG = (V, E, N)$ with:*

$$\begin{aligned} V &= \nu(\pi) \\ E &= \epsilon(\sigma) - N \\ N &= \{n \in \epsilon(\sigma) | \nexists v \in \nu(\sigma) : \sigma(n) \subseteq \sigma(v)\} \end{aligned}$$

The SSPG is able to represent the structural aspects of a spatial partition, but it does not maintain the labelling information of the partition. Because a SSPG is defined based on a given partition, the spatial mapping for the partition is known. Therefore, the label for any edge or vertex in a SSPG can be determined through the associated spatial mapping, but this is insufficient for our purposes as we require an explicit representation of labels. However, the SSPG has the property that it is a *plane nodeless pseudograph* (PNP):

Theorem 2. *Given a partition π of type A, its corresponding SSPG is a plane nodeless pseudograph.*

Proof. The definition of a nodeless graph states that a nodeless graph may contain nodeless edges; therefore, an SSPG is nodeless by definition. Similarly, the definition of a pseudograph states that the graph may contain multiple edges between the same vertices and loops. An SSPG is therefore a pseudograph by definition because it does not exclude such features. Now we must prove that a SSPG is a plane graph. The edges of an SSPG are taken directly from a spatial partition, which is embedded in $\mathbb{R}^2$, indicating that the SSPG is an embedded graph. By the definition of spatial partitions, an edge in a partition is a border defined by a one-dimensional point set consisting of points mapped to a single label. Furthermore, this label is derived from the regions which the border separates. Assume that there exists two borders in a spatial partition that cross, which implies that the SSPG for this partition will contain two edges that intersect. In order for this to occur, these edges must separate regions that overlap, which violates the definition of spatial partitions. Therefore, a spatial partition cannot contain two borders that intersect, except at endpoints. Because the edges of an SSPG are taken directly from a spatial partition, then no two edges of an SSPG can intersect. Thus, the SSPG is a plane nodeless pseudograph. □

Although an SSPG can be easily obtained from a spatial partition, using a SSPG to model spatial partitions is inadequate for two reasons. First, spatial partitions depend on the concept of labels, so the graph representation of a partition must include a label representation. The SSPG does not implicitly model the labels of regions, edges, or vertices; rather, it depends on the existence of a spatial mapping. Second, the edges in the SSPG are taken directly from a spatial partition, which is defined on the concept of infinite point sets that we cannot directly represent discretely. Despite these drawbacks, the SSPG does have the nice property that because a SSPG is defined based on a given spatial partition, we know that any given SSPG is *valid* in the sense that it represents a valid spatial partition. Therefore, we proceed in two phases: we first define a type of graph that is capable of discretely representing the structural properties, and explicitly

representing the labeling properties, of spatial partitions. We then show how we can derive graphs of this type from spatial partitions. This allows us to define a valid graph representation of any given spatial partition.

It follows from Theorem 2 that an embedded graph that models a spatial partition such that its edges and vertices correspond to the partition's edges and vertices, respectively, must be a PNP. However, a PNP does not model the labeling of spatial partitions. In order to model labels in a graph, we must associate labels with some feature in a graph. In spatial partitions, the labels of boundaries can be derived from the labels of the regions they represent. Thus, it is possible to derive all edge and vertex labels in a partition from the region labels. Therefore, we choose to associate labels in a graph with features that are analogous to regions in spatial partitions. We are tempted to associate labels with minimum cycles in a graph representing a spatial partition; however, this is not able to accurately model situations in which a region in the spatial partition contains another region such that the boundaries of the regions are disjoint (e.g., the region labeled A_2 and the hole it contains in Figure 6-1A). Instead, we associate labels with *minimum polycycles* (MPCs) in graphs representing partitions. A minimum polycycle in a PNP G is a set of minimum cycles consisting of a minimum cycle c_o (the outer cycle), and all other minimum cycles in G that lie within c_o, and not within any other minimum cycle. Note that a minimum cycle of a plane graph induces a region in the plane defined by a Jordan curve. Therefore, we can differentiate between the interior, boundary, and exterior of such a region. We denote the region induced in the plane by the minimum cycle C of plane graph G as $R(C)$. We now formally define minimum polycycles:

Definition 11. *Given a plane nodeless pseudograph G, a* minimum polycycle *of G is a graph $MPC = (V, E, N)$ where $V \subseteq V(G)$, $E \subseteq E(G)$, $N \subseteq N(G)$, and $C(MPC) \subseteq C(G)$, containing an* outer cycle $c_o \in C(MPC)$ *and zero or more* inner cycles $c_1, \ldots, c_n \in C(MPC)$ *such that:*

$$
\begin{aligned}
&(i) \quad \forall v \in V : (\exists d \in C(MPC) | v \in V(d)) \\
&(ii) \quad \forall e \in E : (\exists d \in C(MPC) | e \in E(d)) \\
&(iii) \quad \forall n \in N : (\exists d \in C(MPC) | n \in N(d)) \\
&(iv) \quad \forall c_i \neq c_o \in C(MPC) : \partial R(c_i) \subseteq (\partial R(c_o) \cup R(c_o)^\circ) \\
&(v) \quad \nexists c_j, c_k \in C(MPC) | c_j \neq c_o \ \wedge \ c_k \neq c_o \ \wedge \ c_j \neq c_k \\
&\qquad \wedge \partial R(c_j) \subseteq (\partial R(c_k) \cup R(c_k)^\circ) \\
&(vi) \quad \nexists d \in C(G) | (\partial R(d) \subseteq (\partial R(c_o) \cup R(c_o)^\circ)) \ \wedge \ \neg (d \in C(MPC)) \\
&\qquad \wedge (\forall c_i \neq c_o \in C(MPC) : \neg (\partial R(d) \subseteq (\partial R(c_i) \cup R(c_i)^\circ))
\end{aligned}
$$

Thus, a MPC induces a region in the plane that may contain holes defined by the minimum cycles that lie in the interior of the outer cycle. Furthermore, by associating labels with MPCs in a graph representing a spatial partition, we do not need to explicitly represent the labels of edges and vertices in the graph since they can be derived by simply finding all MPCs that an edge or vertex participates in. We denote the set of all MPCs for a PNP G as $MC(G)$.

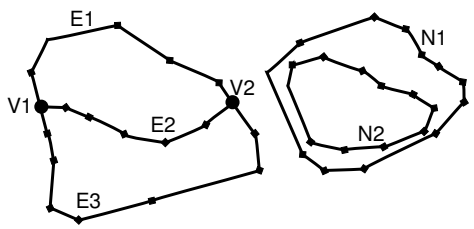
The second problem with SSPGs is that the edges are defined as infinite point sets. We require a discrete representation of edges. Therefore, in addition to assigning labels to MPCs, we define our new type of graph such that its edges are arcs consisting of a finite number of straight line segments. We define the *labeled plane nodeless pseudograph* (LPNP) as a PNP with labeled MPCs, denoted *faces*, and edges modeled as arcs as follows:

Definition 12. *Given an alphabet of labels Σ_L, a* labeled plane nodeless pseudograph *is defined by the four-tuple $LPNP = (V, E, N, F)$ consisting of a set of vertices, a set of arcs forming edges between vertices, a set of arc loops forming nodeless edges, and a set of faces, with:*

$$
\begin{aligned}
V &\subseteq \mathbb{R}^2 \\
E &\subseteq V \times V \times (\mathbb{R}^2)^n \textit{ where } n \textit{ is finite and each edge is an arc (arcs} \\
&\quad \textit{with endpoints in } V \textit{ and segment endpoints in } \mathbb{R}^2\textit{)} \\
N &\subseteq (\mathbb{R}^2)^n \textit{ where } n \textit{ is finite (nodeless arc loops)} \\
F &\subseteq \{(l \in \Sigma_L, m \in MC((V, E, N)))\}
\end{aligned}
$$

Recall that in the definition of spatial partitions, an unbounded face is explicitly represented with an empty label corresponding to the exterior of the partition. We do not explicitly model this unbounded face in the LPNP for two reasons: (i) we cannot guarantee that the edges incident to the unbounded face of a LPNP will form a connected graph, and (ii) if the edges incident to the unbounded face do form a connected graph, we cannot guarantee that it will be a cycle. However, we can determine if an edge in a LPNP is incident to the unbounded face if it participates in only a single MPC. This follows from the fact that each edge separates two regions in a partition. Because all bounded faces are modeled as MPCs in a LPNP, an edge that participates in only a single face must separate a MPC and the unbounded face.

The LPNP allows us to discretely model a labeled graph structure, but we have not yet discussed how we can obtain a LPNP for a given spatial partition. Note that because the edges of a LPNP are defined as arcs, they cannot directly represent edges from a partition. Instead, each edge in a LPNP is an approximation of an edge in its corresponding spatial partition. Therefore, we define the approximation function α that takes an edge from a partition and returns an arc which approximates that edge. Given a spatial partition π of type A and an edge approximation function α, we derive the corresponding LPNP in a similar manner as we derived the SSPG from a partition. The set of vertices in the LPNP is equivalent to $\nu(\pi)$. In order to calculate the nodeless edges, we consider the refinement $\sigma = \textit{refine}(\pi)$. Nodeless edges are then identified as all edges in σ whose label is not a subset of any vertex label. The set of edges is then difference of $\epsilon(\pi)$ and the set of nodeless edges. Finally, each labeled MPC consists of the set of



$$
\begin{aligned}
V = &\ \{\ V1, V2\ \} \\
E = &\ \{\ E1, E2, E3\ \} \\
N = &\ \{\ N1, N2\ \} \\
F = &\ \{\ (A, (\{V1, V2\}, \{E1, E2\}, \varnothing)), \\
&\quad (A, (\varnothing, \varnothing, \{N1, N2\})), \\
&\quad (B, \{V1, V2\}, \{E2, E3\}, \varnothing)), \\
&\quad (\perp, (\varnothing, \varnothing, \{N2\}))\ \}
\end{aligned}
$$

Figure 6-2. A labeled plane nodeless pseudograph for the partition in Figure 3-1. The edges and vertices are marked so that the sets of vertices, edges, nodeless edges, and faces can be expressed more easily.

approximations of the edges surrounding each region in σ along with the label of the corresponding face in π. Given an edge e, we use the notation $V_e(e)$ to indicate the set of vertices that e connects. Figure 6-2 depicts the LPNP for the partition shown in Figure 3-1.

Definition 13. *Given a spatial partition π of type A, edge approximation function α, and $\sigma = refine(\pi)$, we derive a $LPNP = (V, E, N, F)$ from π as follows:*

$$
\begin{aligned}
V &= \nu(\pi) \\
E &= \{\alpha(e \in \epsilon(\sigma) | \neg(\alpha(e) \in N))\} \\
N &= \{\alpha(n \in \epsilon(\sigma) | \nexists v \in \nu(\sigma) : \sigma(n) \subseteq \sigma(v))\} \\
F &:= \forall r \in \rho(\sigma) | r \subseteq s \in \rho(\pi) : (l, (V_m, E_m, N_m)) \in F \ \textit{where} : \\
&\qquad l = \quad \pi^{-1}(r) \\
&\qquad E_m = \quad \{\alpha(e) | e \in \omega(\sigma) \ \wedge \ e \subseteq \partial r \ \wedge \ \alpha(e) \in E\} \\
&\qquad N_m = \quad \{\alpha(n) | n \in \omega(\sigma) \ \wedge \ n \subseteq \partial r \ \wedge \ \alpha(n) \in N\} \\
&\qquad V_m = \quad \textstyle\bigcup_{e \in E_m} V_e(e)
\end{aligned}
$$

Note that it is possible to approximate edges in a partition in multiple ways. Thus, a single spatial partition may have multiple LPNPs that represent it.

In the definition of LPNPs, no restrictions are placed on the labels of MPCs. Therefore, it is possible for an labeled graph to fit the definition of an LPNP, but be labeled in such a way that violates the definition of spatial partitions. In other words, a LPNP may be labeled such that no spatial partition exists from which the LPNP can

be derived. A simple example of this is a LPNP containing two MPCs that have the same label and share an edge. Because edges only separate regions with different labels in a partition, this LPNP can not be derived from any valid spatial partition. Thus, the set of all valid LPNPs is larger than the set of LPNPs that can be derived from some spatial partition. We define a LPNP that can be derived some spatial partition as a *spatial partition graph* (SPG).

Definition 14. *A* spatial partition graph *G is a* labeled plane nodeless pseudograph *such that there exists some valid partition from which G can be derived.*

6.3 Properties of Spatial Partition Graphs

In the previous section, we defined the type of spatial partition graphs and showed how a SPG can be derived from a given spatial partition. However, we currently define a SPG as being valid only if it can be derived from a valid spatial partition. Given a labeled graph in the absence of a spatial partition, we currently cannot determine if the graph is a SPG. In this section, we discuss the properties of SPGs such that we can determine if a SPG is valid by examining its structure and labels.

Given a LPNP in the absence of a source partition from which it can be derived, we must ensure that structural and labelling properties of the LPNP are consistent with the properties of spatial partitions. To achieve this, we examine the properties of spatial partitions, defined by Definition 2 and Definition 3, and show how these properties are expressed in SPGs derived from spatial partitions. We then define the properties of SPGs and show that any LPNP that satisfies these properties is a SPG.

Recall that a SSPG is a PNP. It follows from Theorem 2 that any graph that models the edges of a partition as graph edges and vertices of a partition as graph vertices must be a PNP. Therefore, a graph cannot be a SPG if it is not a PNP. This is already expressed indirectly by the definition of SPGs as LPNPs.

