Extending Relational DBMS for Spatiotemporal Information

by

Xiaomei Xu

SIMON FRASER UNIVERSITY

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
July 1990

Approval

Name: Xiaomei Xu

Degree: Master of Science

Title of Thesis: Extending Relational DBMS for Spatiotemporal Information

Examining Commitee:

Dr. Stella Atkins, Chairman

Dr. Jia-Wei Han
Senior Supervisor

Dr. Wo-Shun Luk

Dr. Nick Cercone

Dr. Thomas K. Poiker
Department of Geography
Simon Fraser University
External Examiner

July 5, 1990
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Extending Relational Database Management Systems for Spatiotemporal Information.

Author: _____

(signature)

Xiaomei Xu

(name)

July 9, 1990

(date)

ABSTRACT

A spatiotemporal database is a spatial database in which data objects may change their spatial locations and/or shapes with time. We consider three components, *theme, location,* and *time*, in the design. Based on previous studies of spatial and temporal databases, this thesis extends the relational data model to an extended relational model with the flavor of object-oriented databases in order to represent complex data objects with spatiotemporal information. A spatiotemporal query language, STSQL, is developed as an extension of the relational database language SQL. Moreover, an efficient spatiotemporal data storage structure and two indexing mechanisms are developed to facilitate the search of spatial data objects changing with time in the specified spatial framework. Sample queries from geographical applications are supplied to demonstrate the power and usefulness of the approach.

ACKNOWLEDGMENTS

I owe a great debt of gratitude to Dr. Jiawei Han for his guidance, support, and encouragement, without which this thesis would not have been possible. Thanks to Dr. Tom Poiker, my external examiner, who made many thoughtful suggestions. I would also like to thank the other members of my examining committee, Dr. Woshun Luk and Dr. Nick Cercone, for discussions and comments on the thesis.

Encouragement from and discussions with fellow graduate students, especially Jie Zhang, made the experience a pleasant one. I am thankful to Ed Levinson who helped greatly to improve the presentation of this thesis.

Sincere thanks are due to my respected parents too, for their love, care and education. This work is dedicated to them.

My deepest appreciation to my husband, Yan Chen, for his support and understanding throughout this entire effort.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

Database management systems (DBMS), which provide high-level data models, data independency, data integration, data security, transaction management, and other facilities, have been widely used to manage textual information in business applications. With the advent of computer vision, graphics, and CAD/CAM technologies, many applications, for example map processing in geography, have evolved representations of the abstracted real world, both textually and graphically. However, it is known that traditional DBMSs have some limits, such as the ability to handle spatial or historical information. Thus it has become essential to develop or extend existing DBMSs to store and manage in an integrated fashion a vast amount of image data (multi-dimensional data) as well as textual data. For managing multi-dimensional data the DBMSs must support complex object representations as well as operations on the objects since these data naturally have topologic, structural and geometric features, such as location, shape, and size. Databases whose contents are continuosly updated, and allow queries of both old and new information must incorporate *time*. DBMSs that handle spatially changing objects over time are called *spatiotemporal* databases.

1.2. Background

From the earliest civilizations to the modern times, spatial data have been collected by navigators, geographers, and surveyors. The spatial distribution of the features of the earth's

surface, or topography, is rendered into pictorial form by cartographers. Whereas topographical maps can be regarded as for general purpose, maps of the distribution of rock types, soil series or land use are made for limited purposes. These special-purpose maps are often referred to as *thematic* maps because they contain information about a single subject or theme. Digital map data, line drawings, and region adjacency graphs are all instances of spatial data that are usually organized in a discrete structural form as opposed to the iconic form of the grey tone or color image. Such structural organizations can be derived by hand-gathering data or by segmenting an image, associating attributes with the image segments, and determining relationships between segments. Essentially, an image is a snapshot of the situation seen through the particular filter of a given surveyor in a given discipline at a certain moment in time. Actually, space is indivisibly coupled with time in geography. More recently, satellite imaging has made it possible to see how landscapes change over time and to follow the slow march of desertification or erosion or the swifter progress of forest fires, flood, locust swarms or weather systems.

This thesis is not concerned with the origin or generation of the sequence of spatial data sets. We assume that a sequence of images, as shown in Figure 1.1, is stored in the database, each having a unique time stamp from when the image was derived, and that the information contained in these images has already been abstracted to higher level descriptions. In other words, an image is composed of a group of spatial objects which are interpreted as points, lines, or polygons. An object denotes a logical entity, such as a city, or a farm, instead of a fixed-size grid cell tessellation. For example, a city can be represented as a point or a polygon depending on the level of detail or the scale of a map. Furthermore, we assume that the spatial relationships among objects in different images have also been established. We will discuss

how to model and represent the objects contained in the image sequence and how to query the past, present, or changed information during some period of time.

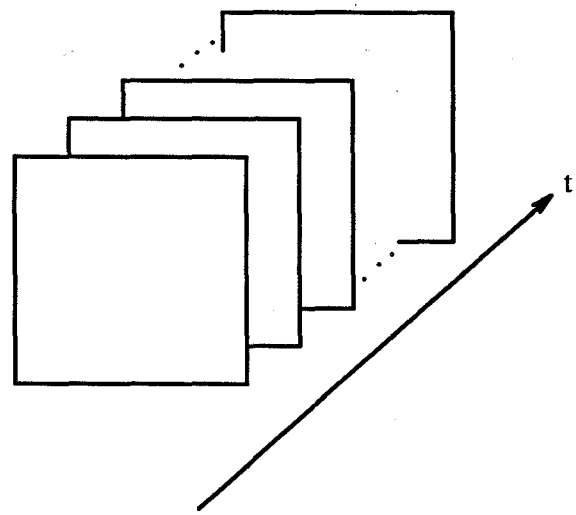

Figure 1.1 A image sequence

1.3. Possible Solutions to Spatiotemporal Database Designs

In order to make a DBMS meet the functionality and performance requirements of such diverse application areas, many efforts have been made in studying a suitable set of data modeling tools, special operations, user interfaces and system architectures. Three suggestions have been made for a more generalized DBMS [Bou85]:

(1) Put together specific DBMS and add a common user interface layer. For example, one DBMS could be for classical data, one for image data, and another for graphical data. The interface would simulate the extended model by converting schemas and queries into their relational counterparts. This is the approach used in GEM, which offers an entity-relationship database interface [TsZ84]. The attractiveness of such an approach lies in its inexpensive implementation using reliable existing technology. Its major shortcoming is its performance. The greater the difference between the end-user model

and the database model, the more complex is the translation process. Eventually this will be inefficient. This method is a short term solution.

(2) Take an existing database model and extend its capabilities. For example, the relational model could be extended for this purpose. This method is considered to be a medium-term solution.

(3) Define a new database model that is suitable for any kind of data and provides extensibility, like an extensible database, semantic database, or object-oriented database model [CDF86, GCK89, Gut89]. This approach is a long-term solution since no efficient implementation of such DBMS has been found sufficient for all kinds of applications, although they exist for specific domains.

This thesis extends the idea of relational database system designs with object-oriented flavor. The emphasis is on extending the relational data model to capture spatiotemporal semantics, to support the extended relational spatiotemporal query language STSQL and physical data organizations. The focus is mainly on new access methods. The relational model has a well understood formal basis that facilitates effective database design and query processing. Furthermore, relational database technology is well established, and the extensions that we will make to the relational model benefit from this state-of-the-art technology. The attraction of the relational model in practice has been that queries in these languages are expressed independently of the way in which relations are stored; thus the user is spared many low level computational problems. Yet another reason for extending the relatonal model is that a relational DBMS is today the dominant choice for most database applications. By contrast, the theory and practice of object-oriented database technology are still at the developmental stage. Consequently, extensions to the systems will enable users who have invested in

relational technology to attain the additional functionality needed for newer applications.

1.4. Organization of the Thesis

This thesis consists of five chapters. In chapter 2, we analyze the requirements imposed on DBMS by some geographical applications and then discuss several ways to extend the relational model in order to represent spatiotemporal information and other structural hierarchies to capture the meaning of the dynamic world. We also define our database schemes and algebraic operations on our proposed databases. Chapter 3 illustrates the query language STSQL, an extension of SQL, which exploits the operations on spatiotemporal attributes in our extended relational model. Sample queries in temporal geographical applications are supplied to demonstrate its capabilities. In chapter 4, we address the physical organization methods. A spatial data structure and a tree-based organization method are developed. Moreover, extensions of the R-tree indexing structure to index on spatiotemporal data are studied with two indexing schemes, MR-tree and RT-tree. The algorithms for constructing the two trees are presented and their computational complexities are analyzed. Finally we provide our concluding remarks and future research directions in Chapter 5.

CHAPTER 2

EXTENDING THE RELATIONAL MODEL TO REPRESENT GEOGRAPHIC INFORMATION

2.1. A Framework for Temporal Geographic Information

Geographic data consists of three components — thematic, spatial, and temporal parts [Bur86, LaC88]. The geographic information is related to and is dependent upon the geographic phenomena of interest to users. Such phenomena have to be spatially modeled and can be represented as geometric descriptions. To model such information, it is necessary to consider the geographic spatial features, relationships and temporal features which are involved in its use.

Geographic Spatial Features

Geographic data may exist in a database as individual objects with spatial features. These discrete objects can be grouped according to their spatial dimensionalities:

(1) 0-D features, such as oil wells or cities;

(2) 1-D features, such as roads, pipelines or rivers;

(3) 2-D features, such as geological regions, land-use zones or municipalities;

(4) 3-D features, such as buildings or hills.

Note that the dimensionality can change depending on the level of detail with which one is concerned [FeP86].

Continuous features, such as the earth's surface, usually require different treatments than those for discrete ones. Our database system is targeted at managing large sets of

discrete, related objects.

Geographic Relationships

There are two broad types of geographic relationships:

(1) Spatial relationships: A spatial relationship states how actual features fit together in space. That is, they involve properties such as the connectivity, adjacency, and proximity of spatial objects. The issue of spatial relationships is important since they need to be retrieved and analyzed in many kinds of applications.

(2) Taxonomic relationships: These relationships concern how classes of features fit together. That is, they describe the hierarchy of classification (or categorization). For example, the general class called "forest" might include specific classes "spruce", "pine", etc.

Geographic Temporal Features

Time and the examination of changes play important roles in spatial information analysis. There are, in general, two types of temporalities:

(1) Aspatial temporalities: Thematic changes, i.e., changes in feature attributes, are called aspatial temporalities. An example would be a farm that has been converted from agricultural use to other uses. We would say that the usage attribute of the farm object has changed.

(2) Spatial temporalities: Changes in an object's spatial definitions are called spatial temporality. One example is geometric changes, such as growth of urban areas causing city boundary changes. If an object's shape is described by a geometric representation, shape changes will be described by geometric temporality . Another example is a change of

topology that describes object adjacencies. Topological temporality pertains to the movement of objects relative to one another. Examples of these situations are shown in Figure 2.1. The striped object could change to a dotted object indicating that its usage attribute has changed to a different value in situation (a); it may also change in shape in situation (b); or its topological relation with another object may change as in situation (c).