Definition 2 formally defines constraints on spatial mappings that specify the type of partitions. These constraints indicate that (i) the regions in a partition are regular open

point sets, and (ii) the borders separate uniquely labeled regions and carry the labels of all adjacent regions. From these properties of partitions, we can derive properties of SPGs. By (i), we infer that all edges and vertices in a SPG must be part of a MPC. If an edge is not part of a MPC, then the edge does not separate two regions. Instead, it extends either into the interior of a MPC or into the unbounded face of the SPG, forming a cut in the polygon induced by the MPC or the unbounded face. If a vertex exists that is not part of a MPC, then it is either a lone vertex with no edges emanating from it, or it is part of a sequence of edges that are not part of a MPC. In the first case, the vertex either exists within the region induced by a MPC or the unbounded face of the LPNP, forming a puncture. In the second case, the vertex exists in a sequence of edges that is not part of a MPC, which we have already determined to be invalid.

By the second property of spatial partitions (ii), we infer that edges must separate uniquely labeled MPCs. Therefore, there cannot be an edge in an LPNP that participates in two MPCs with the same label. Fut hermore, every region in a partition must be labeled. It follows that every MPC in a SPG must be labeled. Because the unbounded face is not explicitly labeled in a SPG, one special case exists: a MPC forming a hole in a SPG (i.e., labeled with $\perp$) cannot share an edge with the unbounded face, as this would result in an edge separating two regions with the same label.

Definition 3 further identifies properties of spatial partitions. According to this definition, edges in spatial partitions always have two labels, and vertices always have three or more labels. Recall that in LPNPs, the unbounded face of the graph is not explicitly labeled. Therefore, in a SPG, all edges must participate in either one or two MPCs. Edges incident to the unbounded face of the graph will participate in only one MPC. The requirement that vertices have three or more labels in a spatial partition indicates that at a vertex, at least three regions meet. It follows that in a SPG, each vertex has at least a degree of three. Furthermore, because the unbounded face is not explicitly labeled, vertices must have at least two labels in a SPG (i.e., a vertex must

participate in at least two MPCs). We summarize the properties of SPGs and show that any LPNP that satisfies these properties is a SPG:

Definition 15. *An SPG G has the following properties:*

(i)	G *is a plane nodeless pseudograph*	*(Theorem 2)*
(ii)	$\forall e \in E(G) \cup N(G), \exists (l, X) \in F(G) \vert e \in E(X) \cup N(X)$	*(Definition 2(i))*
(iii)	$\forall v \in V(G), \exists (l, X) \in F(G) \vert v \in V(X)$	*(Definition 2(i))*
(iv)	$\forall v \in V(G) : degree(v) \geq 3$	*(Definition 3)*
(v)	$\forall e \in E(G) \cup N(G) :$	
	$1 \leq \vert\{(l, X) \in F(G) \vert e \in E(X) \cup N(X)\}\vert \leq 2$	*(Definition 3)*
(vi)	$\forall (l_1, X_1), (l_2, X_2) \in F(G) \vert l_1 = l_2 :$	
	$(\nexists e_1 \in E(X_1) \cup N(X_1), e_2 \in E(X_2) \cup N(X_2) \vert e_1 = e_2)$	*(Definition 2(ii))*
(vii)	$\forall m \in MC(G), \exists f = (l, X) \in F(G) \vert m = X$	*(Definition 2(ii))*
$(viii)$	$\forall e \in E(G) \cup N(G) \vert$	
	$\vert\{(l, X) \in F(G) \vert e \in E(X) \cup N(X)\}\vert = 1 : l \neq \{\perp\}$	*(Definition 2(ii))*

Theorem 3. *Any LPNP that satisfies the properties in Definition 15 is a SPG.*

Proof. The properties listed in Definition 15 indicate how the properties of spatial partitions are expressed in SPGs. From Theorem 2, we know that a valid SPG must be a PNP. Definition 2(i) states that all regions in a partition must be regular open sets. Because the faces in a SPG are analogous to regions in a spatial partition, this means that all edges and vertices must belong to some face; otherwise, they form a puncture or cut in some face of the SPG. Definition 15(ii) and Definition 15(iii) express this requirement. Definition 2(ii) states that borders in a partition between regions carry the labels of both regions. This implies that an edge in a spatial partition separates regions with different labels, and that every region in a spatial partition has a label. Definition 15(vi) and Definition 15(vii) express this by stating that if an edge participates in two faces of a SPG, those faces have different labels, and that every MPC in a SPG is a labeled face of the SPG. Because the unbounded face is not labeled, we must explicitly state that no edge

that participates in a cycle forming a hole can have only a single label, as this implies that an edge is separating two regions with the $\perp$ label (Definition 15(viii)). Definition 3 states that edges in a partition carry two region labels, and that vertices carry three or more region labels. Because the unbounded face is not labeled in a SPG, we cannot directly impose these properties on a SPG. Instead, we observe that the number of region labels on a vertex in a partition indicates a minimum number of regions that meet at that vertex. Therefore, we can express this property in SPG terms by stating that vertices in a SPG must have degree of at least three (Definition 15(iv)). The edge constraint from Definition 3 can be specified in terms of a SPG by the property that an edge must participate in exactly one or two faces, indicating that the edge will have exactly one or two labels (Definition 15(v)). Therefore, all properties of partitions are expressed in terms of SPGs in Definition 15, and any LPNP that satisfies these properties is a SPG. □

We now have the ability to either derive a valid SPG from a spatial partition, or verify that a SPG is valid in the absence of a spatial partition from which it can be derived. Finally, we show that given a valid SPG, we can directly construct a valid spatial partition that exactly models the SPG's spatial structure and labels. Recall that a spatial partition is defined by a spatial mapping that maps points to labels and satisfies certain properties. Therefore, to construct a partition from a SPG, we must be able to derive a spatial mapping from a SPG. We can construct such a mapping based on the labeled MCPs of a SPG. Each MCP of a SPG induces a polygon in the plane that is associated with a label. Each of these polygons is a spatial region, and is defined by its boundary, which separates the interior of the polygon from its exterior. Therefore, we can identify the interior, boundary, and exterior of such a polygon. We use the notation $R(X)$ to denote the polygon induced in the plane by MCP X. A point that falls into the interior a polygon can therefore be mapped directly to that polygon's label. The labels of points belonging to edges in the SPG are slightly more difficult to handle. Each point belonging to an edge is mapped to the labels of each face in which the edge participates. If an edge

happens to be incident to the unbounded face (it is an edge participating in a single minimum cycle), it also is mapped to the $\perp$ label. Similarly, each vertex is mapped to the labels of each cycle in which it participates. If a vertex is incident to the unbounded face, it is also mapped to the $\perp$ label. A vertex is incident to the unbounded face if and only if it is the endpoint of an edge that is incident to the unbounded face. In order to define this mapping, we first provide a notation to distinguish between edges and vertices that are incident to the unbounded face, and those that are not. We then show how to derive a spatial partition from a SPG:

Definition 16. *Given a SPG G, we distinguish two sets of edges: the set containing edges incident to the unbounded face, denoted $E_\perp$, and the set containing edges not incident to the unbounded face, denoted E_b. Likewise, we distinguish two sets of vertices: the set containing vertices incident to the unbounded face, denoted $V_\perp$, and the set containing vertices not incident to the unbounded face, denoted V_b. These sets are defined as follows:*

$$
\begin{aligned}
E_\perp(G) &= \{e \in E(G) \cup N(G)| \ |\{(l, X) \in F(G) | e \in E(X) \cup N(X)\}| = 1\} \\
E_b(G) &= (E(G) \cup N(G)) - E_\perp(G) \\
V_\perp(G) &= \{v \in V(G) | (\exists e \in E_\perp | v \in V_e(e))\} \\
V_b(G) &= V(G) - V_\perp(G)
\end{aligned}
$$

Definition 17. *Given a SPG G, we can directly construct a spatial partition π of type A as follows:*

$$A = \{l | (l, X) \in F(G)\} \cup \{\perp\}$$

$$
\pi(p) = \begin{cases}
\{l | (l, X) \in F(G) \ \wedge \ p \in R(X)^\circ\} & \text{if} \quad \exists (l, X) \in F(G) | p \in R(X)^\circ \quad (1) \\
\{\perp\} & \text{if} \quad \nexists (l, X) \in F(G) | \\
 & \qquad p \in R(X)^\circ \cup \partial R(X) \quad (2) \\
\{l | (l, X) \in F(G) \ \wedge \ p \in \partial R(X)\} & \text{if} \quad (\exists (l, X) \in F(G) | p \in \partial R(X)) \\
 & \qquad \wedge \ (\nexists e \in E_\perp(G) | p \in e) \quad (3) \\
\{l | (l, X) \in F(G) \ \wedge \ p \in \partial R(X)\} \cup \{\perp\} & \text{if} \quad (\exists (l, X) \in F(G) | p \in \partial R(X)) \\
 & \qquad \wedge \ (\exists e \in E_\perp(G) | p \in e) \quad (4)
\end{cases}
$$

In Definition 17, the type of a spatial partition derived from a SPG consists of the set of labels of faces in the SPG. The mapping is then defined by finding the labels of all faces a point participates in. If a point p lies in the interior of a face (1), then the label of that point will be the label of the face. If p does not lie in the interior or boundary of any face (2), it maps to the label $\perp$. If p lies on a boundary that is not incident to the unbounded face (3), it is mapped to the set of labels of all faces that include p in its boundary. Note that because face boundaries include vertex points, this case handles the mapping of both edge and vertex points. Similarly, if p lies on a face boundary that is incident to the unbounded face (4), then p is mapped to the set of labels from all faces which include p, as well as the label $\perp$.

CHAPTER 7
THE IMPLEMENTATION MODEL OF SPATIAL PARTITIONS

At this point, we have an abstract model of Map Algebra that precisely defines the semantics of maps and their operations and predicates. Furthermore, we have shown how to represent maps in a discrete manner such that they maintain the properties of the abstract model. In this chapter, we provide an implementation model of Map Algebra consisting of a data model for maps and algorithms for computing map operations. Furthermore, we show how operations over maps can be combined to form new map operations. We then describe a prototype implementation of Map Algebra and perform a performance comparison of the Map Algebra implementation with an existing GIS.

7.1 Map2D: an Implementation Model of Map Algebra

The implementation model of map Algebra takes into account the abstract and discrete models of spatial partitions, as well as the user view of maps in databases presented in Chapter 5. Because the goal of this thesis is to incorporate maps as first-class citizens in databases, our implementation model is designed to implement Map Algebra in a database. Therefore, the data model and algorithms assume the existence of a database that can support a type called *Map2D* either natively or through some extension mechanism.

7.1.1 A Data Model For Representing Spatial Partitions

The data model for the implementation model of Map Algebra follows the ideas presented in Chapter 5 for a data model of maps in databases. Recall that this model is centered around the concept of storing map labels in a traditional database table so that traditional SQL queries can be posed over them, while the map geometry is stored in a database column type. We follow the same approach in our implementation model of Map Algebra. In order to explain the concepts of the implementation model more clearly, we will refer the map shown in Figure 7-1 throughout the following discussion.

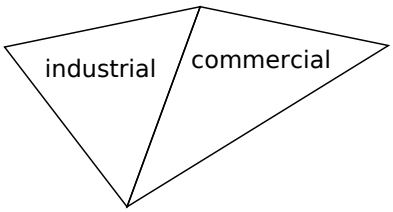


Figure 7-1. A map showing an industrial and commercial zone.

The label portion of a map is implemented nearly identically to the explanation given in Chapter 5. The labels of a map are stored as tuples in a database table. We impose the restriction that a table containing map labels must contain a column named *region_id* of type integer that will be used to associate a label with a region contained in a map geometry. The advantage of this approach is that the label table can be used in standard SQL queries without any SQL extensions. Also, labels can be manipulated with standard SQL, and no special purpose label operations must be introduced.

The geometry portion of a map is more complex than the label portion. Our goal is to incorporate the geometry portion of a map directly as a column type in a database. Therefore, we must implement the geometry portion as a single object that can be incorporated into a DBMS through extension mechanisms or through extending the DBMS code itself. There are several considerations that must be addressed in the design of the geometry: (i) the map geometry must support map operations that will be performed over it; and (ii) the concept of regions must be maintained in the geometry and map regions must be able to be identified, associated with labels in a label table, and reconstructed.

In order to address the first concern, we make the observation that nearly all map operations can be expressed in terms of the *intersect* and *relabel* operations presented in the abstract model of Map Algebra [65, 83, 87]. Although the relabel operation can have geometric consequences in a map, it is primarily an operation over labels. The intersect operation, however, is primarily a geometric operation that has consequences regarding the labels of a map. Because geometric operations tend to be more computationally intensive

than label operations, we focus the design of the map geometry on the requirements of the intersect operation. In Chapter 2, we showed that geometric operations are computed optimally by using a plane sweep algorithm. Thus, a plane sweep algorithm is an ideal choice for implementing the intersect operation, and we will design our data model based on the requirements for a plane sweep algorithm.

The plane sweep algorithm requires a specific input structure for geometric objects; specifically, geometric objects are required to be represented as a sequence of *halfsegments* ordered in *halfsegment order*. For *regions*, this is achieved as follows: we define the type $halfsegment_r = \{(s, d) | s \in segment, d \in bool\}$ where *segment* is the type incorporating all straight line segments bounded by two *points* p and q, and a *point* is a coordinate (x, y). A halfsegment is a hybrid between a point and a segment since it has features of both geometric structures. For a halfsegment $h = (s, d)$, if d is true (false), the smaller (greater) endpoint of s is the *dominating point* of h, and h is called a *left* (*right*) *halfsegment*. Hence, each segment s is mapped to two halfsegments $(s, true)$ and $(s, false)$. Furthermore, halfsegments are typically annotated with an *interior-above* flag, which indicates whether the interior of the region lies above or below the halfsegment. In addition to the use of halfsegments, the representation of a region object requires an order relation on halfsegments. Let dp be a function which yields the dominating point of a halfsegment. For two distinct halfsegments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$ with a common endpoint p, let α be the enclosed angle such that $0° < \alpha \leq 180°$. Let a predicate $rot(h_1, h_2)$ be true if, and only if, h_1 can be rotated around p through α to overlap h_2 in a counterclockwise direction. We define a complete order on halfsegments as:

$$
\begin{aligned}
& h_1 < h_2 \Leftrightarrow \\
& dp(h_1) < dp(h_2) \vee && (1) \\
& (dp(h_1) = dp(h_2) \ \wedge \ (\ \ (\neg d_1 \ \wedge \ d_2) \vee && (2a) \\
& \qquad\qquad\qquad\qquad (d_1 = d_2 \ \wedge \ rot(h_1, h_2)) \vee && (2b) \\
& \qquad\qquad\qquad\qquad (d_1 = d_2 \ \wedge \ collinear(s_1, s_2) \ \wedge \ len(s_1) < len(s_2)))) && (3)
\end{aligned}
$$

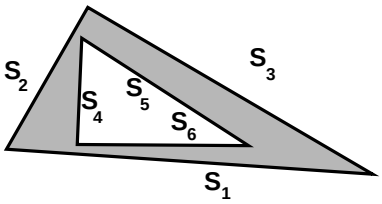


Figure 7-2. A complex region object with each segment labeled.

Since a segment can be substituted by two halfsegments, a region object can be implemented as an ordered sequence (array) of halfsegments. As an example, Figure 7-2 shows a complex region object (with a single face containing a hole) whose segments are labeled s_i. Let $h_i^l = (s_i, true)$ and $h_i^r = (s_i, false)$ denote the left and right halfsegments of a segment s_i respectively. The ordered sequence of halfsegments for this complex region is $\langle h_1^l, h_2^l, h_6^l, h_4^l, h_4^r, h_5^l, h_2^r, h_3^l, h_5^r, h_6^r, h_3^r, h_1^r \rangle$.

Although this definition of halfsegments is sufficient for computing geometric operations, maps must take into account the labels of regions in addition to geometric information. Therefore, we modify the halfsegment definition to include region labels instead of an interior-above flag. Therefore, each halfsegment will contain two region labels represented as integers, the label for the region that lies above the halfsegment, and the label for the region that lies below the halfsegment. Thus, we have $halfsegment = \{(s, a, b, d) | s \in segment, a \in \mathbb{Z}, b \in \mathbb{Z}, d \in bool\}$. We assume that the region labels correspond to regions in a label table identified by a *region_id* column, or to some specified value indicating the exterior of a map.

Therefore, we represent a map geometry as a sequence of ordered halfsegments, each labeled with the region that lies above the halfsegment, and the region that lies below the halfsegment. Note that the definition of spatial partition graphs defines a SPG by its edges and nodeless edges, which in turn are defined as connected sequences of straight line segments. The halfsegments in the implementation model of spatial partitions represent these straight line segments in a SPG. Therefore, the geometry of a map represented as a SPG and a map represented in the implementation model of spatial partitions in the

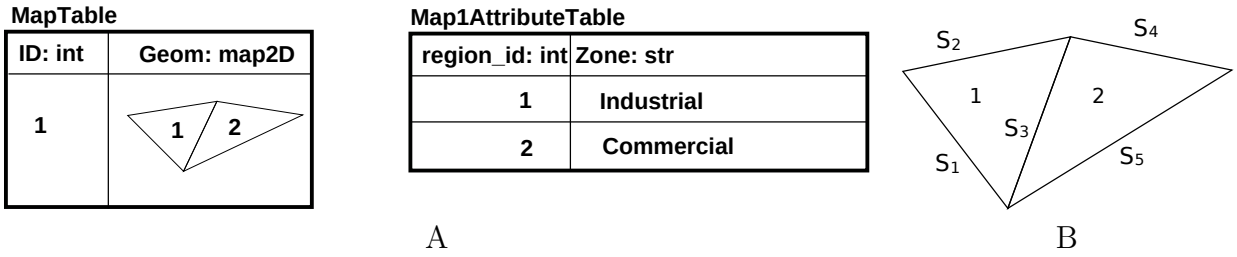


Figure 7-3. Two different views of a map. A) The map represented in the implementation model of Map Algebra. B) The map with its segments labeled.

same embedding space are identical. Furthermore, it is possible, based on region labels, to reconstruct an SPG given a map represented in the implementation data model of Map Algebra. This means that the properties of spatial partitions defined over SPGs can be checked and enforced at the implementation level.

As an example of the implementation data model of Map Algebra, consider the map shown in Figure 7-1. If we have a database that is aware of the column type *map2D* that implements the geometry portion of a map, as we have shown, then the map in Figure 7-1 could be stored in the database as shown in Figure 7-3A. If we label the segments of the map as shown in Figure 7-3B and we assume a region label of 0 for the exterior, then the halfsegments in halfsegment order would be: $(h_1^l, 1, 0)$, $(h_2^l, 0, 1)$, $(h_1^r, 1, 0)$, $(h_3^l, 1, 2)$, $(h_5^l, 2, 0)$, $(h_2^r, 0, 1)$, $(h_3^r, 1, 2)$, $(h_4^l, 0, 2)$, $(h_4^r, 0, 2)$, $(h_5^r, 2, 0)$ where $h_i^l = (S_i, j, k, true)$ and $h_i^r = (S_i, j, k, false)$ for segment S_i.

7.1.2 Algorithms for Map Operations

In Chapter 2, we saw that research efforts have provided algorithms to compute spatial operations over map geometries. In this section, we provide algorithms for the three fundamental map operations of *intersect*, *relabel*, and *refine* based on the implementation data model of Map Algebra. We then show how these operations can be combined to form new operations. Note that in the literature, the intersect operation is frequently referred to as *overlay*.

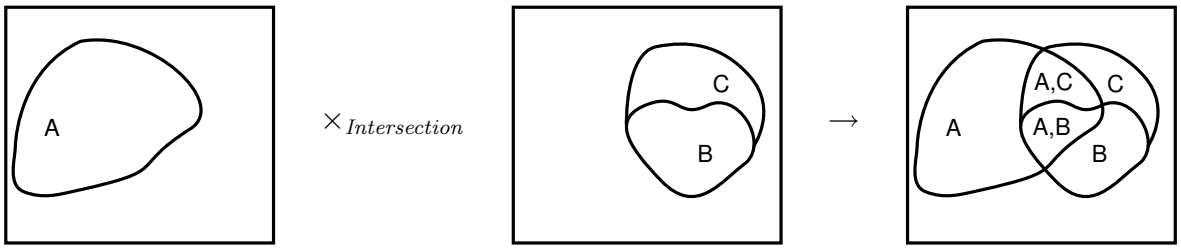


Figure 7-4. The result of the *intersect* operation applied to two maps.

7.1.2.1 Intersect

The intersect operation and its variants have received significant attention in database, computational geometry, and GIS literature. Our implementation follows the concepts presented in these papers, but is tailored to our particular data model. Recall that the intersect operation, as defined in the abstract model, takes two maps and returns a third such that each point in the third map carries the label assigned to each identical point in the argument maps. Figure 7-4 depicts two sample maps and the result of the intersect operation applied to them.

In essence, our intersection algorithm is identical to a traditional plane sweep algorithm in that it traverses the argument maps in halfsegment order and computes the intersections of line segments, and the resulting regions in the result map. The main difference in our algorithm to traditional plane sweep algorithms is the handling of region labels. Recall that regions in the result of an intersection algorithm will carry labels from both argument maps. Therefore, we must determine the labels of all regions in the result map in addition to computing the geometric intersection of the argument maps. We achieve this in two steps. First, as the plane sweep algorithm progresses, we mark each halfsegment with the labels of all regions whose interiors directly border the halfsegment, or contain the halfsegment. We call this *label aggregation*. For example, Figure 7-5 shows two maps represented in the implementation model of Map Algebra, complete with region

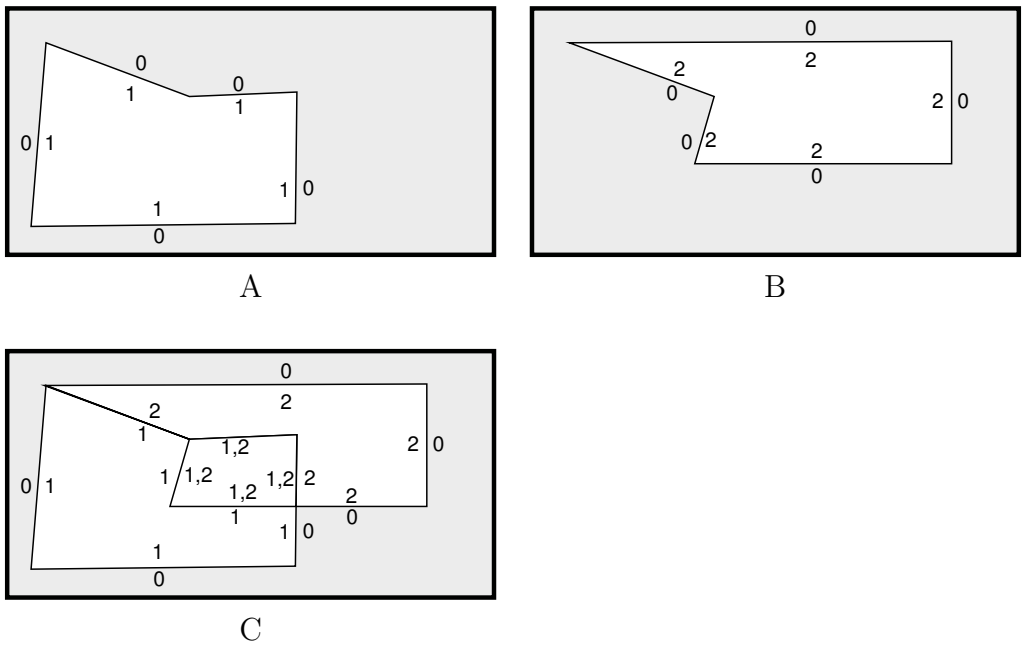


Figure 7-5. Aggregating map labels in an intersect operation. A) The first argument map. B) The second argument map. C) The the result of aggregating the argument maps' labels during an intersect operation.

labels on the segments. The third map shows the aggregated labels as a result of label aggregation during an intersection. Note that segments that lie in the interior of a region from one of the original maps will have the label of that region on both sides of the halfsegment.

Aggregated labels provide the information necessary to identify the labels from the original regions that should be applied to a region in the result of an intersection of two maps. However, the halfsegment definition used by the implementation model of Map Algebra only allows two integers to be used as region labels. Therefore, in addition to label aggregation, we use a *label mapping* which maps aggregated labels to a new label value that uniquely identifies the region. It is straightforward to implement such a mapping using an AVL tree with the search key consisting of the pair of labels from the original maps. Therefore, each region in the result map that consists of an overlapping portion of regions from both argument maps will have an aggregate label, and that aggregate label will be mapped to a new region label in the result. As the plane sweep

progresses during an intersect algorithm, the result map is annotated with the result of the label mapping, not the aggregated labels.

The final problem that must be addressed is how to determine the aggregate labels. As the plane sweep algorithm progresses, it maintains a sorted list of halfsegments called the *status structure* that contains all halfsegments that intersect the sweep line. These halfsegments are sorted in the order according to where they intersect the sweep line. If we assume the sweep line is vertical, then the halfsegments are sorted with respect to the y coordinate of the point where they intersect the sweep line. We assume that each halfsegment in the status structure is marked with a flag indicating which of the original maps it belongs to. Given this information, we are able to compute the aggregate labels for a halfsegment based on the halfsegment below it in the status structure. For example, assume that one argument map to the intersect operation is called A and the other B. When a halfsegment h from map A is added to the status structure, we look at the halfsegment immediately below it in the status structure, halfsegment j. Note that we look at the halfsegment below the current one due to the halfsegment ordering defined previously (all halfsegments that are less than the current halfsegment will already be present in the status structure, or will have been processed). If h and j are from different maps, then we look at the label from j for the region that lies above it. If that region is the exterior, then it is clear that h does not intersect a region from the opposing map (again, this is due to the halfsegment ordering). If the above label for j is a region label, then h lies in the interior of the region bounded by halfsegment j. Therefore, h is marked with the label of the region above j for both its below and above labels. If halfsegment h and j are from the same map, then we copy any aggregated labels from j to h. This is because both h and j lie in the same region from the opposing map, and thus, should be aggregated with that region's label.

A special case of label aggregation occurs when two halfsegment h and j, respectively from each argument map, overlap along a line. In this case, the above label from each

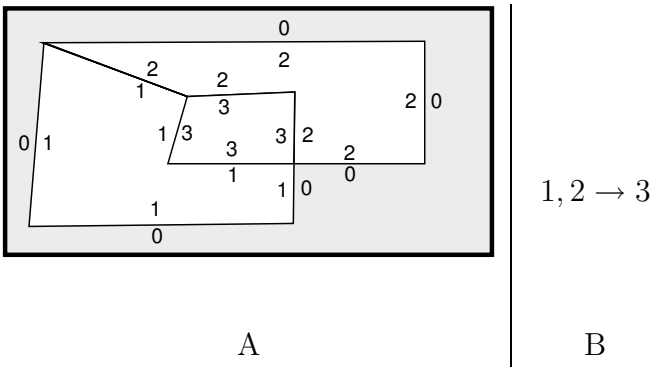


Figure 7-6. The result of the plane sweep portion of the intersect algorithm for the maps shown in Figure 7-5. A) The labeled geometry. B) The mapping.

halfsegment is combined, and the below label from each halfsegment is combined to form the aggregate label. Because a boundary is present from regions from both maps, then neither h or j lie in the interior of another region, so no label will be aggregated to *both* the above and below labels of either halfsegment.