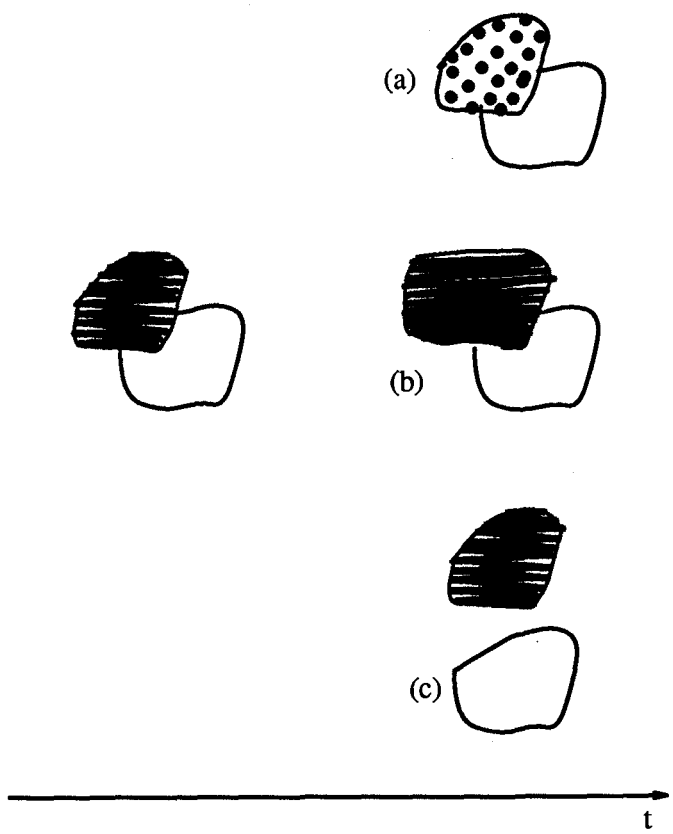

Figure 2.1 Spatial changes.

2.2. Spatial Data Representations

There are three database-oriented representation schemes for solid geometric object modeling which have been examined using different database models [KeW87]. The first is primitive instancing, where each geometric object is defined as a special instance of a generic

primitive object. That is, one relation is created for every generic object type, and the attributes of the relation correspond to the parameters that describe the geometric objects. Each geometric object is stored as a tuple of the relation corresponding to the generic object class. An example of a generic object class is bracket with four holes, as shown in Figure 2.2. The representation scheme of this generic record type would be defined as follows.

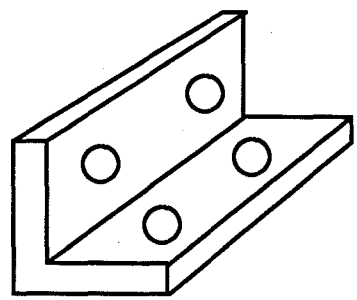

Figure 2.2 A geometric part.

```
generic type BRACKET_4H
        length: real
        width:  real
        height: real
        material: char
        HOLES: array [1:4]
        record
                diameter: real
                location; array [1:3]
        end record
end generic type BRACKET
```

The second method is constructive solid geometry, which is a widely used representation in CAD/CAM systems. In this method an object is described as a composition of a few primitive objects. The composition is achieved via motional or combinatorial operators. The format is defined by the following context-free grammar:

```
<mechanical part> ::= <object>
<object>        ::= <primitive> |
                <object> <motion op> <motion argument> |
                <object> <set operator> <object>
```

<primitive> ::= cube | cylinder | cone | ...
<motion op> ::= rotate | scale | ...
<set operator> ::= union | intersection | difference | ..

For the object "bracket with four holes", the decomposition tree is shown in Figure 2.3. (A triangle "Δ" represents the minus operation which means a part is composed of one sub-part is subtracted from the other sub-part. The sub-parts can be further divided until a primitive is reached. "U" represents the addition operation.)

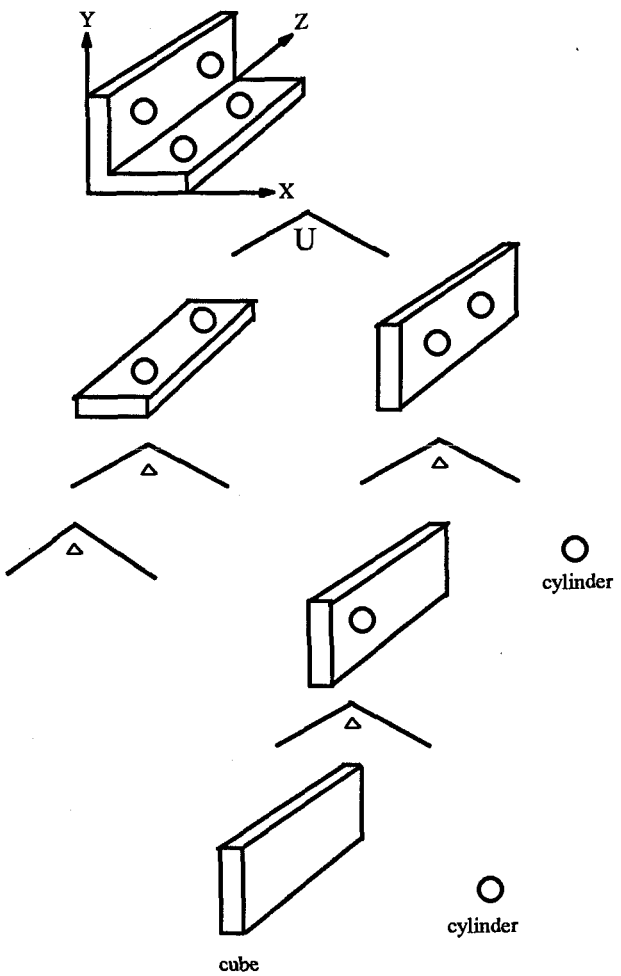

Figure 2.3 A decomposition tree.

The third method is boundary representation. A solid object is segmented into its nonoverlapping faces. Each, in turn, is modeled by its boundary edges and vertices. Again the bracket example is represented in Figure 2.4. Note that this representation scheme consists of

three abstraction levels, that is, faces, edges, and vertices. By contrast, the second method may lead to a deep tree for a complex object.

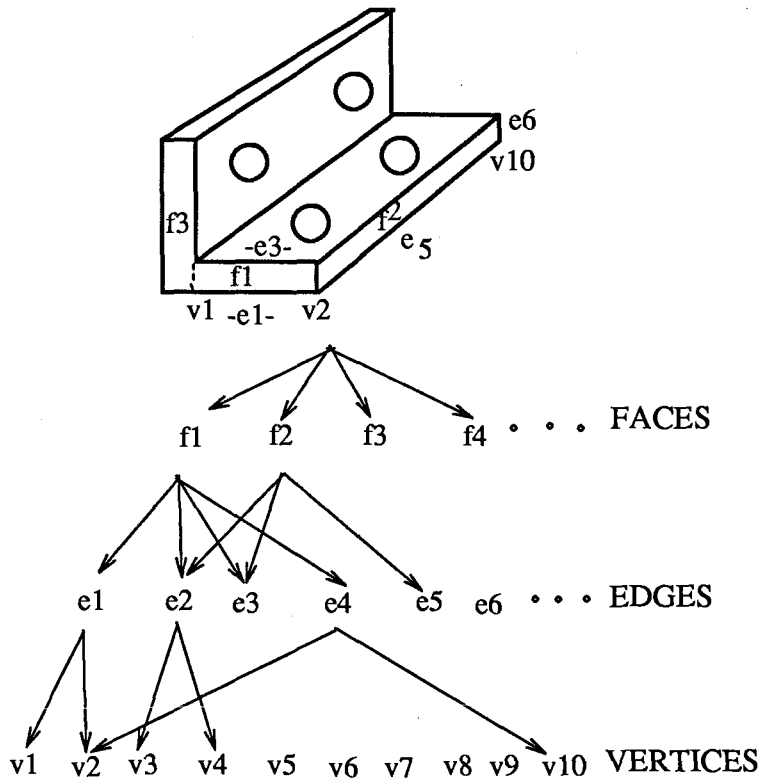

Figure 2.4 A boundary representation.

In geography, entities in one class are unlikely to be exactly the same, as is the case for geometric parts. Moreover, they can not be decomposed spatially into small number of primitive parts which have a constant number of parameters. Consider land use information contained in a 2-D map, as shown in Figure 2.5. The spatial features are the boundaries for different types of regions such as cities and rivers, etc, all in various shapes. Thus only the third method, the boundary representation discussed above, can be considered a representation scheme for geographic information since geographical data can be reduced to three basic topological concepts, regions, lines, and points [Bur86], which are chosen as primitives.

Without the loss of generality, we focus on 2-D map data. The methods can be similarly extended to handle higher dimensional data.

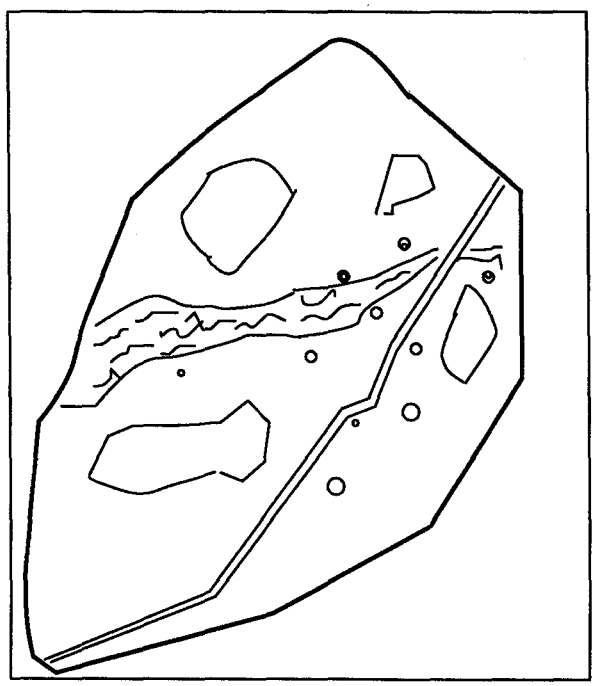

Figure 2.5 A map.

2.3. Relational Models for Geographic Spatial Data

A logical model describes data at the conceptual and view levels, and is used to specify both the overall logical structure of the database and a higher level description of the implementation. At this point we will not examine the formal definition of a relation. What is meant by the terms *attribute* and *primary key* will be provided later in subsequent sections. For now, we will study the ways in which relations are used, and in particular the distinction between storing data about real world objects, and the relationships between such objects. In this section, we describe how to model geographic data using a relational model.

At the conceptual level, geographic data can be viewed as objects which form object classes, and one relation may be created for each class. All features are described in the

attributes of a relation. The relation could have one attribute for object identity and other attributes for an object's location, statistic information, relationships, etc.

2.3.1. Purely Relational Model

The original relational model was proposed by Codd in his seminal paper [Cod70]. This showed that a collection of relations could be used to model the relationships between real world items. The form of a relation is deliberately chosen to be simple. Unfortunately, it is so simple that spatial objects have to be decomposed into different relations. Let us illustrate this on relational boundary representation schema described below in Table 2.1 for the example of map data.

MAP_Component

ID	NAME	REGION
1	city_a	a1
2	lake_1	l1
3	city_b	b1

REGION

ID	LINE
a1	line1
l1	line2
b1	line3

LINE

ID	POINT
line1	point1
line2	point2
line3	point3
...	...

POINT

ID	X	Y
point1	0	1
point2	3	4
point3	2	5
...	...	...