The result of the plane sweep portion of the intersect algorithm will be a new map in the implementation model of Map Algebra and a label mapping that maps pairs of labels from the original maps, to labels in the new map. Thus, given the maps in Figure 7-5A and Figure 7-5B, the result at this point is shown in Figure 7-6.

Finally, the label table for the new map must be constructed based on the information from the label tables of the original maps. The new label table contains all columns from the label table of the original maps, except it will contain only a single *region_id* column. Furthermore, each column name will have the integer 1 or 2 appended if it came from the label table in the first or second argument map to the intersect operation, respectively. This ensures that no naming conflicts occur.

The last step of the intersect algorithm is to populate the new label table. First, all labels for regions in the first argument map that have portions that did not intersect any region from the second map are added to the label table. These regions can be determined during the plane sweep and kept in a list. Second, the same procedure is done for regions in the second argument map that contain a portion that does not overlap any region

Map1AttributeTable

region_id: int	description: str
1	region of interest

A

Map2AttributeTable

region_id: int	description: str
2	high temperature area

B

Map3AttributeTable

region_id: int	description1: str	description2: str
1	region of interest	
2		high temperature area
3	region of interest	high temperature area

C

Figure 7-7. The resulting label table from an intersect operation. A) The label table for the map shown in Figure 7-5A. B) The label table for the map shown in Figure 7-5B. C) The label table computed as the result of an intersect operation.

from the first argument map. Note that for these labels, the values in columns from the opposing map will be given default or empty values. Then, the mapping from aggregate labels to new region labels is used to construct the labels for overlapping regions from the argument maps. Thus, if the label tables for the maps shown in Figure 7-5A and Figure 7-5B are assumed to be those shown in Figure 7-7A and Figure 7-7B, the the label table of their intersection would be the table shown in Figure Figure 7-7C. The algorithms for these procedures are given in Algorithm 2 and Algorithm 1.

It is known that a sweep line algorithm can run optimally in $O(n \lg n + k)$ time for input maps with a total of n segments and k intersections. Furthermore, our algorithm utilizes a mapping of region labels implemented as an AVL tree. Therefore, we must take into account the time complexity of adding region labels. In the worst case, every region from one map can intersect every region from the opposing map. Therefore, for two maps respectively containing r and s regions, the complexity of computing the region label mapping is $O(rs \lg rs)$. However, the number of intersecting regions in two maps tends to be low, especially compared to the number of segments in the maps. Furthermore, geometric configurations, in practice, typically do not approach such extreme

Algorithm 1: The plane sweep portion of the *intersect* algorithm.

Input: Two maps M and P as defined by the implementation model of Map Algebra. An empty mapping A of labels from the original maps to labels in the result map. Empty lists L_M and L_N to hold labels from the original maps that exist in the result map.

Output: Map Q

```
1  Initialize sweep line structures.  while not end of sweep do
2      if h is a left halfsegment then
3          Advance sweep line to h. h is the left-most halfsegment yet to be processed;
4          Find j, the halfsegment below h in the status structure;
5          if h and j are from different maps then
6              Aggregate the above and below labels for h based on j;
7              Update the mapping A if any overlapping regions are detected;
8          end
9          Update the list L_M if no overlapping regions are detected for a label from
           M;
10         Update the list L_N if no overlapping regions are detected for a label from N;
11     else
12         If overlapping regions exists, update h's label based on mapping A;
13         Add h and its corresponding left halfsegment to Q;
14     end
15 end
```

cases of many regions overlapping many other regions. We will provide running times of an implementation of this algorithm over actual spatial data in later sections that support these claims. Therefore, the running time of this algorithm in the worst case is $O(n \lg n + k + rs \lg rs)$, with the $n \lg n + k$ terms typically dominating the running time.

7.1.2.2 Relabel

The relabel operation, as defined by the abstract model for maps, takes a map and a relabeling function which is then used to alter the labels of a map. A relabeling may have geometric consequences. For instance, relabeling a region to the exterior label effectively removes a region from a map. Another example is relabeling a region such that its new label is identical to an adjacent region will merge the two regions, causing any boundaries that exist between them to disappear.

Algorithm 2: The algorithm to compute the label table for intersecting two maps.

Input: The label tables for two maps M_L and P_L. A mapping A of labels from the original maps to labels in the result map. Lists L_M and L_N of labels from the original maps that exist in the result map.

Output: Label table Q_L

```
1  Create the label table Q_L;
2  for each region label in L_M do
3  |  Compute a tuple for Q_L consisting of label attributes from L_M and default
   |  values for attributes from L_N;
4  |  Insert the new tuple in Q_L;
5  end
6  for each region label in L_N do
7  |  Compute a tuple for Q_L consisting of label attributes from L_N and default
   |  values for attributes from L_M;
8  |  Insert the new tuple in Q_L;
9  end
10 for each entry in mapping A do
11 |  The mapping entry consists of region labels r_M, r_N, and r_Q for region labels in
   |  M, N, and Q, respectively;
12 |  Compute a tuple for Q_L consisting of label attributes from L_M and L_N with
   |  region_ids equal to r_M and r_N, respectively;
13 |  Insert the new tuple with region_id r_Q in Q_L;
14 end
```

From an implementation perspective, defining a method for a user to create a relabeling function that can then be passed to an operation is non-trivial. Therefore, we approach the problem of relabeling from a different perspective. Our approach is to allow a user to modify a label table for a map, and then run a general relabeling operation that determines any geometric implications of the modifications to the label table, and updates the map geometry accordingly. This approach does not require a method of passing a relabeling function to the relabel operation, while not sacrificing any generality.

As was briefly mentioned in the examples above, the relabeling operation can have two geometric consequence: (i) labels may be removed from a label table, which implies that the region with that label has been deleted; and (ii) a label in the label table may be altered such that it is identical to another label in the label table, meaning that two regions have been merged into a single region (requiring any shared boundaries between

the regions to be removed). Thus, our relabel operation must be able to detect and enforce these changes in the map geometry.

The relabel operation consists of two parts, a duplicate label identification part, and a geometric consistency enforcement part. The duplicate label identification part identifies labels in the label table that are identical, but that have different region_id values. This is achieved by computing a Cartesian product of a label table with itself, and then disregarding all tuples that are not identical except for their region_ids. The result is a relation containing pairs of identical labels that belong to different regions. This relation is traversed, and a mapping is constructed such that the region_id value of duplicate labels are mapped to the same region_id value. Furthermore, a list is constructed of all region_id values that exist in the label table.

Once the mapping of duplicate labels and the list of region_ids in the label table are constructed, they are used to determine changes to the map geometry. This is achieved by traversing the halfsegment sequence of the map geometry. For each halfsegment, the region label for the region above the halfsegment is checked against the label mapping and label list. If the region label is not in the label list, then the corresponding label has been removed from the label table, and the region should no longer exist in the map. Therefore, the region label for the halfsegment is changed to the exterior label. If the region label is in the label list, then the mapping is checked. If the label exists in the mapping, then it is changed to the label to which it is mapped. The same procedure is performed for the label of the region below the current halfsegment. Finally, once the above and below labels of the halfsegment have been processed, a final check is performed to ensure that the above and below labels are not identical. If these labels are identical, then the halfsegment borders two regions with identical labels, and thus, is removed. Algorithm 3 summarizes the relabel algorithm.

Given a map with r regions and n halfsegments, it takes $O(r^2)$ time to compute the Cartesian product, since each region has a single label. The mapping of region labels

Algorithm 3: The relabel algorithm.

Input: A map M and its corresponding label table L_M.
Output: Map M with the geometric changes implied by its label table L_M applied.

```
1  Compute the Cartesian product L_M × L_M;
2  Keep only the tuples in L_M × L_M that have different region_ids and identical labels;
3  Create mapping A that maps region_ids for equivalent labels to the same region_id;
4  Create list V of region_ids in L_M;
5  for each halfsegment h in M do
6  |  Get the above label a of h;
7  |  Get the below label b of h;
8  |  if a exists in mapping A then
9  |  |  Replace a with the value to which it maps;
10 |  end
11 |  if a does not exists in list V then
12 |  |  Replace a with the exterior label;
13 |  end
14 |  if b exists in mapping A then
15 |  |  Replace b with the value to which it maps;
16 |  end
17 |  if b does not exists in list V then
18 |  |  Replace b with the exterior label;
19 |  end
20 |  if a = b then
21 |  |  Remove h from M;
22 |  end
23 end
```

can be implemented using an AVL tree, which results in a complexity of $O(r \lg r)$. An AVL tree can also be used to implement the list of region_ids, which then facilitates searching the list in $O(r \lg r)$. Finally, the halfsegment list must be traversed once, which takes linear time $O(n)$. Therefore, the total time complexity for this algorithm is $O(n + r^2 + r \lg r)$. Because the number of regions in map tends to be small compared to the number of halfsegments, this algorithm performs well in practice. Furthermore, it is possible to create specialized relabel algorithms for specific types of relabeling that are much faster, but this algorithm is general in the sense that it can correct a map geometry with respect to any modifications a user makes to a label table. This turns out to be

very valuable when using relabel in conjunction with other existing operations in order to create new operations.

7.1.2.3 Refine

The refine operation, as was defined previously, relabels region faces such that every region in a map has only a single face. In other words, any regions that consist of multiple faces in a map will be altered such that each face is a single region in the map after a refine operation has been performed. In the implementation model of Map Algebra, this is achieved by altering the label table of a map such that two faces of a region will have unique labels, and every face of a region in a map will have a unique region_id. We compute this in two steps. First, the map geometry is modified so that each region face is identified by a unique region_id. Second, the label table is modified so that every region_id in the map geometry refers to a unique label in the label table.

The first step of the refine algorithm requires a technique known as *cycle walking*. In essence, given the left most halfsegment of a cycle (i.e., a region face in a map), it is possible to find each successive halfsegment that makes up the cycle of halfsegments efficiently. First, we present the algorithm to walk the cycles in a *region*. We then extend the method to work on cycles in a map. We then discuss the second part of the refine operation

Walking Cycles in Regions.

Because this this section deals with regions, we will assume a halfsegment structure as defined for regions; i.e., a halfsegment contains a segment, a boolean flag indicating if it is a left or right halfsegment, and a boolean flag indicating if the interior of the region lies above or below the halfsegment. Furthermore, we develop this algorithm to determine the *component view* of a region. In essence, the algorithm determines which cycles in a region are outer cycles, which are hole cycles, and to which outer cycle each hole belongs. It also determines if a region is valid. Although all of these properties are not required for the cycle walk portion of the refine algorithm, they are none the less useful, so we include

them. We assume that the input to our algorithm is a sequence of ordered halfsegments. If the input represents a region, the algorithm returns the region with its cyclic structure information; otherwise, the algorithm exits with an error message indicating the input sequence does not form a semantically correct region. We begin by providing a high level overview of the algorithm and then present the algorithm and provide a discussion of its details.

In general terms, the algorithm must identify all cycles present in the halfsegment sequence, and classify each cycle as either an outer cycle or a hole cycle of a particular face. To accomplish this, each halfsegment is *visited* once by the algorithm. Note that due to the definition of the type *region*, each segment belongs to exactly one cycle. When a halfsegment is visited, the algorithm marks the halfsegment indicating to which face and cycle it belongs, and whether that cycle is an outer cycle or a hole cycle. The algorithm does not alter the input when marking halfsegments, rather a parallel array to the input sequence is used to represent the cycle information. The algorithm visits halfsegments by stepping through the input list sequentially.

The first halfsegment in the input sequence will always be part of the outer cycle of a face, due to the definition of complex regions and the halfsegment ordering defined previously. Therefore, it can be visited and marked correctly. Once a halfsegment has been visited, it is possible to visit and correctly mark all other halfsegments in the cycle that it belongs to in a procedure which we denote as the *cycle walk*. Thus, all halfsegments that form the cycle to which the first halfsegment in the input sequence belongs are then visited. The algorithm then begins stepping through the remaining halfsegments. The next unvisited halfsegment encountered will be part of a new cycle. The algorithm then visits this new halfsegment. The algorithm can deduce whether this halfsegment is an outer cycle of a new face or a hole in an existing face by examining where the halfsegment lies in relation to already known cycles. To determine this, we use a plane sweep algorithm to step through the halfsegments. Thus, we can take advantage of the plane sweep status

structure to find whether or not the current halfsegment lies in the interior of a previously visited face. Once the new halfsegment is visited, we perform a cycle walk from it. Then, the algorithm continues stepping through the input list until it reaches another unvisited halfsegment, visits it, and repeats this procedure. The algorithm is shown in Algorithm 4.

To properly describe the algorithm outlined in Algorithm 4, we introduce several notations. The function $info(h)$ for a given halfsegment h returns its cyclic information, that is, its *owning* cycle, and if part of a hole, its owning face. A cycle *owns* a halfsegment if the halfsegment is part of the boundary of the cycle, and a face owns a hole if the hole is inside the face. We define the function $NewCycle(h)$ to annotate h with a unique identifier for a new cycle. Let f be a halfsegment belonging to an outer cycle of a face. The function $Owns(h, f)$ annotates the halfsegment h to indicate that it belongs to a hole in the face that owns f. Finally, we employ the function $Visit(p)$ to mark a point p as having been visited. The function $Visited(p)$ is used to verify is point p was marked as visited already. Points are only marked as visited when a halfsegment with dominating point p has been visited during the cycle walk. We mark points as being visited in order to identify the special case of a hole cycle that meets the outer cycle of a face at a point. The function $Visited(h)$ is used to verify if halfsegment h has been visited already. A halfsegment has been visited if it has been annotated with face/hole information. For a halfsegment h, we can directly compute its corresponding right (left) halfsegment h_b, which we call its *brother* by switching its boolean flag indicating which end point is dominant. We define the next halfsegment in the cycle to which h belongs as h_+ such that the dominating endpoint of h_b is equal to the dominating point $dp(h_+)$ and $h_+ \neq h_b$ and h_+ is the first halfsegment encountered when rotating h_b clockwise (in an outer cycle) or counter-clockwise (in a hole cycle) around its dominating point. The previous halfsegment in the cycle is similarly defined as h_-.