Table 2.1 Purely relational model for a map data.

Notice that a map representation is broken up into different relations, where the relationships among the tuples in various relations are achieved via user-generated attribute values. This

makes the model difficult to use. In order to retrieve and manipulate the data, one is required to have an intrinsic knowledge of the underlying schema definition. In order to display a lake, we have to retrieve all the points which involve joining four relations MAP_Component, REGION, LINE and POINT.

Fortunately, we can extend the database modeling capabilities to handle spatial information in several ways. A fundamental choice for representation of a complex object is whether its structure should be visible or hidden at the level of the relational data model. That is, whether the object should be described by a collection of tuples from various relations, or by a single attribute value from a specific attribute domain for this kind of object. For manipulation, this determines whether the internal structure is accessible to the general facilities of the query language or only to domain-specific operations. The two ways of handling complex objects have been called structural and behavioral object orientations [Dit86, Gut89].

2.3.2. NF^2 Data Model

The NF^2 (non-first-normal form) data model is one of the structural approaches which has originated from database technology and is essentially motivated on technical grounds [RKS88]. The model provides facilities for mapping spatial objects onto database structures and for retrieving these objects as entities. For our map data example, the relation is shown schematically in Table 2.2.

Compared with the purely relational model listed in Table 2.1, the same query, "displaying a lake", that is, "to find the points belonging to the lake", requires about the same number of joins. The NF^2 queries would be at least as complex as in the pure relational model [KeW87] and the model implicitly incorporates references to tuples of different relations. Thus it is a hybrid of the relational and hierarchical data model. There is, however, a data

redundancy problem; for example, p_2 is an end point of lines l_1 and l_2, but the same point has to be stored for each line.

MAP_Component						
ID	NAME	REGION				
		ID	LINE			
			ID	POINT		
				ID	LOCATION	
					X	Y
1	city_a	a1	line1	point1	0	1
				point2	3	4
			line2	point2	2	5
				point5	...	...
...	...	...	...	...	...	...

Table 2.2 NF^2 model for the map data representation.

2.3.3. Abstract Data Types

Extensibility can be provided via new data types, operators, and access methods [GCK89, Gut89]. One approach for behavioral extensions to relational database systems is the use of user-defined data types such as geographic attributes, and their methods. The behavioral approach has an application-oriented flavor. The identification of an object is largely determined by what the user perceives to be an entity that can be manipulated as a whole. Its internal representation should be hidden from the user. In geography, the basic spatial entities are regions, lines and points. Once these three data types as well as operations on them are defined, then they can be used as any other built-in data types. That is, attributes can be of a type that has been previously defined by the user as a data type. The graphical input

and output are example operations on these data types.

Returning to the map data example to demonstrate this method, in Table 2.3, the boundary of a city or other 2-D entity is represented under the data type *REGION*, while a road is stored in another relation which has the data type *LINE* to represent the course of a road.

MAP_Component

ID	NAME	REGION
1	city_a	a1
2	lake_1	11
8	city_b	b1

Road_map

ID	NAME	LINE
4	H_22	line30
10	F_1	line27
...	...	...

Table 2.3 Two relations with extended data types — REGION and LINE.

To implement the spatial domain, such as REGION or LINE, there are two choices. One is that a new data type is defined on top of the relational model. For example, a data type can be a "structured" NF^2 object or network-like structure [Mit89]. In the above example, a REGION object is seen as a string of characters in a table such as a1, a2 by ordinary users, but it is represented internally as a nested structure similar to table 2.2. The only advantage is that a user is relieved from knowing the complex internal structure if the query language provides the operations on the abstract data types. The disadvantage is its inefficiency. Another way is

to modify the physical model to have long fields. This will be discussed in Chapter 4.

2.4. Relational Models for Geographic Relationships

A thematic map can be stored in a relation table as in our map data example. Spatial relationships of the objects in the map are implicitly embedded and can be detected using special functions provided by the query language. These will be studied in Chapter 3. Of course they can also be modeled explicitly at the relational level. For example, a binary relation ADJACENCY can be used to represent any two objects which are neighbor. However, this requires excessive space and involves redundancy. Nevertheless, there is an advantage to storing the spatial relationships at the relational level; a query may be sped up since only table lookup is performed and no computation is required. This is a space-time tradeoff.

Taxonomic relationships concern relationships among objects classes, subclasses and individuals which are usually organized into hierarchies or networks. These relationships should be expressed explicitly in our relational model in order to allocate individuals to classes or derive all instances of a class. The classification used for one purpose often seems to cut across the classification requirements of another purpose. To model the hierarchy, the relational model should be enhanced to support the domain whose values are names of relations [BaA90] or the attributes of type "component_of". General N:M relationships can be expressed by attributes of type "reference". The association of tuples is achieved via the so-called surrogate attributes [Cod79]. In such a relational model, a user must stimulate pointers by comparing identifiers in order to traverse from one relation to another (typically using the join operator). In contrast, the attributes of semantic models may be used as direct conceptual pointers. Thus, users must consciously traverse through an extra level of indirection imposed

by the relational model. For this reason, the relational model has been referred to as being value-oriented as opposed to object-oriented.

2.5. Relational Models for Geographic Temporal Information

Usually, databases carry the most recent data. As the new data becomes available through updates, the old values are discarded. These changes are viewed as modifications to the state with the out-of-date data being updated to the present one. Such databases are called *snapshot* databases, since they only contain current information which is a snapshot of reality. In geography, however, the past states of a database are valuable and are often used for analysis.

In the database community, the generalization of snapshot databases and their underlying relational model have been recently focused on *aspatial temporal databases*, which represent the progression of states of an enterprise over an interval of time. In such databases, changes are viewed as additions to the information in the database. One way is to base temporal databases on the snapshot model, with time appearing as an additional temporal attribute. The database model does not incorporate temporal attributes, instead, the query language must translate queries and updates involving time into retrievals and modifications on the underlying snapshot relations [Ari86, Sno87]. Another approach is to extend the semantics of the relational model to incorporate time directly, as in NF^2 or the time-stamping attribute method [TaG89]. In this thesis we will examine a model, which is, in some sense, intermediate between these two, where time varying data is visualized as a time sequence collection.

A more theoretical research topic on temporal databases is the formulation of the semantics of time, which is closely related to knowledge representation issues [ClM90]. Clifford and Warren have suggested using the entity-relationship model for formulating intensional logic [ClW83]. This logic serves as a formalism for the temporal semantics of a temporal database much as first-order logic serves as a formalism for the snapshot relational model. This issue, however, is beyond our scope.

Conceptually we can view a temporal relational database as a data cube whose depth is the time dimension. There are three choices to represent the time dimension which are (1) relational level versioning, (2) tuple level versioning, and (3) attribute level versioning.

Relation level versioning stores a new version as a separate table with the same schema but different time stamp whenever any of its attributes changes. As an example, a relation, *parts*, whose content changes with time, is shown in Table 2.4. In this method, both changed and unchanged information are all stored as a new snapshot. As a result, much redundancy is involved.

	NAME	COLOR	SIZE
01 Jan 1980	A	red	10
	B	blue	20
12 May 1981	A	red	10
	B	blue	20
	C	red	30
15 Oct 1981	A	white	10
	C	red	30
	D	green	40

Table 2.4 Parts represented by three relations.

Tuple level versioning marks each tuple with time. Usually two special attributes *FROM* and *TO* are used to represent valid time in the relational scheme. New tuples are added by appending them to the relation. Tuples are updated by amending their time attributes. The result is a large table that is accessed by matching tuples to specific time, and, as a result, querying for all the data at a specific time will suffer from such a growth. For example, present tense data may be queried more often than past tense data. The *parts* relation, represented in tuple level versioning, is shown in Table 2.5.

NAME	COLOR	SIZE	FROM	TO
A	red	10	01 Jan 1980	14 Oct 1981
B	blue	20	01 Jan 1980	14 Oct 1981
A	white	10	15 Oct 1981	now
C	red	30	12 May 1981	now
D	green	40	14 Oct 1981	now
...	...	...	...	...

Table 2.5 Parts relation represented in tuple versioning method.

If we sort on an attribute, such as NAME or FROM, we could get a different view. Table 2.6 is the result of sorting on NAME attribute from Table 2.5.

NAME	COLOR	SIZE	FROM	TO
A	red	10	01 Jan 1980	14 Oct 1981
A	white	10	15 Oct 1981	now
B	blue	20	01 Jan 1980	14 Oct 1981
C	red	30	12 May 1981	now
D	green	40	14 Oct 1981	now
...	...	...	...	...

Table 2.6 Parts sorted on the NAME attribute.

NAME	COLOR	FROM	TO
A	red	01 Jan 1980	14 Oct 1981
A	white	15 Oct 1981	now
B	blue	01 Jan 1980	14 Oct 1981
C	red	12 May 1981	now
D	green	14 Oct 1981	now
...	...	...	...

Table 2.7 Tuple versioning for COLOR attribute of the parts.

NAME	SIZE	FROM	TO
A	10	01 May 1980	01 Jan 1982
A	11	02 Jan 1982	now
B	20	01 Jan 1980	01 May 1980
B	30	12 May 1980	14 Oct 1981
C	30	12 May 1981	now
D	40	14 Oct 1981	now
...	...	...	...

Table 2.8 Tuple versioning for SIZE attribute of the parts.

If another attribute, SIZE, in the *parts* relation changes differently from the COLOR attribute, the information should be horizontally split into two tables, one recording the beginning and ending time of the color, and the other recording that of size; the size and color of a part do not necessarily have the same beginning and ending time. This forced splitting of the horizontal format of a relation is called the *horizontal anomaly*. As a result, a logical unit of information, such as color of part A, is also split into several tuples. The forced splitting of a logical unit of information into more than one tuple is called the *vertical anomaly* [GaV85].

The third approach to managing changing information is attribute level versioning. Attribute versioning requires fields with variable lengths to hold lists of time-stamping

attribute values. A nested historical relational model is defined where each attribute value is kept as a < *time interval* , *value* > pair so that each attribute can have its own time interval. As a result, the attribute versioning method expresses the lifespan of a logical unit better than other constructs. Attribute time stamping also avoids storing unchanged information, which happens in relational level versioning, and avoids tuples being broken into unmatched versions within or across tables, which is common in tuple level versioning methods. However, the attribute versioning method may cause some redundancy when two or more attributes in a relation change at the same pace, and only one time stamp is needed for these attributes. This problem can be solved by grouping these attributes of a relation into one compound attribute in the physical organization to avoid repeatedly storing the same time stamps. However, this could require more processing time. Another shortcoming is that the whole relation will be searched when only one time slice is needed. However, if such a relation is well indexed, this problem can be alleviated.

NAME	COLOR	SIZE
<01Jan80,now, A>	<01Jan80,14Oct81, red> <15Oct81,now, white>	<01May80,14Jan82, 10> <02Jan82,now, 11>
<01Jan80,14Oct81, B>	<01Jan80,14Oct81, blue>	<01Jan80,01May80, 20> <02May80,14Oct81, 30>
<12May81,now, C>	<12May81,now, red>	<12May81,now, 30>
<14Oct81,now, D>	<14Oct81,now, green>	<14Oct81,now, 40>

Table 2.9 Parts represented in attribute versioning.

2.6. The Extended Relational Model for Geographic Spatiotemporal Information

We have discussed the extended relational model for atemporal spatial and aspatial temporal information. In this section we study an extended relational model to represent spatial objects that are changing with time, that is, *spatiotemporal* information.

To associate temporal characteristics of geographic objects with their spatial features, we propose an extended relational model that is a combination of abstract data type NF^2 and attribute level versioning in an object-oriented way. We consider the logical time, which is when the objects are derived, to be associated with the attributes. First of all, each object has a unique identity, which is explicitly expressed in a relation from the *OID* domain. The *OID* definition differs from the *key* definition. That is, objects with the same identity in two or more different relations of a database are considered as the same object. Secondly, its corresponding spatial information is represented as an abstract data type. The instances of attributes from spatial domains such as regions, lines, and points are seen as icons, and in a graphical window, they can be viewed as geometric objects, which will be described in Chapter 3 in detail. Thirdly, the temporal information of a spatial object is associated with its spatial attribute, *OID* attribute and other attributes separately. These attributes are recorded in NF^2 fashion. As for spatiotemporal information added, since some formal definitions of nested relational databases have been explored in [RKS88], we will follow their lead and use this formalism to extend the definitions of our databases.

2.6.1. The Model

Definition 2.1: Let T be a set of time points mapped to the natural numbers, i.e., 0,1,2,...,n in increasing order. 0 is the relative beginning point. The symbol n denotes the present time instant and increments as time advances. A **time interval** $t = [\, i\, , j\,]$ is the temporal primitive

and includes all the possible consecutive time points between i and j, where $i, j \in T, i \leq j$, and i is the starting time and j is the ending time.

A time unit is user-defined and can be a second, minute, hour, day, etc. This information is stored in the database dictionary. For instance, if the year is defined as the time unit and the database starts recording events from 1930 to the year 1990, then 1930 is mapped to 0 and 1990 is mapped to 60 and T is $0, 1, 2, \cdots, 60$; a time interval [40 , 50] conatains the years between 1970 and 1980.

Definition 2.2: An **atom** $A = < t , v >$ is the fundamental construct of our model, where t is a temporal interval and v is a value from an attribute domain. Atom A denotes that the value v is valid over t. A **spatiotemporal atom** ST is an atom where v is from the spatial domain. The **key_atom** is an atom where v is an OID that is unique throughout a database. An **ordinary atom** is an atom where v is from other domains.

Definition 2.3: A **database scheme** is a collection of relation schemes $R_1, R_2, \cdots, R_x$ of the form $R_j = (R_{j_1}, R_{j_2}, \cdots, R_{j_n})$, for each j, and $1 \leq j \leq x$, where R_{j_i}, $1 \leq i \leq n$, are names. R_j is internal if R_j appears on the right-hand side of some scheme; otherwise it is external. $R_j = (R_{j_1}, R_{j_2}, \cdots, R_{j_n})$ is an internal scheme if R_j is internal. $R_j = (R_{j_1}, R_{j_2}, \cdots, R_{j_n})$ is a **spatiotemporal scheme** if: (1) R_j is external; (2) R_{j_1} is the name of key attribute which is composed of a $< key\_atom >$ set; (3) any R_{j_i}, $2 \leq i \leq n$, forms a spatial attribute (which is composed of spatiotemporal atoms), ordinary attribute (which is composed of ordinary atoms), or is an internal scheme. An internal scheme may not have a key attribute. A **tuple** of an external relation R_j , denoted as $(e_{j_1}, e_{j_2}, \cdots, e_{j_n})$, where e_{j_1} is $< t , OID >$, and for any i, $2 \leq i \leq n$, e_{j_i} is $<t_1,v_1>, \cdots, <t_k,v_k>$, such that $t_l \cap t_m = \varnothing$ and $t_1 \cup t_2 \cup \cdots \cup t_n = t$, for all l

and m, $1 \leq l < m \leq k$, represents a history of an object. That is, those atoms of an attribute in a tuple have disjoint temporal components which are within the time interval of the key atom.

An attribute is a column of a relation which contains a homogeneous set of data of the same type. If a relation contains all the objects in a map, the spatial history of the map from the above definition is represented by the spatiotemporal attribute of the relation. Each spatiotemporal atom of an object specifies a spatial description of the corresponding object existing during a certain time interval that is the temporal component .

We assume an object changes in space consecutively and continuously so that we do not need disjoint time intervals for one atomic value. As an example of such a scheme considering the previously discussed map data example, the following table demonstrates our approach.

ID	NAME	BOUNDARY
		REGION
<0,20, id1>	<0,20, city_a>	<0,3, a.1> <4,5, a.2> <8,12, a.3> <13,20, a.4>
<0,20, id3>	<0,20, lake_1>	<0,1, 1.1> <2,8, 1.2> <9,20, 1.3>
<0,13, id8>	<0,13, city_b>	<0,13, b>

Table 2.10 Extended relational model for the map data.

The instance <0 , 3 , a 1.1> means city_a had the boundary representation a1.1 during time <0 , 3>. a1.1 is a symbolic notation whose spatial expression can be displayed on a graphical window. An object is unlikely to have two atoms $<t_i , t_j , v>$ and $<t_k , t_q , v>$ where t_j and t_k are not consecutive. That is, the features of an object may change periodically.

If this situation does occur, for example, city_a has <0 , 3 , a 1.1> and <20 , 30 , a 1.1> instances in its boundary attribute, we could store the two atoms as two instances in time ascending order, or we could define the *time set* as a set of disjoint time intervals. In ther latter case, the atom should be $< s , v >$, where s is a time set and v is a value.

Time, as captured in temporal databases, is not an isolated concern but rather an inseparable feature of operational data. Our database model consolidates the operational and temporal concerns into a single unit. The discrete and finite set of successive relation instances completely describes the development of the enterprise captured by our data model over the entire period of time covered by the database. That is to say, our model can capture the completeness of an object's development in a logical sense.

The atomicity of events provides the level of detail needed for retrospective restructuring of information. In our database, how often to gather the spatial data defines the temporal density or the so-called granularity. The state of the database, or any object in it, at any point of time during the period between two successive events, can be inferred from available database states. The most common rule for derivation of such intermittent states is that a recorded value prevails until being changed by the recording of a subsequent event. Otherwise the unavailable information should be explicitly stored as *unkown* values.

Furthermore, by treating the geographical data as logical objects, the user is provided with a high level view. The ADT approach for spatial objects relieves users from reading or manipulating coordinates themselves, and the representation of geometric objects is hidden from the users. Using this method, the change from vector representation to raster or vice-versa would not cause the relations to be redefined or application programs to be rewritten, only the corresponding operations on spatial data would be correspondingly modified.

Another reason to choose the abstract data type is that image data are conducive to visual representations so that different display facilities may require different operations. Attribute versioning plus NF^2 expresses clearly the time existence intervals for an object and an explicit time span for each of its properties.

2.6.2. The Relational Algebraic Operations

Standard relational algebraic operations can be applied to our extended spatiotemporal relations with some modifications to deal with issues created by the spatial data types and temporal components. There are five fundamental operations of the relational algebra: *select* , *project* , *cartesian product* , *union* , and *set difference* . The first two operations are unary operations while the other three are binary operations. Two extra operations *spatial join* and *unary function* operations are introduced here for handling relations with spatial attributes. We will not discuss some operations either concerning only nested structural relations, such as *NEST* and *UNNEST*, which are defined as in [RKS88], or naming, renaming an object, as well as other operations that do not involve retrieving.

(1) **UNION**: The union of two relations R and S that have the same scheme definition, denoted as $R \cup S$, is the set of tuples which are in R or S if these tuples have different key_atoms, or which are combined tuples if a tuple in R and a tuple in S have the same *OID* in their key_atoms, but possibly different temporal components in their attributes. Those combined tuples agreeing on the value parts of atoms are coalesced by taking the union of the temporal sets from corresponding components of each tuple.

(2) **SET DIFFERENCE**: The difference between two relations R and S that have the same scheme definition, denoted as $R - S$, is the set of tuples which are in R but not in S , or

that are in both R and S but with the reduced time span from corresponding components of each tuple if they represent the same objects with different temporal components.

(3) **PROJECT**: Projection is an operation that selects specified attributes from a relation, denoted as $\pi_{i,j,\ldots,k}(R)$. This is the same as the standard projection operation except the result is still in nested form instead of a flattened one.

(4) **SELECT**: The selection operation, denoted by σ, identifies the tuples that are defined in Definition 2.3 to be included in the new relation. This operation consists of conditions which contain operands, arithmetic comparison operators, logical operators, as well as the *spatial, temporal,* and *spatiotemporal predicates* that will be defined in next chapter. Operands can be spatial data as long as the comparator is compatible with the operands. A temporal predicate, represented in comparison expressions of temporal components in attributes, can be either $TIME = t$, which generates a view of a database for this or any specific time instance as specified, or $TIME = [t_1, t_2]$ which describes the objects existence in a database during the period specified. When a selection is performed with projection on one attribute, the time in temporal predicates is referred to as the temporal component associated with this attribute. For example, if R is a spatiotemporal relation with a spatial attribute ST, which represents spatial data of objects in a map, the query $\pi_{ST}\sigma_{TIME=n}(R)$ outputs the values of spatiotemporal atoms in attribute ST if their temporal components include n. That is, the output is the present view of the map. If we change the selection predicate of above query from $TIME = n$ to $TIME = i$ and i is mapped to 1960, then the query generates the map for 1960.

(5) **CARTESIAN PRODUCT**: The product of two relations is the concatenation of every tuple of one relation with every tuple of the other one, denoted as $R \times S$. What we are

concerned with is what the meaning of the product should be when both relations have spatial attributes.

Note that in our spatiotemporal databases, the CARTESIAN PRODUCT has no meaning when totally unrelated spatial relations for different temporal values are joined together. Hence, we define the *STJOIN* operation on spatial relations.

(6) **STJOIN**: The spatiotemporal join, denoted as $R \times_{f_T(i,j)} S$, where f is a binary spatial function and T is a temporal component, is applicable only when R and S are spatiotemporal relations which have spatial attributes i and j, and the two spatial attributes are compatible within the function f. We distinguish the following two cases.

(a) *T is given as a pair of time instances, (t_l , t_r);*

Let $er_i = \{(<rt_1 , rv_1 > , \cdots , < rt_k , rv_k >)\}$ be the set of spatiotemporal atoms at colomn i of one tuple of R and $es_j = \{(<st_1 , sv_1> , \cdots , <st_m , sv_m >)\}$ be the set of spatiotemporal atoms at colomn j of one tuple of S. Since $rt_x \cap rt_y = \varnothing$, for $1 \le x, y \le k$, and $x \neq y$, and $st_x \cap st_y = \varnothing$, for $1 \le x, y \le m$, and $x \neq y$, to compute $R \times_{f_T(i,j)} S$ we

i) Compute $R \times S$;

ii) For each tuple of $R \times S$, we generate another column from er_i and es_j which is $<T, f(rv_x , sv_y) >$, where $t_l \in rt_x, t_r \in st_y$, and $rv_x \in er_i$, $sv_y \in es$.