Classifying Outer and Hole Cycles: By using a sweep line, the algorithm steps through the halfsegment sequence to find the smallest unannotated halfsegment h,

Algorithm 4: The algorithm for deriving the component view of a region.

```
   Input: Sequence of unannotated halfsegments H
   Output: Sequence H with fully annotated halfsegments
 1 while not end of sweep do
 2 |   Advance sweep line to h. h is the left-most halfsegment yet to be annotated;
 3 |   Using sweep line status, determine h as part of an outer cycle or a hole cycle;
 4 |   NewCycle(h); Visit(dp(h));
 5 |   if h belongs to a hole then
 6 |   |   Using sweep line status, retrieve halfsegment f from its owning outer cycle;
 7 |   |   Owns(h, f);
 8 |   |   Set cycle walk mode to use counter-clockwise adjacency;
 9 |   else
10 |   |   Set cycle walk mode to use clockwise adjacency;
11 |   end
   |   /* Begin walking the cycle */
12 |   c ← h+;
13 |   while c ≠ h do
14 |   |   if Visited(dp(c)) then
15 |   |   |   q ← c; c ← c−; NewCycle(c); Owns(c, h);
16 |   |   |   while dp(c) ≠ dp(q) do
   |   |   |   |   /* Trace back anchored hole */
17 |   |   |   |   info(c−) ← info(c); c ← c−;
18 |   |   |   end
19 |   |   else
20 |   |   |   info(c) ← info(h); Visit(dp(c)); c ← c+;
21 |   |   end
22 |   end
23 end
```

create a new cycle for this halfsegment, and mark its dominating point as visited (line 2-4). At this point, the algorithm needs to determine whether h belongs to a hole cycle (line 5) or an outer cycle (line 9). If a cycle is identified as a hole cycle, the outer cycle to which it belongs must also be identified (line 6-7), and the cycle must be walked using counter-clockwise adjacency of halfsegments (line 8). Recall that the plane sweep algorithm maintains the sweep line status structure, which is a ordered list of *active* segments, such that it provides a consistent view of all halfsegments that currently intersect the sweep line, up to the current *event* (the addition or removal of a halfsegment). By examining the halfsegment directly below a halfsegment h in the sweep

line status, we can determine whether h is a part of an outer cycle or a hole cycle of an existing face. In other words, if halfsegment p is directly below halfsegment h in the sweep line status structure and the interior-above flag of p is set to *true*, it follows that h is either in the interior of the cycle to which p belongs, or h is part of the cycle to which p belongs. Recall that as soon as a halfsegment is classified as being apart of a hole or face, the cycle to which it belongs is walked (Section 7.1.2.3) and all other halfsegments in that cycle are marked accordingly (lines 12-22). Therefore, if a halfsegment belongs to the same cycle as any halfsegment that has been previously encountered by the sweep line, it is already known to which face and/or hole cycle it belongs. Furthermore, all halfsegments that are less than a given halfsegment in halfsegment order have already been classified. Therefore, we can determine if an unmarked halfsegment belongs to a hole or outer cycle by examining the halfsegment immediately below it in the sweep line status structure.

From the definition of a face, the outer cycle of a face of a region always covers (encloses) all of its hole cycles. This means that the smallest halfsegment of this face is always a part of the outer cycle. This is also true for the entire region object where the smallest halfsegment in the ordered sequence is always a part of the first outer cycle of the first face. Furthermore, due to the order relation of halfsegments and the cyclic structure of a polygon, the smallest halfsegment of a face will always be a left halfsegment with the interior of the face situated above it. Thus, when we process this halfsegment, we set its interior-above flag to indicate this fact. Since we have classified this cycle as an outer cycle, we can walk the cycle and set the interior-above flag for all halfsegments of this cycle. For example, Figure 7-8A illustrates the case where the smallest halfsegment of the sequence is processed and the cycle is classified as an outer cycle.

Once the first outer cycle of a face in a region has been processed, we continue to process halfsegments that have not yet been classified based on the plane sweep status structure. Figure 7-8B shows an example. Here, we add/remove visited halfsegments into/from the sweep line status in sequence ordered up to the smallest unvisited

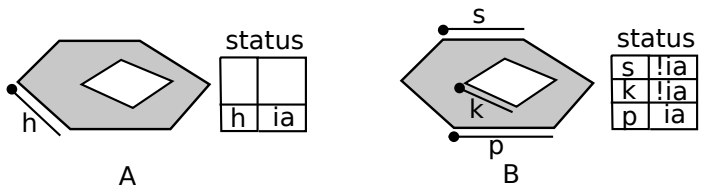


Figure 7-8. Processing halfsegments in a region. A) Processing the smallest halfsegment h of the sequence. B) Processing halfsegment k of a cycle.

halfsegment k. This halfsegment *must* be the start of a new cycle that we must now classify. We know k is the start of a new cycle because all halfsegments of an existing cycle that include a halfsegment j such that $j < k$ must have been marked as visited by the walking process. Once we reach this new cycle represented by its starting halfsegment k, we add this halfsegment into the sweep line status. We classify the type of cycle k belongs to by examining the interior-above flag of the halfsegment p (its predecessor) which was already visited and sits immediately below k in the sweep line status structure. If the predecessor indicates that the interior of the face is above it (the interior-above attribute of p is set to true), then k lies in the interior of the cycle to which p belongs; thus, k must be part of a hole cycle and the interior of the face to which k belongs must lie below k. If the interior-above flag of p indicates that the interior of the face to which p belongs is below p, then the current halfsegment k must be part of an outer cycle of a new face. In case that there is no predecessor, then the current halfsegment must be a part of an outer cycle of a new face, because it does not lie in the interior of any other face's outer cycle. Once the cycle is classified as either an outer cycle of a new face or a hole cycle of an existing face, the cycle walking procedure is carried out to determine all halfsegments that belong to the cycle.

Walking Cycles: In general terms, we use the phrase *walking a cycle* to indicate the traversal of a cycle such that each halfsegment that forms the cycle is visited. Furthermore, the halfsegments in such a traversal are visited in the order in which they appear in the cycle. In other words, given a halfsegment h, all halfsegments in the cycle to which h belongs are found by repeatedly finding h_+ until the the original halfsegment

is encountered again. For example, when walking the outer cycle of the region in Figure 7-2 in clockwise order beginning from S_1, the halfsegments would be encountered in the order $h_1^l, h_1^r, h_3^r, h_3^l, h_2^r, h_2^l$. The two main challenges to this portion of the algorithm are (i) to identify cycles correctly such that they correspond to the unique representation of a region as stated in the definition of complex regions, and (ii) to achieve this efficiently. In this section we show how to satisfy the first challenge. Time complexity is discussed in the next section.

When a halfsegment h is encountered by the algorithm that has not yet been classified, it is classified as belonging to a hole or outer cycle in line 5. If h belongs to an outer cycle, then the cycle walk portion of the algorithm in lines 12-22 is executed. Due to the halfsegment ordering and the definition of regions, the smallest unvisited halfsegment in the input sequence that the plane sweep encounters is always a left halfsegment of an outer cycle of a face and the interior of that face always lies above the halfsegment. If we rotate h_b clockwise around its dominating point, it will intersect the interior of the face. Thus, the first halfsegment encountered when rotating h_b clockwise around its dominating point will be part of the outer cycle of the same faces (except for a special case discussed below) and will be h_+. We know this to be true because if we find h_+ in this fashion and it turns out to be part of another face, then two faces would intersect, which is prohibited by the definition of complex regions. It follows that each successive halfsegment in the outer cycle can be found by rotating the brother of the current halfsegment clockwise around its dominating point because the location of the interior relative to the halfsegment can always be deduced based on the previous halfsegment encountered in the cycle walk.

One special case occurs when walking outer cycles: the existence of a hole in a face that meets the outer cycle at a point. When walking an outer cycle that contains such a hole, the halfsegments that form the hole will be classified as being part of the outer cycle using the procedure just described. In order to remedy this, we mark each point that is a dominating point of a halfsegment encountered during the cycle walk (line 20).

Each time we find a new halfsegment that is part of an outer cycle, we first check if its dominating point has been visited yet (line 14). If it has been visited, then we know that we have encountered that point before, and a hole cycle that meets the outer cycle must have been discovered. When this happens, we loop backwards over the cycle until we find the halfsegment whose dominating point has been visited twice (lines 15-18). The halfsegments forming the hole are then marked as such. The remainder of the outer cycle is then walked.

Walking a hole is identical to walking an outer cycle, except that a counter-clockwise rotation from h_b is used to find h_+. A counter-clockwise rotation is required because the interior of the face is intersected by h_b when rotating h_b around its dominating point. When walking holes, the special case exists that two holes may meet at a point. Thus, we employ the same strategy to detect this case as we did with the special case of a hole meeting a face (lines 15-18).

Walking Cycles in Maps.

Walking cycles in a map is similar to walking cycles in a region. However, there are some differences. First, because regions in a map can share halfsegments that separate them, we cannot simply mark a halfsegment as being visited. Instead, we must identify if we have visited the halfsegment when walking the cycles of the region above the halfsegment, or below the halfsegment. Therefore, two parallel arrays are required to store this information, one to mark the above side of halfsegments that have been visited, and the other to mark the below side. The second difference is that the purpose of the refine algorithm is to give every face in a map a unique label. Therefore, we must remember the region_id of each face we have visited, and change the region_id of any face that has the same region_id of a face that has already been visited to a new, unique region_id. We achieve this by maintaining a mapping of region_id values from the original map to region_id values in the refined map. At the first halfsegment encountered for each cycle, we check the mapping to see if the region_id for the new cycle is in the mapping. If not,

then it is the first time that region_id has been encountered. In this case, we add the region_id to the mapping such that it maps to itself. If the region_id is already in the mapping, then we add the old region_id to the mapping such that it maps a new region_id x; furthermore, as the cycle corresponding to the face is being walked, the region_id for every halfsegment in the cycle is changed to x. These are the only modifications required for the cycle walking algorithm presented above to be applied to the refine operation of maps.

Extending the Label Table.

The second part of the refine operation alters the label table for a map so that it is consistent with the map geometry created in the cycle walking portion of the refine algorithm. In order to ensure that region labels remain unique, the label table for the map being refined is altered so that it contains an additional column, which we will call *ref*, of type integer. This column is given a default value, such as -1, that is not used as a region_id value. Once the label table has this new column, then the tuples corresponding to the labels of the new regions (which previously were faces of regions) in the map must be inserted. This is achieved using the label mapping computed during the cycle walk. Recall that the cycle walk portion of the algorithm returns a mapping A from region_ids belonging to the map before the cycle walk portion of the algorithm has taken place, to region_ids belonging to the map after the cycle walk portion of the algorithm has taken place. Therefore, for each region_id i in A that maps to a new region_id j that is not in the label table, a tuple with region_id j must be inserted into the label table with identical attribute values as the tuple with region_id i, except that the *ref* column must contain a different value than the tuple with region_id i. Therefore, the value of j is used in the *ref* column, since it is unique. Once every entry in the mapping has been processed, the refine operation is complete. Algorithm 5 and Algorithm 6 summarize this.

From the discussion of the cycle walk portion of the algorithm, it is clear that the plane sweep and altering of halfsegment labels takes $O(n \lg n)$ time for a map with n

Algorithm 5: The cycle walk portion of the refine algorithm.

Input: A map M.
Output: M modified according to the refine operation, and a label mapping A.

```
1  Initialize sweep line structures;
2  while not end of sweep do
3      Get next halfsegment h with an unvisited side;
4      if above side of h has not yet been visited then
5          Check mapping A for above label a of h;
6          if a is in mapping A then
7              Set a to map to new region_id c in A;
8              Change a to c in h;
9              Walk the cycle as in Algorithm 4, marking the appropriate side of
               halfsegments as being visited and changing appropriate labels;
10         else
11             Set a to map to a in A;
12             Walk the cycle as in Algorithm 4, marking the appropriate side of
               halfsegments as being visited;
13         end
14     end
15     if below side of h has not yet been visited then
16         Check mapping A for below label b of h;
17         if b is in mapping A then
18             Set b to map to new region_id c in A;
19             Change b to c in h;
20             Walk the cycle as in Algorithm 4, marking the appropriate side of
               halfsegments as being visited and changing appropriate labels. Use
               counter-clockwise ordering around points.;
21         else
22             Set b to map to b in A;
23             Walk the cycle as in Algorithm 4, marking the appropriate side of
               halfsegments as being visited. Use counter-clockwise ordering around
               points.;
24         end
25     end
26 end
```

halfsegments. The addition of a mapping of region labels to new region labels can be implemented using an AVL tree, and thus has a complexity of $O(r \lg r)$ time for a map containing r faces. Finally, the portion of the algorithm that alters the label table must alter each existing tuple, plus insert a new tuple for each new region identified in the cycle walk portion of the algorithm. Thus for a map containing r faces, this takes $O(r)$ time

Algorithm 6: The label table modification portion of the refine algorithm.

Input: A label table L_M for map M and a mapping A from the cycle walk portion of the refine algorithm.