(b) *T is absent.*

To derive the $R \times_{f(i,j)} S$ we

i) Compute $R \times S$;

ii) For each tuple of $R \times S$, replace $R.i$ and $S.j$ with the set, $re_i \times_{\cap} se_j$, $\{< rt_1 \cap st_1 , f(rv_1 , sv_1)> , \quad <rt_1 \cap st_2 , f(rv_1 , sv_2)> , \quad \cdots ,$

$<rt_1 \cap st_k , f(rv_1 , sv_m)>$, $<rt_2 \cap st_1 , f(rv_2 , sv_1)>$, $\cdots$,

$<rt_2 \cap st_k , f(rv_2 , sv_m)>$, $\cdots$, $<rt_k \cap st_1 , f(rv_k , sv_1)>$, $\cdots$,

$<rt_k \cap st_k , f(rv_k , sv_m)>$ } and remove all the atoms with empty time intervals.

(7) **UNARY FUNCTION**: The unary function operation of a relation, denoted by $F(R)$, will generate a relation that is R plus one extra attribute whose values are the results of the unary spatial function F. R is a spatiotemporal relation that has a spatial attribute at column i. The attribute should be compatible with the function F. Assume a tuple of R has the set of spatiotemporal atoms, $st = < t_1 , v_1 > , \cdots , < t_k , v_k >$ at column i. Then the extra attribute of this tuple is defined as $< t_1 , F(v_1) > , \cdots , < t_k , F(v_k) >$.

From the above discussions on the algebraic operations, we now make a final statement: *The set of operations* $\cup , - , \pi , \sigma , \times , \times_f$ *and* F *generates valid relations in our database. That is, the results of the operations are representable in our database.* To verify its validity, we first have to show that it is possible to represent each result relation from the five fundamental operations. This is obviously shown by the result tuples and attributes in the relation construction from the definitions of the operations. Secondly, we need to show that each result of the spatial join and unary function on spatial attributes forms valid attributes. This follows if each spatial function used in our database is well defined on the spatial domain. Readers are referenced to the function definitions in Chapter 3, which accomplish this goal.

CHAPTER 3

STSQL: AN EXTENDED SQL FOR SPATIOTEMPORAL DATABASES

3.1. High Level View of a User Interface

Geographic information is a heterogeneous collection of spatial and non-spatial data which can be managed in our extended relational database system in an integrated way. Conventional query languages of relational databases, designed for storage, retrieval, and manipulation of alphanumeric data, are hard or impossible to be used directly to express queries concerning spatial or spatially changing information and graphical results. Two examples are "find the portion of Canada Highway 1 that is enclosed within the boundaries of city Vancouver", and "find the coverage area changes of B.C. forests between 1970 and 1980". Therefore, additional capabilities are required, such as the retrieval of spatial and temporal information through specified spatial relationships and descriptions, or temporal primitives in spatiotemporal databases. Interactions with graphical input and display of spatial data are also required.

Our user interface differs from conventional information systems by various graphical representations of spatial objects, and the specific interaction between spatial and non-spatial data. Thus the interface of our spatiotemporal database system is an integration of graphical and textual representations. Users articulate their instructions through the dedicated interface to communicate with the system. This interface must include tools and query language for all the essential operations. By treating geographic entities as objects, the interface can provide

users high level manipulations of geographic features, relieving a user from manipulating the complex internal representation of geographic information directly. Geographic objects could be "highway 99", or "farm A", etc, which have feature attributes.

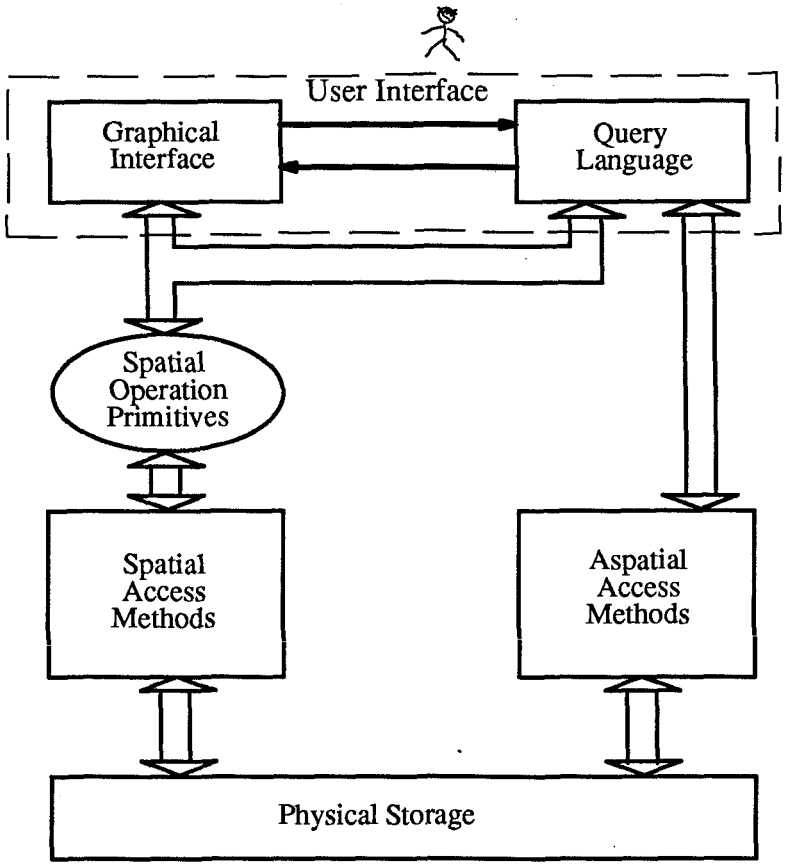

Figure 3.1 The System Structure.

There are three major types of interactions that must be integrated into the user interface. The first is the user interaction with graphically displayed results. [EgF88] provides a good example of the graphical interface design. Next is user lexical conversation with query language, and we focus on extending SQL functionalities. The last one is interaction between query language processor and graphical display facility. Figure 3.1 shows our system architecture. The graphic interface handles visualization like DISPLAY the geographic objects in a graphic window. The STSQL, an extended SQL, handles alphanumerical information in a

tabular form. These two are built on the top of DBMS kernel.

3.2. Graphic Interface Facility

The development of graphic interfaces is encouraged by the advent of workstation technology, which provides relatively inexpensive bit-mapped displays and pointing devices. Interactive-graphic presentation is powerful for mapping systems because the content of maps can be quickly modified. Objects can be added to, removed from, or modified on an existing map without the need to start with a new drawing from scratch again. This requires the graphic interface to provide the tools to manipulate a map. This issue is rather involved with implementation details, so we will not further discuss it.

3.3. Interaction Between Graphic Interface and Query Language Processor

The approach used to extend the relational DBMS does not alter the relational view of data. The results are thus naturally expressed in the form of tables. Moreover, the temporal versions of an object are grouped together for display in a table. A domain is composed of instances and opearations which may be user-defined. Among the set of opearations applicable to an object, a user can define opearations to display and enter data in a way appropriate to human understanding.

Graphical results are represented in a tabular form. These can be portrayed as icons. To display the contents of such a result, the particular icon is pointed to with the help of a mouse. Clicking a mouse on an icon displays the contents of the data in graphical form. More than one object or even a complete layer may be displayed and superimposed on the same window. Figure 3.2 illustrates this possibility.

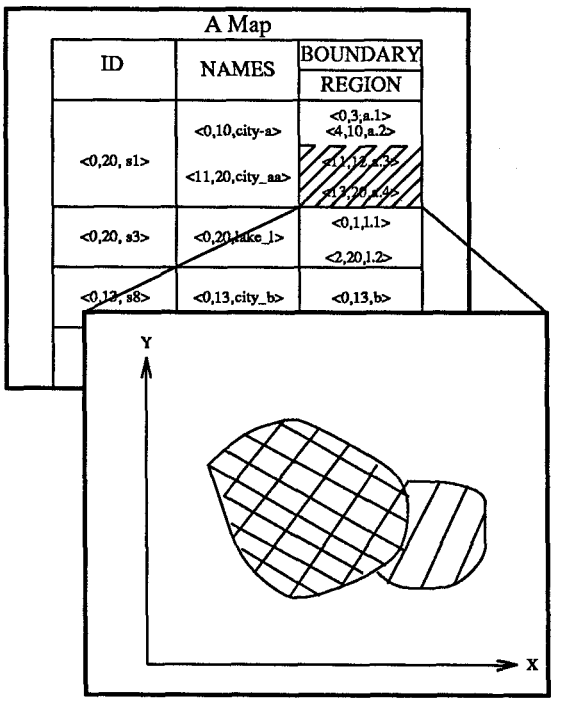

Figure 3.2 Portry the spatial data to graphical window.

3.4. Extending SQL

There have been some studies on expanding query languages for spatial databases and temporal databases in literature respectively. A QBE-like language PICQUERY is designed for PICDMS [JoC88], PSQL is an extension of SQL for pictorial databases [RoF88], and TQuel is an extension of Quel for temporal databases. However, different database models require different query languages manipulating databases in different ways. Here we present our sptiotemporal query language STSQL. The basic specifications of operations in query language reflect the notions of the data model, and its two generic parts directly correspond to objects in our database system, namely, the specifications of atemporal spatial and spatiotemporal information. The syntactic form we have selected is based on a subset of SQL, with necessary extensions for the specification of the query's spatial, temporal aspects. The

relational model requires that relations be defined over domains, each of which has one form of data type or another. STSQL, corresponding to our extended relational model, extends the definition of relations over spatial and spatiotemporal types of domains defined in a disciplined way.

The choice of extending SQL as the basic query interface has been influenced by the increasing acceptance of SQL as the standard interface for relational DBMS because of its simplicity and easiness for learning. Due to the length, we only discuss the STSQL retrieval operations. The number of operators depends on each individual domain and an application. We make no attempt to come up with a universal set of operators. Instead, the database system should be flexible enough to allow the users to define additional specialized domain operators, that is, to provide the DBMS with the extensibility. Such techniques in general have been discussed in [BLW88, CDR86, CDF86, Gut89]. Specificly, the user defined types for managing complex objects and their associated methods in an extensible relational DBMS have been implemented in LISP and C [GCK89].

3.4.1. Temporal Criteria For Data Retrieval

The SQL retrieve statements consist of three basic components: *SELECT* the target list, specifying what to derive or calculate, *FROM* specifying from which relations to select attributes or compute the functions, and a *WHERE* clause, specifying which tuples participate in the derivation. To specify temporal conditions, it is more natural to add the *WHEN* clause instead of embedding them into *WHERE* clause. Similarly, *DURING*, *AFTER*, *BEFORE*, *AT*, *IN* and *SINCE* can be included into the STSQL as temporal constructs. The temporal information will be helpful to answer questions like "What, when, and where did a change occur? ".

WHEN is the temporal analogue to SQL *WHERE*-clause. The clause consists of a temporal conditions and two time variables whose values are attached with attributes concerned. That is, every instance of an attribute has a starting time stamp and ending time stamp. So that the attribute value A is associated with a temporal element $t = [s, e]$ $(\leq [0, n])$ which gives the life span of the value, denoted as $< t, A >$, and s and e can be referenced by A.start_time and A.end_time, respectively. A *WHEN*-clause may explicitly represent the temporal conditions by specifying the time associated to the specific attributes. Alternately, *DURING*, *AFTER*, *BEFORE*, *AT*, *SINCE* and *IN* clauses only specify the time itself which is the time implicitly associated to the attributes occurred in *WHERE* clause or to the attributes concerned in *SELECT* statement. During implementation, these temporal clauses should be transformed to the equivalent *WHEN*-clauses. A *WHEN*-clause can also contain ordinary conditions in the same way as those expressed in a *WHERE* clause, but it indicates the time instances or intervals when the conditions are satified. If the time is not mentioned in query, it implies *present* tense by default. The temporal function *TIME* is defined below in order to get the time when specified conditions are satisfied.

TIME : Returns the time when conditions in *WHERE* are satisfied.

TIME(x) : Returns the time associated to attribute x.

3.4.2. Spatial Criteria For Data Retrieval

We extend SQL spatial functionalities by adding new operators to operate on the spatial data types and to explore the spatial relationships embedded in our relations. Retrieval operations are discussed to show how the operations can be used to answer spatial and/or temporal questions. To query the database and to retrieve information from it, the *SELECT* command

is used. To manipulate the spatial attributes, spatial functions are placed in the *SELECT* command. The spatial predicates can appear in the *WHERE*- or *WHEN*-caluse. We distinguish functions and predicates, where functions return data sets as results and predicates return either *true* or *false*. Functions are usually called in *SELECT* clauses and predicates play roles in conditional clauses. There are four types of functions, namely unary, binary, aggregation, and high level functions. The aggregate functions, similar to that in standard SQL, return a single value as a summary of information about a group of rows in a column. These functions and predicates are defined as follows . We use REGION, LINE, POINT, and NUM to indicate parameters and results of functions are from region, line, point, and numerical attribute domains, respectively. X and Y are used when a function can have more than one type of parameters or results.

Spatial Functions

(1) Unary Functions:

CENTER*(REGION)* → *POINT* : The result is the center point of a region object.

AREA*(REGION)* → *NUM*: The result is the area of a region object.

PERIMETER(REGION) → *NUM* : The result is the perimeter of a region object.

LENGTH*(LINE)* → *NUM* : The result is the length of a line object.

(2) Binary Functions:

INTERSECTION*(X,Y)* → *X* :

If both *X* and *Y* are REGIONs, the result is the common part of two region objects.

If both *X* and *Y* are LINEs, the result is the intersecting points of two line objects.

If *X* is LINE and *Y* is REGION, the result is the segments of a line object which are

included in a region object.

UNION(*REGION, REGION*) → *REGION*: The result is the region which belongs to any of two region objects.

DIFFERENCE(*REGION, REGION*) → *REGION* : The result region is the first region object subtracting the second region object.

DISTANCE(*X,Y*) → *NUM* :

If both *X* and *Y* are POINTs, the result is the distance between two point objects.

If *X* is POINT and *Y* is REGION, the result is the distance between the region center and the point object.

If *X* is POINT and *Y* is LINE, the result is the length of the shortest orthogonal line from a point to a line object.

If *X* is REGION and *Y* is LINE, the distance is measured from the region center to the line that is defined in the same way as the distance between a point object and a line object mentioned above.

Other interpretations are also possible.

(3) Aggregation functions:

MINIMUM(*REGION*) → *REGION* : Returns the region object with the minimum area among a group of rows of one column selected.

MINIMUM(*LINE*) → *LINE* : Returns the line object with the minimum length among a group of rows of one column selected.

MAXIMUM(*REGION*) → *REGION* : Returns the region object with the maximun area among a group of rows of one column selected.

MAXIMUM(*LINE*) → *LINE* : Returns the line object with the maximun length among a

group of rows of one column selected.

AVERAGE*(REGION)* → *NUM* : Returns the average area of a group of region objects selected.

AVERAGE*(LINE)* → *NUM* : Returns the average length of a group of line objects selected.

SUM*(LINE)* → *NUM* : Returns the sum of lengths of a group of line objects selected.

SUM*(AREA)* → *NUM* : Returns the sum of areas of a group of region objects selected.

COUNT*(*)* → *NUM* : Returns the number of values of a group of rows in one column selected.

NEAREST*(POINT)* → *POINT* : Returns the point object which is the nearest to the point among a group of objects selected.

FURTHEST*(POINT)* → *POINT* : Returns the point object which is the furthest to the point among a group of objects selected.

(4) High Level Functions:

MOVING_DIR*(X)* → *SOUTH, SOUTHEAST, etc* : Returns the direction object *X* moved during the time period or the time instance specified in *WHEN* -clause.

MOVING_SPEED*(X)* → *NUM* : Returns the division of a distance and a time period, where the distance is that object *X* shifted during the time period specified in *WHEN* -clause. If *X* is REGION, then the center of a region is concerned.

CHANGING_RATE*(REGION)* → *NUM* : Returns the area changing ratio of a region object during the time period specified in *WHEN* -clause.

INCREASED*(REGION)* → *REGION* : Returns the increased portion of a region object during the time period specified in *WHEN* -clause.

REDUCED*(REGION)* → *REGION* : Returns the reduced portion of a region object during

the time period specified in *WHEN* -clause.

CHANGED*(REGION)* → *REGION* : Returns the changed portion of a region object during the time period specified in *WHEN* -clause. This portion can be the increased or reduced part.

Spatial Predicates

(1) Simple Spatial Predicates:

REGION **INTERSECTS** *REGION* : Returns TRUE if one region object overlaps with another region object; otherwise FALSE.

REGION **IS_COVERED_BY** *REGION* : Returns TRUE if the first region is completely contained in the other region; otherwise FALSE.

REGION **NOT_COVERED_BY** *REGION* : Returns TRUE if the first region is not contained in the other region; otherwise FALSE.

REGION **IS_NEIGHBOR_OF** *REGION* : Returns TRUE if two regions have some common boundary; otherwise FALSE.

LINE **INTERSECTS** *LINE* : Returns TRUE if one line intersects the other line; otherwise FALSE.

POINT **IS_NORTH_OF** *POINT* : Returns TRUE if the first point is located to the north of another point; otherwise FALSE.

IS_SOUTH_OF , IS_EAST_OF, ... : Similarly defined as above.

POINT **WITHIN** *REGION* : Returns TRUE if the point is within the region.

POINT **NOT_WITHIN** *REGION* : Returns TRUE if the point is not within the region.

LINE **INTERSECTS** *REGION* : Returns TRUE if the line intersects the region.

POINT **INTERSECTS** *LINE* : Returns TRUE if the point is on the line.

(2) High Level Predicates:

IS_MOVED(*X*) → *BOOL* : Returns *TRUE* if the center of an object is moved or *FALSE* otherwise during the time period specified in *WHEN*-clause.

IS_INCREASED(*REGION*) → *BOOL* : Returns *TRUE* if the area of an object is increased or *FALSE* otherwise during the time period specified in *WHEN*-clause.

IS_REDUCED(*REGION*) → *BOOL* : Returns *TRUE* if the area of an object is reduced or *FALSE* otherwise during the time period specified in *WHEN*-clause.

IS_CHANGED(*X*) → *BOOL* : Returns *TRUE* if the shape of an object is changed or *FALSE* otherwise during the time period specified in *WHEN*-clause.

3.4.3. Sample Queries in STSQL For Data Retrieval

3.4.3.1. Sample Schema

In geography, a land information database may consist of several layers. Each layer is composed of a sequence of maps derived at different time. A land use map, which classifies areas according to forestry, urban area, wet land, agriculture, and so on, can be represented by a relation *landuse*. We may represent provinces of Canada in one relation, called *province*, which divides a space into subspaces (provinces) according to municipality. Another relation *city* contains the information about cities' locations, and populations, etc. Some other important geographic entities are highways, railways, lakes, rivers, and farmlands. Some of these entities will be wholly contained in a province and others will cross its boundary. As an example, we assume the following relational schemes exist in our database.

city(*id:OID, name:CHAR, in_province:CHAR, population:INT, region:REGION, center:POINT*)

province(*id:OID, name:CHAR, population:INT, region:REGION*)

lake(*id:OID, name:CHAR, region:REGION*)

park(*id:OID, name:CHAR, region:REGION*)

forest(*id:OID, name:CHAR, region:REGION, species:CHAR*)

highway(*id:OID, name:CHAR, course:LINE, width:INT, condition:CHAR*)

river(*id:OID, name:CHAR, length:REAL, course:LINE, width:REAL, dept:REAL*)

soil(*id:OID, type:CHAR, region:REGION, area:REAL, condition:CHAR, class:CHAR*)

landuse(*id:INT, name:CHAR, area:REAL, region:REGION, usage:CHAR*)

There are three types of queries as far as spatiotemporal information is concerned. One is to get the time when spatial or other conditions are satisfied. The second one is to get spatial or other information at a specific time instance or during a period. The last one is mixed of the above two kinds of queries. We use the following examples to explain how temporal and spatiotemporal information is retrieved and how those spatiotemporal functions and predicates are used one by one.

3.4.3.2. Spatial Query Examples

In this section, we use the following examples to demonstrate how spatial information should be queried.

(1) Find the area and perimeter of a given region, say, B.C. province.

SELECT **AREA**(*region*), **PERIMETER**(*region*)
FROM province
WHERE province.name = "B.C."

The above query is an example of using unary functions.

(2) Find the cities within Alberta.

SELECT C.name
FROM city C, province
WHERE province.name = "Alberta"

AND city.center **IS_INSIDE** *province.region*

Some queries can be retrieved directly from an attribute search , others need first search and then compute. The user himself may decide to choose a direct attribute searching or computing, or leave the problem to the system semantic optimizer. For example, when a query is to find the area of a province, if there is an *area* attribute in province relation, the user should retrieve this attribute instead of calling AREA function to compute the area. Similarly, in the above example 3, since there is an attribute *in_province* in relation city, the query "find the cities in Alberta province" can be written in the following way.

SELECT name
FROM city
WHERE city.in_province = "Alberta"

(3) What is the total area of regions with 'clay' soil type and the regions are forest?

SELECT **SUM(INTERSECTION***(S.region , L.region))*
FROM soil S , landuse L
WHERE S.type = "clay"
AND L.usage = "forest"

In this example, both *soil* and *landuse* relations have an *area* attribute. However, the information is useless here since we need to generate new regions that satisfy the two conditions, that is, the soil type should be 'clay' and this land is also forest. Such areas are summed up by nesting the *INTERSECTION* in the *SUM* function.

(4) Find the highways which intersect with highway 55.

SELECT G.name, G.course
FROM highway H , highway G
WHERE H.name = "hwy-55"
AND H.course **INTERSECTS** *G.course*

This example shows how the predicate INTERSECTS is used for detecting one line segment intersecting another.

(5) Find the lakes one part of which is in B.C. and the other is in Alberta.

```
SELECT L.name, L.region
FROM lake L, province P
WHERE P.name = "Alberta"
 AND L.region INTERSECTS P.region
 AND L.region INTERSECTS
        SELECT region
        FROM province
        WHERE name = "B.C."
```

The predicate INTERSECTS in this example is used for detecting the overlapping of two regions.

(6) Find the lakes within B.C. province.

```
SELECT lake.name, lake.region
FROM lake , province
WHERE province.name = "B.C."
 AND lake.region IS_COVERED_BY province.region
```

This example shows that to detect one region within another, the spatial predicate IS_COVERED_BY can be used.

(7) Find the land whose distance to a highway with good condition is less than 500 meters.

```
SELECT S.id, S.region, H.course, DISTANCE (S.region , H.course)
FROM soil S, highways H
WHERE H.condition = "GOOD"
 AND DISTANCE(S.region , H.course) < 500
```

The function DISTANCE appears in *SELECT* and *WHERE* clauses. The result of the function can be treated as the simple value which may be compared with other values if the function is used in *WHERE* statement.