Output: Label table L_M modified according to the label mapping A.

```
1 Add a new column named ref to L_M with a default value;
2 for each entry a → b in A do
3 |   if a! = b then
4 |   |   Construct tuple t identical to the tuple with region_id a in L_M;
5 |   |   Set the region_id of t to b;
6 |   |   Set the ref column value of t to b;
7 |   end
8 end
```

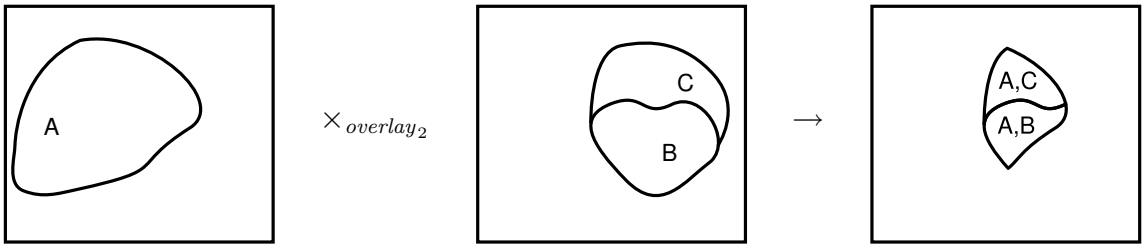


Figure 7-9. The result of the $overlay_2$ operation applied to two maps.

since each tuple is modified or inserted once. Therefore, the time complexity for the refine algorithm is $O(n \lg n + r \lg r + r)$. Since the number of halfsegments in a map is typically much greater than the number of regions, this tends to be dominated by first term.

7.1.2.4 Combining operations to form new operations

It is possible to combine the previous three operations to create new operations. For instance, a common operation for maps is to overlay two maps such that the result map only contains the intersecting portions of regions in the original maps. We denote this operations $overlay_2$ (to differentiate it from the term *overlay* which typically refers to the intersect operation described above), and show an example in Figure 7-9.

In order to compute the $overlay_2$, we simply combine the *intersect* and *relabel* operations defined previously. Given two argument maps, M and P, we first compute

the intersect operation, resulting in map Q. Recall that the label table for the map Q will have all columns from the label tables for M and P; furthermore, the column names corresponding to columns that came from the first argument map, M, will have a 1 appended, and the column names corresponding to columns that came from the second argument map P, will have 2 appended. In order to compute the $overlay_2$, we must remove all regions from Q that cover an area that is covered by a region in only a single argument map. We can identify those regions due to the fact the the attributes in the label table for all attributes from a single argument region will have a default value. Therefore, we must simply remove the tuples that satisfy this property. Once those tuples are removed, the relabel operation is executed over Q, which enforces the geometric consequences of removing tuples from the label table of Q (i.e., the corresponding regions are removed). After the relabel, Q contains the result of the $overlay_2$ operation applied to M and P.

7.2 A Prototype Implementation of Map Algebra

In order to test the implementation model of Map Algebra presented in the previous section, we have produced a prototype implementation of the data model and the intersect, relabel, and refine algorithms. Our implementation is in C++ and uses an Oracle database to store the geometric and attribute data for maps. The implementation follows the presented model very closely. We used Oracle database extension mechanisms to create a database type *map2D*, which holds the map geometry in the form of ordered halfsegments, each annotated with the ID of the region that lies above and below it. The attributes are stored in a label table. However, due to limitations in the extensibility mechanisms of Oracle, the intersect, relabel, and refine operations were not able to be implemented in the database itself. Instead, the map geometry data must be read from the database by an external library which runs the algorithms, and the writes the result back to the database. Although this is not optimal in the sense that map data must be copied to and from the database, because we were able to take advantage of database

extension mechanisms for the map geometry, we are still able to use the transaction functionality provided by the DBMS.

Recall that a hypothesis we made to motivate the use of maps as first class citizens in databases was that spatial systems would be easier to implement and less complex. Our implementation confirms this at two levels. First, we are able to directly use functionality provided by databases. By using database extension mechanisms to implement Map Algebra, we are able to utilize database transactions, persistent storage, and multi-user access. Furthermore, we are able to use standard SQL to pose queries over attribute data, as well as maps. Furthermore, a single map type is used to represent spatial data in the database and in the viewer, so we completely avoid a middleware layer, and can directly display the results of operations without any post-processing. Therefore, this hypothesis has been confirmed by our implementation. In the following sections, we look at the performance of map operations in our system in order to test the proposed hypothesis that implementing maps as first class citizens in spatial systems increases performance.

7.3 Performance Comparison of Map2D Algorithms with an Existing GIS

In order to test the effectiveness of our Map Algebra model, we have conducted a performance comparison between our *Prototype Map Algebra Implementation* (PMAI), and an existing GIS. We have chosen to compare PMAI against a commercial GIS product that has existed for many years, is considered and industry standard, and that is used worldwide. We will refer to this system as GISX. GISX has been optimized throughout its lifecycle for performance, and thus, is an ideal candidate to judge the performance of our system against.

7.3.1 Method of Comparison

The goal of our comparison is to test the relative performance of operations of GISX and PMAI. Therefore, we will take a data set, use it as input for operations in both systems, and compare the time it takes to run those operations. In order to make a fair comparison, we choose operations implemented in both systems and that have the same

semantics. Because maps are not first class citizens in GISX, it is clear that the algorithms for operations in both systems will be different, so we choose operations that should have the same result in both systems given the same input data. Therefore, we choose to test the *overlay* operation (what we call intersect in Map Algebra), and the *intersection* operation, what we call $overlay_2$.

The overlay operation is a good choice to compare the two systems because it is implemented directly in both PMAI and GISX. Furthermore, overlay forms the basis of many other operations, so it should provide insight into the relative performance of map processing in general in both systems.

The $overlay_2$ (*intersection*) operation in PMAI is built using a combination of intersect and relabel; therefore, PMAI does not have a special purpose algorithm optimized for this operation. Thus, we expect the performance of this algorithm in PMAI to be less than that of the overlay algorithm. However, a comparison with the GISX implementation should provide insight into the feasibility of using combinations of intersect, relabel, and refine to implement other operations, instead of implementing a special purpose algorithm for every operation.

We have chosen three data sets to use as input for operations, each taken from the TIGER 2007 shapefiles provided by the U.S. Census. Each data set consists of the counties of the states of Vermont, Florida, and Texas, respectively. Vermont was chosen as the first data set because it is a small state with few counties. Florida is a medium sized state with about fifty counties. Finally, Texas is a large state with about two hundred counties. Therefore, each data set reflects a different size in terms of the number of halfsegments used to represent the state, and an increasing number of counties, which form the regions in the maps. Furthermore, the TIGER data set is usefull because it provides attributes for each county.

In order to test each operation, a map is chosen from one of the three data sets. The original map is stored, along with a copy of the map that has been moved in space slightly

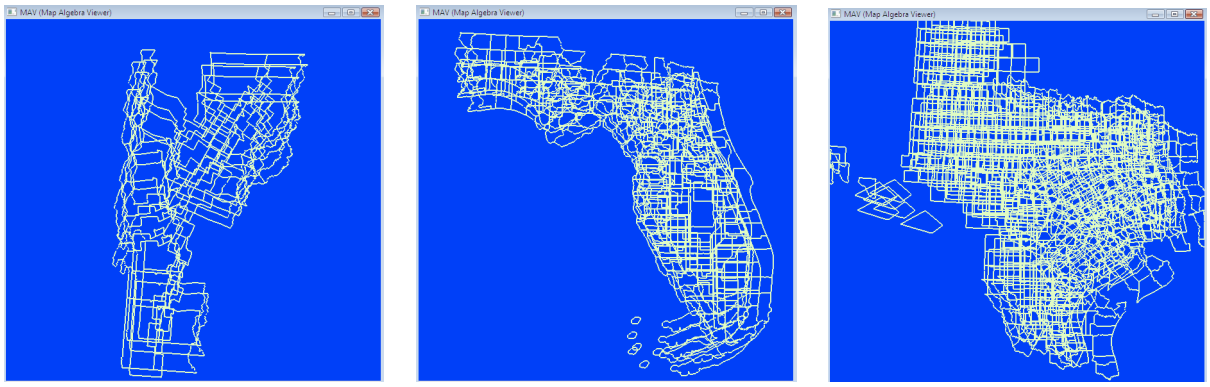


Figure 7-10. Images of the PMAI displaying the final result of the *overlay* algorithm test for each of the data sets used.

such that a significant portion of the new map intersects the old map. These two maps are then used as input to an operation. The time it takes to run the operation is recorded, and the operation is repeated two more times on the input and the average running time is kept. The result of the operation is then used as input to the operation again, with another copy of the original map that has been moved in a different direction than the first copy, but that still overlaps a significant portion of the original map. Again, the running time is averaged for three runs over these inputs, and the output is used in the next iteration. This process is repeated five times for each data set, and the running times are recorded. For reference, the final map generated for each data set after completing this process five times using the *overlay* operation is shown in Figure 7-10. These images are screenshots of our PMAI used to test Map Algebra concepts. Note that some counties are excluded from the data sets because the regions that represent them in the TIGER data files do not conform to the definition of complex regions. In other words, these regions include cycles that are not closed or that contain dangling segments.

We have chosen to perform the experiment as stated above because for each iteration that an operation is run, the number of halfsegments needed to represent the map increases, as well as the number of regions in the map. Thus, as the number of iterations increases, we are able to see how the running time of each operation scales as the argument data size and complexity increases. Furthermore, because we have chosen a

small, medium, and large data set, we are also able to determine (in the early iterations) how each operation performs on small and non-complex data. Therefore, the results of running these operations should provide a general picture of the performance of each system.

Note that because of the design differences of GISX and PMAI, simply timing the total execution from issuing the command until command completion is misleading. This is because PMAI stores maps in a database, and thus, data must be transferred across a network connection for loading and storage. GISX stores data in files, so it must simply read the data from disk. Therefore, we do not count the time it takes to load files for operations. By observing the disk activity patterns of GISX during operations, it seems that data is loaded before an operation begins executing. Thus, we feel this is a reasonable decision. Furthermore, data files are on the order of a few megabytes to tens of megabytes, which takes a negligible amount of time to load from disk compared to the running time of the operations tested. The GISX algorithm seems to write the result of operations incrementally throughout the running of an operation. Therefore, we write the result of operations in PMAI to disk and include that in the overall running time (we do not include the network transfer time for writing back to the database). We achieve a fine grained control over which portions of operations are timed in PMAI through the use of a *stopwatch* software mechanism that can be started and stopped from within the PMAI source code, and which computes time on the order of milliseconds. The GISX algorithm provides its own timing mechanism that records the total time of an operation on the order of seconds. Thus, we are able to time comparable input/output functionality between operations, and the time it takes to compute the result.

7.3.2 Results

The results of running the tests described above are summarized in Figure 7-11 and Figure 7-12. Each graph shows the average running time for an algorithm in both PMAI and GISX, and the number of halfsegments in the resulting map. In Figure 7-11,

it is clear that the PMAI algorithm runs more quickly than its counterpart in GISX in every instance. Furthermore, as the number of halfsegments increase the performance gap between the algorithms tends to increase as well.

Recall that there are two methods of computing a map overlay, the filter and refine methods that treat each map as a collection of individual regions, and the object method that treats a map as a single object. The PMAI approach uses the object method; therefore, the number of overlapping regions in two argument maps does not affect the running time of geometric portion of the algorithm. However, in the filter and refine approach, a geometric algorithm must be computed for every pair of regions that overlap. In many cases, this means that a region must be processed multiple times, once for each region whose bounding box overlaps its bounding box. Given our method of testing, it is possible that the increased number of regions in each successive overlay operation is causing the GISX algorithm's performance degradation. However, the filter and refine approach should perform significantly better than the PMAI approach when few regions overlap. This is because most regions will be identified during the filter step and not be included in the refine step. The object method, however, involves all regions in two argument maps in every case. In other words, the PMAI algorithm does not make use of any filtering based on bounding boxes. In order to test this assumption, we computed the intersect operation between the final maps of Vermont and Florida that result from the experiment described above using the *overlay* operation. The resulting map contained 840294 halfsegments 3056 regions. Furthermore, the maps of Florida and Vermont do not intersect, and are far enough apart that their bounding boxes do not intersect. However, when this experiment was run, it took an average 33.67 seconds for GISX to compute the result, and an average of 12.97 seconds for PMAI to compute the result. Thus, the PMAI approach is a good choice even in situations that favor the filter and refine method. This is due to the fact that a significant part of the running time of the overlay algorithm in