(8) Find the distance between Calgary and Vancouver.

```
SELECT S.name , T.name , DISTANCE(S.center, T.center)
FROM city S, city T
WHERE S.name = "Calgary
 AND T.cname = "Vancouver"
```

The function DISTANCE is for calculating the distance between two points which are

from two tuples of one relation. There are no temporal clauses in the above queries, that is, only the present information is concerned. For example, in the last query, the spatial join is on the relation S and T which are the aliases of relation *city*. The function **DISTANCE** takes the the *center* attribute of the relations S and T, where one is the center of *Calgary* and the other is that of *Vancouver*, and output the distance of the centers with present time stamps. A binary function generates a new attribute, such as *DISTANCE* generates an attribute whose values are from numerical domain, and *INTERSECTION* generates a spatial attribute when two relations are *joined* together. Notice that in the queries, no functions appeared in *SELECT* statement and only some spatial predicates are involved in *WHERE* clause. In such a situation, spatial join is performed in a similar way as if a spatial function is involved and the only difference is that the result of a spatial calculation will not be output and is only used as a condition checking.

3.4.3.3. Temporal Query Examples

Temporal information can be used as a condition such as in a *WHEN* statement. This condition can be a specific time instance or a time interval. Time as a value can also be extracted from our databases while certain conditions are satisfied. Such examples are as follows.

(1) What was the population of Burnaby in 1970?

```
SELECT population
FROM   city
WHERE  city.name = "Burnaby"
IN     1970
```

(2) Print present population of each city in B.C. province.

```
SELECT name, population
FROM   city
```

WHERE city.in_province = "B.C"
AT PRESENT

where the *AT* PRESENT is optional.

The above two examples are simple temporal queries which only concern a particular time instance and no spatial information is required.

(3) How many highways were built between 1970 and 1980?

SELECT **COUNT***(id)*
FROM highway
DURING 1970 , 1980

(4) How many species were in forest A before 1960?

SELECT **COUNT***(species)*
FROM forest
WHERE forest.name = "A"
BEFORE 1960

The above two examples are temporal range queries which concern the objects that existed within a period of time.

(5) Find all the cities which are located to the north of Calgary and has existed for less than 80 years.

SELECT S.name
FROM city S, city T
WHERE T.name = "Calgary"
AND S.center **IS_NORTH_OF** *T.center*
MINUS
SELECT S.name
FROM city S, city T
WHERE T.name = "Calgary"
AND S.center **IS_NORTH_OF** *T.center*
IN 1910

In the above example, we first select all the cities which are in the further north than Calgary. The second step is to derive the cities existed at least for 80 years in the subquery. The last step is to subtract the subquery result in the second step from the result from the

first step.

(6) When did the area of forest A become less than that of forest B?

```
SELECT TIME
FROM  forest F, forest E
WHERE  F.name = "A"
 AND  E.name = "B"
WHEN  AREA(F.region) < AREA(E.region)
```

The *WHEN*-clause in this query is to extract the time at which forest A became smaller in area than forest B. This is still refered to as temporal query with a spatial condition.

3.4.3.4. Spatiotemporal Query Examples

In this section, the queries with spatial and temporal information are discussed.

(1) What did the lake L look like 100 years ago?

```
SELECT name , region
FROM  lake
WHERE  lake.name = "L"
IN  1890
```

This query is a simple spatiotemporal query which is to display the spatial data at a specified time instance.

(2) Show the region of Vancouver when its population exceeded 10,000.

```
SELECT city.name , city.region
FROM  city
WHERE  city.name = "Vancouver"
WHEN  city.population > 10,000
```

Notice in the above query that a condition is expressed in *WHEN* clause which is different from putting the same expresion in the *WHERE* clause. After the selection of the tuple whose name is Vancouver from the relation *city*, the *WHEN* statement extracts the temporal part of the first atom in the *population* attribute where the value of the population is greater than 10,000. Within the tuple, we then select an atom from the

region

attribute whose temporal part includes the time point extracted. If we remove the *WHEN* clause and put the condition in the *WHERE* clause, the *TIME* will be present time and the present version of the region will be the output.

(3) What is the area of agricultural land in the south-east of the city Calgary that has not been under cultivation at least for two years?"

```
SELECT soil.id, soil.region, AREA (soil.region)
FROM   soil, city
WHERE  city.name = "Calgary"
 AND  soil.type ≠ "crop"
 AND  soil.region IS_SOUTHEAST_OF city.center
SINCE  1988
```

or substitute *SINCE* clause with *WHEN* clause

```
WHEN   soil.type.start_time >= 1988
 AND   soil.type.end_time = present
```

(4) Determine all forestry regions in province B.C. converted to agriculture use during 1970's.

```
SELECT landuse.id
FROM   landuse, province
IN     1970
WHERE  landuse.usage = "forest"
 AND  province.name = "B.C."
 AND  landuse.region IS_COVERED_BY province.region
 AND  landuse.id
 EXISTS SELECT id
            FROM   landuse
            WHERE  landuse.usage = "crop"
            IN     1980
```

This query requires attribute search. In the land use map, we first find those lands whose usage attributes have value as forest in 1970 and value as crop in 1980. The query can only return those objects which have undergone attribute changes from one to another,

but not including those objects which were partially converted to crop. That is, these forests were reduced in size. In order to detect the partial convertions from forest to crop, we have the following STSQL statements.

```
SELECT CHANGED(landuse.region)
FROM   landuse
DURING 1970 , 1980
WHERE  landuse.id
EXISTS SELECT  landuse.id
          FROM   landuse , province
          IN     1970
          WHERE  landuse.usage = "forest"
           AND   province.name = "B.C."
           AND   landuse.region IS_COVERED_BY province.region
           AND   landuse.region
                 INTERSECTS
                 SELECT landuse.region
                 FROM   landuse
                 WHERE  landuse.usage = "crop"
                 IN     1980
```

The above statements are still not sufficient to answer the partial conversion query since a forest may change its covering area and some parts may be changed to crop and some to others although it overlaps with a crop region. In order to give the exact area of convertion, we have to find the region of a forest in 1970 which overlaps with the region of a crop in 1980. The intersection of the 1979 version of the forest area and 1980 version of the crop area is the actual area which is converted from forest use to agriculture use. The following is the correct answer.

```
SELECT INTERSECTION(L.region , M.region)
FROM   landuse L , landuse M, province
WHERE  L.usage = "forest"
 AND   M.usage = "crop"
 AND   province.name = "B.C."
 AND   L.region FBIS_COVERED_BY province.region
DURING 1970 , 1980
```

(5) Determine Vancouver urban area growth rate during 1970s."

SELECT **CHANGING_RATE** *(region)*
FROM city
WHERE city.name = "Vancouver"
AND **IS_INCREASED** *(city.region)*
DURING 1970 , 1979

(6) How much has the Vancouver urban area been increased during 1970s.

SELECT **INCREASED** *(region)*
FROM city
WHERE city.name = "Vancouver"
DURING 1970 , 1979

This question can also be queried in the following way if we do not use the spatiotemporal function. Notice that the C.area intends to be the area in 1970 and B.area means the area in 1979.

SELECT **DIFFERECE** *(C.region, B.region)*
FROM city C city B
WHERE C.name = "Vancouver"
AND B.name = "Vancouver"
WHEN C.area.start_time $\leq$ *1970*
AND C.area.end_time $\geq$ *1970*
AND B.area.start_time $\leq$ *1979*
AND C.area.end_time $\geq$ *1979*

The above four examples are spatiotemporal range queries. In these queries, when a time range is given, we can use the functions either to detect the spatial changes or to check the existence of some spatial features.

The default options in retrieval operations are defined such that a query that omits the spatial or temporal portion retains the standard meaning of the corresponding SQL SELECT operation. To summarize, the STSQL format is laid out in the Appendix. The syntax is presented in Backus Normal Form.

3.5. Implementation of STSQL Functions

We have proposed the spatial functions and predicates which take parameters from spatial domains and output spatial data, numerical values, or other values correspondingly. Temporal information is used in temporal clauses as conditonal statements to decide which version of a spatial representation to participate in an operation and what temporal component to be attached to the result atom value. In our extended relational database, a column of a relation contains a homogeneous set of data of the same type. Spatial functions are applied to spatial columns and the results are generated as new columns. Once spatial data in such a column is selected as a spatial function parameter, what has been left is to use geometric algorithms to efficiently compute the function.

Unary and binary functions require calculations geometrically, such as *AREA*, *CENTER*, and *INTERSECTION*, which require calculating an area, finding a center point, and generating an intersection portion by a given representation, respectively. For example, a simple polygon can be divided into several triangles and the area of the polygon is the sum of each triangle's area. This can be done in linear time once the triangles have been created. If a region is represented in a quadtree-based raster method (see Chapter 4.7.1), *AREA* is derived by accumulating the areas of related leaf nodes. If 3-D data is involved, function *VOLUME* should be efficiently processed. These issues are studied in computational geometry [CoH79].

The spatial aggregation functions are nothing more than searching the spatial data and recording either the maximun, minimun, or other values to do comparisons or calculations. But the implementation of spatial aggregate functions is time consuming since it may need to search an entire map. In some cases, we wish to avoid exhaustive search of the whole plane, i.e., the spatial attribute of a relation or even the whole database, thus searching strategies are

necessary. As an example, to compute the aggregation function *NEAREST*, we first locate where the target object p is , and make a circle as small as possible such that its center is p and it encloses a few points, so that the search space is narrowed down to a smaller region.

As for the high level functions, we could decompose them into several unary, binary functions and other primitive operations. These operations then can be solved one by one. For example, the function *CHANGE_RATE*(a) can be derived in the following steps. Firstly from the temporal clause we decide the two time points t_s and t_e. The second step is to call *DIFFERENCE* function to derive a new region which is the difference between two versions of area a at the two time points. The third step is to call funtion *AREA* to calculate the area of the derived region. The last step is the division of two numbers, $area / (t_e - t_s)$. Spatial predicates can be processed in a similar way as spatial functions except returning boolean values. An alternative of deriving the same result could be making some modifications in the second and third steps of the above procedure. Do the calculation of the areas of the two versions of spatial objects concerned in the second step. Then in the third step we get the difference between these two areas. This could sometimes mean a potentially efficient solution, since we may have already stored a attribute *area* in the original relation, thus avoid the need of generating the new relation which represents the difference of the two spatial objects at these two time instances.

Another operation, overlay, which is commonly required in geographic applications, can be performed by a spatial join with the *INTERSECTION* function on two spatial relations. Each relation contains spatial objects whose spatial data in the *REGION* attribute forms a thematic map. The "overlay" concept is that the real world is portrayed by a series of layers in each of which one aspect of reality has been recorded (e.g., topography, soil type, roads, and

rivers, etc). The result of the operation is a new relation that is composed of new spatial objects which partition the plane into disjoint regions. Each resulting region is the intersected portion of two spatial objects in the two thematic maps, respectively.

In spatial databases the calculation of functions harmonizes with the data structures and geometric algorithms, but more important issues for database researchers are the searching efficiency and the storage efficiency which are to be discussed in Chapter 4.