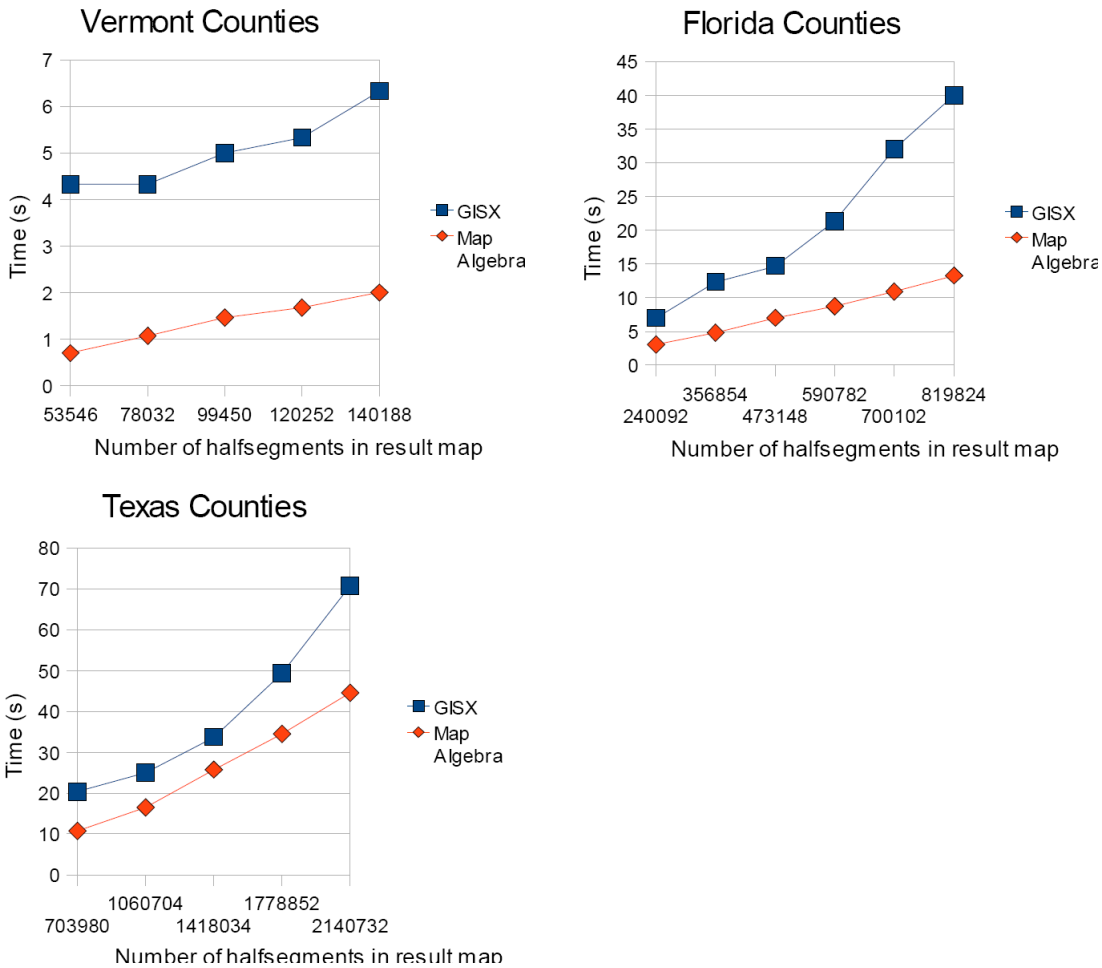


Figure 7-11. Running times for the intersect operation in PMAI and GISX.

PMAI deals with intersecting halfsegments. When no halfsegments intersect, each map is simply traversed.

In Figure 7-12, the running time for PMAI is broken down into its constituent parts. Recall the $overlay_2$ algorithm in PMAI consists first of an *overlay* algorithm, then a query to remove the labels of the non-intersecting portions of the argument maps from the result map, then a *relabel* operation. The running time for each of these components is shown. For the Vermont and Florida data sets, the PMAI algorithm clearly performs better than the GISX algorithm. However, we see that for the Texas data set, the GISX algorithm performs better in nearly every case. The overlay portion of the algorithm runs competitively, but as the complexity of the maps increases in terms of the number of

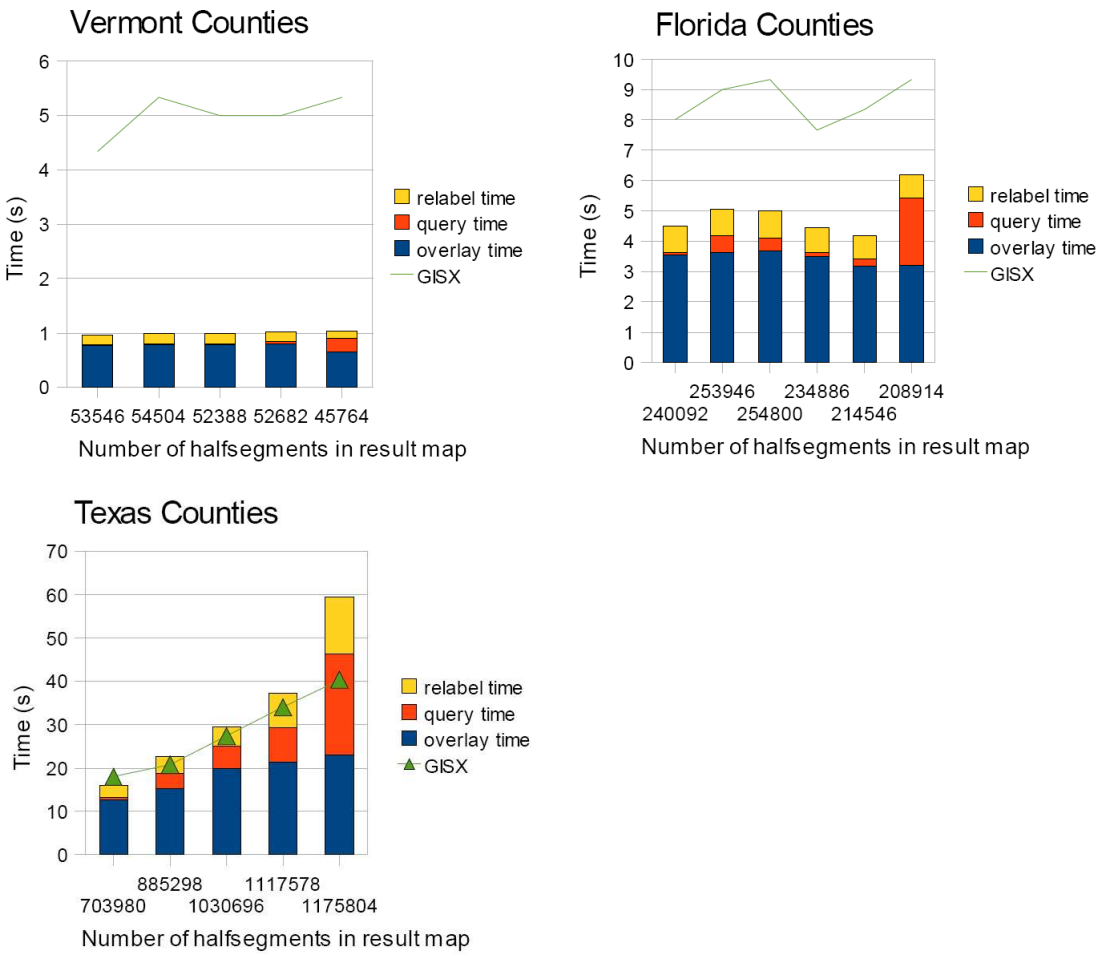


Figure 7-12. Running times for the $overlay_2$ operation in PMAI and GISX.

regions, the query and relabel portions of the algorithms begin to increase. Recall that these algorithms include a quadratic component with respect to the number of regions in the maps involved. Thus, this is reflected in the overall running time. It should be noted that the GISX algorithm is most likely a special purpose algorithm, while the PMAI algorithm is emulated by combining different algorithms. Therefore, even though the performance of the PMAI algorithm is poor in the largest data set, the PMAI algorithm performs well for small and medium sized data sets. Thus, the approach of creating operations by combining other operations is feasible in many cases.

In general, these results show that considering maps as first class citizens in spatial systems results in significant performance gains for geoprocessing. Furthermore, emulating

operations through the combination of other operations, instead of creating a special purpose algorithm for every operation, is also feasible for small and medium data sets. However, it should be noted that the PMAI implementation is rather basic in that no optimizations to reduce memory consumption are used. Thus, if more aggressive memory management techniques are used, large data sets may perform better using this approach. Finally, even in cases that favor filter and refine methods of computing spatial operations, the PMAI approach of using maps as first class citizens still performs well in operations. By incorporating bounding box refinement methods into PMAI, this performance may be able to be increased even further.

CHAPTER 8
CONCLUSION

In this thesis, we have provided a definition of Map Algebra at three levels. At the abstract level, the model of spatial partitions defines a mathematical type for maps and the precise semantics of operations over them. Furthermore, topological relationships between maps, a subject that has not been explored in the literature, were identified and defined. Using this definition, we were able to design a query language for maps in spatial databases and show how it could be implemented using an extended SQL. A discrete model of maps based on the abstract model was then developed in an effort to move Map Algebra into an implementable concept. This model, based on graph theory, provided a mechanism to enforce map type validation at the discrete level by defining the properties of maps at the discrete level. Finally, at the implementation level, we defined a data model for maps and algorithms to compute map operations. We then implemented these algorithms and performed a performance test over the implementation of Map Algebra comparing it to a mature, commercial system. The results showed that the Map Algebra implementation outperformed the commercial GIS in terms of execution times for operations in nearly every test performed. This result validates the approach and the execution of Map Algebra as a competitor to traditional map processing techniques currently in use.

In Chapter 1, we introduced three hypothesis about the effects of integrating maps as first class citizens in databases. Hypothesis 1 claimed that integrating maps as first class citizens in databases would prove to be more efficient than emulating maps in a middleware layer. In Section 7.3, we showed that our Map Algebra implementation outperformed a mature commercial GIS for operations implemented with an algorithm specifically tailored to that operation, and in most cases outperformed the GIS in operations that were implemented by combining other operations in the Map Algebra implementation.

Hypothesis 2 in Chapter 1 stated that implementing maps as first class citizens would lead to an easier implementation of map operations and map systems. We found this to be true in practice. The Map Algebra implementation consisted of a single map data type that was implemented as a software library. This single library was then integrated directly into an extensible DBMS, and a visualizer program. By integrating the library into a DBMS, we were able to directly utilize database services such as transaction control, persistent storage management, and multi-user access. Thus, none of these functionalities needed to be re-implemented in middleware. Because the visualizer program was also aware of the map type, no middleware layer was required to convert data formats for display, or even for operations. Therefore, the implementation was simplified as compared to an implementation requiring a traditional middleware layer.

Finally, hypothesis 3 stated that by integrating maps into databases, we could achieve new functionality. We found that through database extension mechanisms, it is possible to directly use maps in SQL queries in databases. Furthermore, notions such as topological relationships can be implemented and included in the database extension mechanism so that they can be used in SQL statements as well. Furthermore, we were able to identify the types of queries possible over maps in databases, and show how an extended SQL can be used to perform them. The result is that new functionality can be implemented for maps in databases. Furthermore, we defined a map query language, MQL, which was defined such that people without database experience could use the language. Thus, we have made headway into bringing map functionality in databases to the larger scientific community, and not simply database experts.

The results of this thesis show that the concepts presented in Map Algebra form a competitive method to represent, store, and manage spatial data. However, Map Algebra, as presented, is only initial work into the large area of spatial data handling in terms of maps. For example, the spatial partition data model used to define maps in Map Algebra can only represent maps containing region features. Therefore, maps containing point and

line features cannot be represented. Developing a map data model to handle such maps is a significant undertaking, as the complexity of maps increases greatly with the addition of new features. Furthermore, notions such as network processing regarding spatially embedded networks has not been investigated in terms of Map Algebra.

In addition to new directions in spatial data modeling, Map Algebra concepts can also be applied to physical data storeage mechanisms for spatial data. For example, instead of storing collections of regions in separate tuples of a database table, a map may be used to store the data. This would allow the performance advantages of map processing algorithms from Map Algebra to be utilized by more basic spatial types such as regions. Furthermore, using maps to implement spatial database joins has shown promise for being efficient, even when indexes are not available on the data.

In conclusion, by rethinking the storage and representation of maps in spatial databases, we have been able to introduce new, efficient methods of spatial data management and processing. The concepts developed in Map Algebra show promise for improving many aspects of spatial data management, even in cases where traditional spatial types are used instead of maps. Furthermore, there is much work still to be done in this area, and Map Algebra provides a platform to identify new problems and implement solutions.

REFERENCES

[1] E. G. Hoel, S. Menon, and S. Morehouse, "Building a Robust Relational Implementation of Topology," in *SSTD*, 2003, pp. 508–524.

[2] J. Herring, "TIGRIS: A Data Model for an Object-Oriented Geographic Information System," *Computers and Geosciences*, vol. 18, no. 4, pp. 443–452, 1992.

[3] M. DeMers, *Fundamentals of Geographic Information Systems*. John Wiley and Sons, 1990.

[4] P. Burrough and R. McDonnell, *Principles of Geographic Information Systems*. Oxford University Press, 1998.

[5] J. Malczewski, *GIS and Multicriteria Decision Analysis*. John Wiley and Sons, 1999.

[6] I. Heywood, S. Cornelius, and S. Carver, *Introduction to Geographical Information Systems*. Prentice Hall, 2002.

[7] C. D. Tomlin, *Geographic Information Systems and Cartographic Modelling*. Prentice-Hall, 1990.

[8] J. K. Berry, "Fundamental Operations in Computer-Assisted Map Analysis," *Int. Journal of Geographical Information Systems*, vol. 1, no. 2, pp. 119–136, 1987.

[9] A. U. Frank, "Overlay Processing in Spatial Information Systems," in *Proc. of the 8th Int. Symp. on Computer-Assisted Cartography, AUTOCARTO 8*, 1987, pp. 16–31.

[10] M. Schneider, *Spatial Data Types for Database Systems - Finite Resolution Geometry for Geographic Information Systems*. Berlin Heidelberg: Springer-Verlag, 1997, vol. LNCS 1288.

[11] R. Viana, P. Magillo, E. Puppo, and P. A. Ramos, "Multi-VMap: A Multi-Scale Model for Vector Maps," *Geoinformatica*, vol. 10, no. 3, pp. 359–394, 2006.

[12] W. C. Filho, L. H. de Figueiredo, M. Gattass, and P. C. Carvalho, "A Topological Data Structure for Hierarchical Planar Subdivisions," in *4th SIAM Conference on Geometric Design*, 1995.

[13] A. Voisard and B. David, "Mapping Conceptual Geographic Models onto DBMS Data Models," Berkeley, CA, Tech. Rep. TR-97-005, 1997.

[14] R. H. Güting, "Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems," in *Int. Conf. on Extending Database Technology (EDBT)*, 1988, pp. 506–527.

[15] R. H. Güting and M. Schneider, "Realm-Based Spatial Data Types: The ROSE Algebra," *VLDB Journal*, vol. 4, pp. 100–143, 1995.

[16] T. Bittner, "The Qualitative Structure of Built Environments," *Fundam. Inf.*, vol. 46, no. 1-2, pp. 97–128, 2001.

[17] V. Tsetsos, C. Anagnostopoulos, P. Kikiras, P. Hasiotis, and S. Hadjiefthymiades, "A Human-Centered Semantic Navigation System for Indoor Environments," *perser*, vol. 0, pp. 146–155, 2005.

[18] S. Vasudevan, S. Gächter, V. Nguyen, and R. Siegwart, "Cognitive Maps for Mobile Robots-an Object Based Approach," *Robot. Auton. Syst.*, vol. 55, no. 5, pp. 359–371, 2007.

[19] S. Thrun, "Robotic Mapping: a Survey," pp. 1–35, 2003.

[20] R. O. C. Tse and C. Gold, "TIN Meets CAD: Extending the TIN Concept in GIS," *Future Gener. Comput. Syst.*, vol. 20, no. 7, pp. 1171–1184, 2004.

[21] *Boundary Graph Operators for Nonmanifold Geometric Modeling Topology Representations.* Elsevier, 1988.

[22] M. Mantyla, *Introduction to Solid Modeling.* New York, NY, USA: W. H. Freeman & Co., 1988.

[23] B. G. Baumgart, "Winged Edge Polyhedron Representation." Stanford, CA, USA, Tech. Rep., 1972.

[24] M. Raubal and M. F. Worboys, "A Formal Model of the Process of Wayfinding in Built Environments," in *COSIT '99: Proceedings of the International Conference on Spatial Information Theory: Cognitive and Computational Foundations of Geographic Information Science.* London, UK: Springer-Verlag, 1999, pp. 381–399.

[25] U.-J. Rüetschi and S. Timpf, "Modelling Wayfinding in Public Transport: Network Space and Scene Space," in *Spatial Cognition*, ser. Lecture Notes in Computer Science, C. Freksa, M. Knauff, B. Krieg-Brückner, B. Nebel, and T. Barkowsky, Eds., vol. 3343. Springer, 2004, pp. 24–41.

[26] F. R. Broome and D. B. Meixler, "The TIGER Data Base Structure," *Cartography and Geographic Information Systems*, vol. 17, pp. 39–48, Jan 1990.

[27] J. Herring, "TIGRIS: Topologically Integrated Geographic Information Systems," in *8th International Symposium on Computer Assisted Cartography*, 1987, pp. 282–291.

[28] N. S. Chang and K. S. Fu, "A Relational Database System for Images," in *Pictorial Information Systems*, N. S. Chang and K. S. Fu, Eds. Springer, 1980, pp. 288–321.

[29] N. Roussopoulos, C. Faloutsos, and T. Sellis, "An Efficient Pictorial Database System for PSQL," *IEEE Trans. Softw. Eng.*, vol. 14, no. 5, pp. 639–650, 1988.

[30] M. J. Egenhofer, "Spatial SQL: A Query and Presentation Language," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 86–95, 1994.

[31] P. M. Aoki, "How to Avoid Building Datablades that Know the Value of Everything and the Cost of Nothing," 1999, pp. 122–133.

[32] S. Banerjee, V. Krishnamurthy, and R. Murthy, "All Your Data: The Oracle Extensibility Architecture," in *Component Database Systems*, ser. Morgan Kaufmann Series in Data Management Systems, K. R. Dittrich, Ed. Morgan Kaufmann Publisher, 2001, ch. 3, pp. 71–104.

[33] J. Davis, "IBM's DB2 Spatial Extender: Managing Geo-Spatial Information within the DBMS," Tech. Rep., 1998.

[34] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, "GENESIS: An Extensible Database Management System," vol. 14, no. 11, pp. 1711–1730, 1988.

[35] M. Carey and L. Haas, "Extensible Database Management Systems," vol. 19, no. 4, pp. 54–60, 1990.

[36] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg, "A Data Model and Query Language for EXODUS," 1988, p. 413423.

[37] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita, "Starburst Mid-Flight: As the Dust Clears," vol. 2, no. 1, pp. 143–160, 1990.

[38] L. A. Rowe and M. Stonebraker, "The POSTGRES Data Model," 1987, pp. 83–96.

[39] H. J. Schek, H.-B. Paul, M. H. Scholl, and G. Weikum, "The DASDBS Project: Objectives, Experiences, and Future Prospects," vol. 2, no. 1, pp. 25–43, 1990.

[40] M. Scholl and A. Voisard, "Thematic Map Modeling," in *SSD '90: Proceedings of the first symposium on Design and implementation of large spatial databases.* New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 167–190.

[41] M. McKenney, A. Pauly, R. Praing, and M. Schneider, "Ensuring the Semantic Correctness of Complex Regions," in *Advances in Conceptual Modeling - Foundations and Applications, ER Workshops*, 2007, pp. 409–418.

[42] A. Frank and W. Kuhn, "Cell Graphs: A Provable Correct Method for the Storage of Geometry," in *2rd Int. Symp. on Spatial Data Handling*, 1986, pp. 411–436.

[43] M. J. Egenhofer, A. Frank, and J. P. Jackson, "A Topological Data Model for Spatial Databases," in *1st Int. Symp. on the Design and Implementation of Large Spatial Databases.* Springer-Verlag, 1989, pp. 271–286.

[44] M. J. Egenhofer and J. Herring, "Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases," National Center for Geographic Information and Analysis, University of California, Santa Barbara, Technical Report, 1990.

[45] E. Clementini and P. Di Felice, "A Model for Representing Topological Relationships between Complex Geometric Features in Spatial Databases," *Information Systems*, vol. 90, pp. 121–136, 1996.

[46] M. Schneider and T. Behr, "Topological Relationships between Complex Spatial Objects," *ACM Trans. on Database Systems (TODS)*, vol. 31, no. 1, pp. 39–81, 2006.

[47] M. F. Worboys and P. Bofakos, "A Canonical Model for a Class of Areal Spatial Objects," in *3rd Int. Symp. on Advances in Spatial Databases*. Springer-Verlag, 1993, pp. 36–52.

[48] E. Clementini, P. Di Felice, and G. Califano, "Composite Regions in Topological Queries," *Information Systems*, vol. 20, pp. 579–594, 1995.

[49] M. J. Egenhofer, E. Clementini, and P. Di Felice, "Topological Relations between Regions with Holes," *Int. Journal of Geographical Information Systems*, vol. 8, pp. 128–142, 1994.

[50] R. H. Güting, "An Introduction to Spatial Database Systems," *The VLDB Journal*, vol. 3, no. 4, pp. 357–399, 1994.

[51] Z. Huang, P. Svensson, and H. Hauska, "Solving Spatial Analysis Problems with GeoSAL, A Spatial Query Language," in *Proceedings of the 6th Int. Working Conf. on Scientific and Statistical Database Management*. Institut f. Wissenschaftliches Rechnen Eidgenoessische Technische Hochschule Zürich, 1992, pp. 1–17.

[52] U. Lipeck and K. Neumann, "Modelling and Manipulating Objects in Geoscientific Databases," in *ER*, S. Spaccapietra, Ed. North-Holland, 1986, pp. 67–85.

[53] J. L. Bentley and T. A. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 643–647, 1979.

[54] B. Chazelle and H. Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane," *J. ACM*, vol. 39, no. 1, pp. 1–54, 1992.

[55] R. Güting, T. de Ridder, and M. Schneider, "Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types," in *SSD '95: Proceedings of the 4th International Symposium on Advances in Spatial Databases*. London, UK: Springer-Verlag, 1995, pp. 216–239.

[56] M. de Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. Berlin, Germany: Springer-Verlag, 2000.

[57] D. E. Muller and F. P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theor. Comput. Sci.*, vol. 7, pp. 217–236, 1978.

[58] J. Nievergelt and F. Preparata, "Plane-sweep Algorithms for Intersecting Geometric Figures," *Commun. ACM*, vol. 25, no. 10, pp. 739–747, 1982.

[59] D. A. Randell, Z. Cui, and A. Cohn, "A Spatial Logic Based on Regions and Connection," in *International Conference on Principles of Knowledge Representation and Reasoning*, 1992, pp. 165–176.

[60] M. McKenney, A. Pauly, R. Praing, and M. Schneider, "Preserving Local Topological Relationships," in *ACM Symp. on Geographic Information Systems (ACM GIS)*. ACM, 2006, pp. 123–130.

[61] ——, "Local Topological Relationships for Complex Regions," in *Int. Symp. on Spatial and Temporal Databases (SSDT)*, 2007, pp. 203–220.

[62] H. Ledoux and C. Gold, "A Voronoi-Based Map Algebra," in *Int. Symp. on Spatial Data Handling*, Jul 2006.

[63] L. D. Floriani, P. Marzano, and E. . Puppo, "Spatial Queries and Data Models," in *Information Theory: a Theoretical Basis for GIS*, I. C. A. U. Frank and U. Formentini, Eds. Springer-Verlag, Lecture Notes in Computer Science, N.716, 1992, pp. 113–138.

[64] L. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi," *ACM Trans. Graph.*, vol. 4, no. 2, pp. 74–123, 1985.

[65] M. Erwig and M. Schneider, "Partition and Conquer," in *3rd Int. Conf. on Spatial Information Theory (COSIT)*. Springer-Verlag, 1997, pp. 389–408.

[66] M. McKenney and M. Schneider, "Spatial Partition Graphs: A Graph Theoretic Model of Maps," in *Int. Symp. on Spatial and Temporal Databases (SSDT)*, 2007, pp. 167–184.

[67] J. Dangermond, "A Classification of Software Components Commonly Used in Geographic Information Systems," in *Introductory Readings in Geographic Information Systems*, 1990, pp. 30–51.

[68] H. Kriegel, T. Brinkhoff, and R. Schneider, "Combination of Spatial Access Methods and Computational Geometry in Geographic Database Systems," in *SSD '91: Proceedings of the Second International Symposium on Advances in Spatial Databases*. London, UK: Springer-Verlag, 1991, pp. 5–21.

[69] C. R. Valenzuela, "Data Analysis and Modeling," in *Remote Sensing and Geographical Information Systems for Resource Management in Developing Countries*, 1991, pp. 335–348.

[70] A. Guttman, "R-trees: a Dynamic Index Structure for Spatial Searching," in *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1984, pp. 47–57.

[71] R. Finkel and J. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Inf.*, vol. 4, pp. 1–9, 1974.

[72] J. Robinson, "The K-D-B-tree: a Search Structure for Large Multidimensional Dynamic Indexes," in *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1981, pp. 10–18.

[73] J. Nievergelt, H. Hinterberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 38–71, 1984.

[74] R. H. Güting and H.-P. Kriegel, "Multidimensional B-tree: An Efficient Dynamic File Structure for Exact Match Queries," in *GI Jahrestagung*, ser. Informatik-Fachberichte, R. Wilhelm, Ed., vol. 33. Springer, 1980, pp. 375–388.

[75] J. Orenstein, "A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces," *SIGMOD Rec.*, vol. 19, no. 2, pp. 343–352, 1990.

[76] J. M. Patel and D. J. DeWitt, "Partition Based Spatial-Merge Join," *SIGMOD Rec.*, vol. 25, no. 2, pp. 259–270, 1996.

[77] L. Becker, A. Giesen, K. Hinrichs, and J. Vahrenhold, "Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets," in *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases*. London, UK: Springer-Verlag, 1999, pp. 270–285.

[78] P. van Oosterom, "'An R-tree Based Map-Overlay Algorithm," in *EGIS/MARI '94*, Paris, 1994, pp. 318–327.

[79] U. Finke and K. H. Hinrichs, "Overlaying Simply Connected Planar Subdivisions in Linear Time," in *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*. New York, NY, USA: ACM, 1995, pp. 119–126.

[80] U. Finke and K. Hinrichs, "A Spatial Data Model and a Topological Sweep Algorithm for Map Overlay," in *SSD '93: Proceedings of the Third International Symposium on Advances in Spatial Databases*. London, UK: Springer-Verlag, 1993, pp. 162–177.

[81] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable Sweeping-Based Spatial Join," in *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 570–581.

[82] H. Kriegel, T. Brinkhoff, and R. Schneider, "An Efficient Map Overlay Algorithm based on Spatial Access Methods and Computational Geometry," in *International Workshop on DBMSs for Geographical Applications*, Capri, Italy, 1991, pp. 16–17.

[83] M. McKenney and M. Schneider, "Advanced Operations for Maps in Spatial Databases," in *Int. Symp. on Spatial Data Handling*, Jul 2006.

[84] ——, "Topological Relationships Between Map Geometries," in *Advances in Databases: Concepts, Systems and Applications, 13th International Conference on Database Systems for Advanced Applications*, 2007.

[85] J. Dugundi, *Topology*. Allyn and Bacon, 1966.

[86] R. B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. on Computers*, vol. C-29, pp. 874–883, 1980.

[87] M. Erwig and M. Schneider, "Formalization of Advanced Map Operations," in *9th Int. Symp. on Spatial Data Handling*, 2000, pp. 8a.3–17.

BIOGRAPHICAL SKETCH

Mark McKenney was raised in Beaumont, Texas where he attended Monsignor Kelly High School. Upon graduation, Mark attended Tulane University in New Orleans, Louisana where he completed a BS in computer science in 2003 and a MS in computer science in 2004. He then began his Ph.D. studies at the University of Florida. Mark earned his Ph.D. in Computer Engineering in August 2008, and then joined the Department of Computer Science at Texas State University as a tenure-track assistant professor.