CHAPTER 4

PHYSICAL ORGANIZATION OF SPATIOTEMPORAL DATA

4.1. Mapping Relations to Files

A database relation at the physical level may be viewed as a collection of records. The DBMS performance depends on the efficiency of the data structures used to store the records in the database and on how efficiently the system is able to operate on these data structures. As is the case elsewhere in computer systems, a tradeoff must be made not only between space and time, but also between the efficiency of one kind of operation versus that of another.

There are alternative methods of mapping a relation to a file [KoS86]. The simple approach is to store each relation in a separate file. Thus the database system may take full advantage of the file system functionalities provided as a part of the operating system. It is usually the case that tuples of a relation can be represented as fixed-length records, that is, relations are in the first normal form. However, in our extended relational database model, we have introduced the spatial attributes which usually have variable length of data and their time varying versions are organized in the non-first-normal form. Obviously, the fixed-length method and the one-file-per-relation method are not directly applicable to our system. In another approach, a database system does not rely directly on the underlying operating system for file management. Instead, one large operating system file is allocated to the database system. All relations are stored in this single file, and the management of the file is left to the database system.

Data in databases are transferred between disk storage and main memory in units of a block. Since a spatial object may need more than one block to represent its spatial features, pointers are used to trace the spatial data when a relation is stored on the tuple-by-tuple basis such that the relation can be organized in a traditional way or in the partial normalized storage model for aspatial nested relations discussed in [HaO88] as if spatial information is not introduced. That is, the spatial attribute should be stored separately from other aspatial attributes as shown in Figure 4.1. In this chapter we study how the separately stored spatial data can be structured.

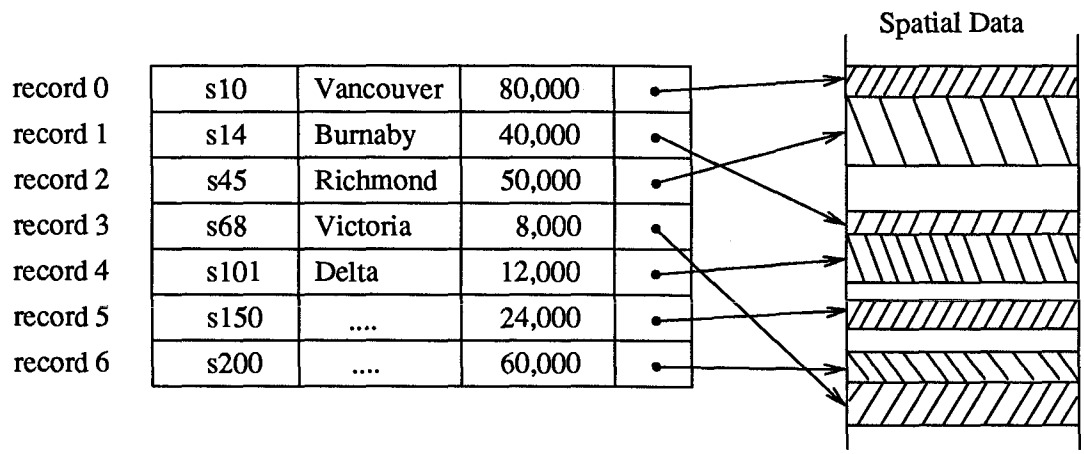

Figure 4.1 A file structure.

Access methods for secondary storage which allow efficient manipulations of a large volume of records are an essential part of a database management system. Traditionally, a relational database is indexed and searched according to its primary key or multi-keys. *B*-trees of one kind or another are the most common index structures for alphanumerical data which can be sorted in a certain one-dimensional order [BaM72]. We can still use these methods to index aspatial attributes in our extended relational databases as before.

However, spatial search according to an object's position is often required in spatial databases, and the classical database storage structures are not appropriate or sufficient for this

purpose. Therefore, new indexing mechanisms should be incorporated in order to explore the spatial neighborhood of objects and to answer spatial queries efficiently. In this chapter, we introduce an efficient data structure and indexing methods specific for the separately stored spatial attribute.

4.2. Spatial Index Methods

There have been many spatial access methods developed for the efficient storage and accessing of spatial objects. Most of the methods represent spatial objects by a large set of small rectangles which are stored in a hierarchical data structures [Sam88]. The minimum bounding rectangle (*MBR*) of an object, with its edges parallel with the axes of the data space, usually serves as a simple geometric key for region object indexes in the spatial databases. An important property of this approximation is that a complex object is abstracted by a limited number of bytes in the same way as the data space and its subspaces may be represented. If arbitrary bounding polygons are used [Jag90], the overlapped and redundant space could be reduced by increasing the storage space to record the divisions (bounding polygons) themselves in the index. However, some objects like highways, rivers which are represented in long line segments are very likely to be running parallel with the diagonals, such that corridor-like bounding polygons which may not be parallell with the axes are necessary in order to narrow the search space down to a smaller sub-space.

Spatial index methods can be summarized into the following four categories according to the principles which guide the decompositions. The first category is the quad-tree based regular recursive decomposition of space, which divides a region into equal-sized quadrants and a sub-region can be further divided into four parts till some condition is satisfied [Sam84].

The second category is the bucket methods or grid files, which include one- and multi-layered grid file structures [SiW88] and the *LSD* tree [HSW89]. The R-tree [Gut84] and its modified version, R^+-tree [SRF87], belong to the third category. Both of the first two categories partition all of the space, while the third one considers only the space that contains objects. The R-tree structure may contain overlapped sub-spaces while the R^+-tree resolves overlaps by increasing the height of the tree. These trees are natural extensions of B-trees for N-dimensional object indexes. The performances of the R^+-tree and the R-tree are analyzed in [FSR87] under the assumption of the line segments being uniformly distributed. Finally, the fourth category includes the transformation schemes that represent the *MBR*s as higher dimensional points which can then be organized by traditional accessing methods. For instance, a two dimensional rectangle is represented in a four dimensional point (U, V, W, Z) where U, V and W, Z are X and Y axes' minimum values and maximum values respectively. However, since the neighboring objects are usually not closely stored in such a structure, only a small group of queries which require the exact *MBR* searchings can be handled efficiently. It is also impossible to be used for arbitrary bounding polygon representations since it requires variable dimensional points.

4.3. Extending the Spatial Indexes for Spatiotemporal Data

In our discussion of spatiotemporal databases for geographic information, we assume that the spatial objects are in non-rectangular shapes. Therefore, we distinguish the abbreviation in index structure from the object data representation. We also assume that a sequence of images are stored in the database, each having a unique time stamp t_i, $i = 0, 1, ..., n$. For example, in Figure 4.2 and Figure 4.3, the spatial objects are represented as O_1, O_2, and O_1',

etc, and the same object O_4 has two representations at two different time t_0 and t_1, which are O_4 and O_4', respectively. The two representations, though differ in shape and/or position, denote the same spatial object.

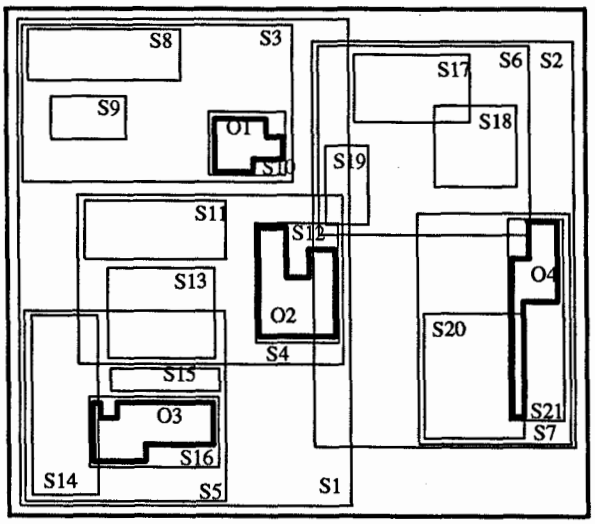

Figure 4.2. The image at time t_0.

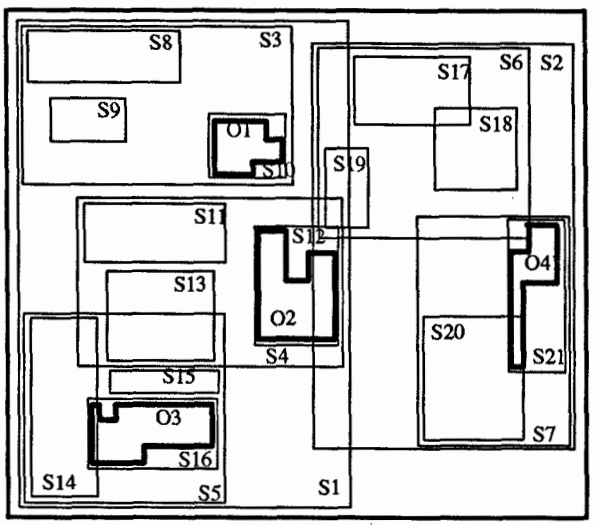

Figure 4.3. The image at time t_1.

To store a sequence of images with incremental time stamps, a simple method is to construct one separate spatial index structure for each image using the previously developed indexing techniques, such as the widely used R-tree and then associate with each image some

temporal information. For example, we can construct a sequence of dense R-trees with their roots stored in a sequential array in ascending time order or indexed in a tree structure. Therefore, search in such a database corresponds to spatial data accessing with different time stamps. However, in most cases, the background information remains stable between successive images, that is, there are only very few changes on the object locations and shapes. In such cases, instead of storing each complete image independently, data sharing should be explored for data storage and indexing.

4.3.1. Typical queries and searching primitives

We confine our discussion of query processing to temporal-relevant spatial queries because other kinds of queries can be processed by certain kinds of aspatial attribute searches. For example, to examine the geometric history of an object, the search is directed according to the object ID or its symbolic name which is usually indexed with the popualr B-tree technique. Thus its temporal/spatial data can be found without searching through spatial index trees.

Our discussion focuses on the following two kinds of search primitives:

(1) search for the objects within a rectangle R at time t_i; and

(2) search for the objects within a rectangle R from t_i to t_j.

Although there could be other kinds of search primitives, the above two primitives represent typical ones involving both spatial and temporal searches. Many other spatio-temporal primitives can be derived from them by changing the search conditions, such as, substituting "*overlapping*" for "*within*" or "*after* t_i" for "*from* t_i to t_n".

4.3.2. Multiple R-trees

The first improvement of indexing for the additional dimension *time* based on spatial access methods is a multiple R-tree structure, MR-tree. An MR-tree is a sequence of R-trees in which the first image at t_0 is stored in an R-tree, the image at t_i is constructed based on the R-tree of t_{i-1} by sharing their common subtrees.

Definition 4.1 Let $I = (i_0, i_1, ..., i_n)$ denote an image sequence derived at time $t_0, t_1, ..., t_n$, respectively, in our spatiotemporal database. An MR-tree is a collection of trees $R_0, R_1, ...$, R_n, where R_0 is an R-tree which is used to index the image i_0, and R_j, $1 \le j \le n$, is constructed from R_{j-1} by the Algorithm 4.1 to index i_j.

To facilitate the processing of queries involving a sequence of images, an MR-tree can be constructed. Assume an R-tree is of the order M, that is, each node has at least $M/2$ and at most M entries which are of the form (S, P), where S is a rectangle that covers all the rectangles of its descendents. The MBRs of the physical spatial objects are stored in the leaves of the tree. The first index tree R_0 is constructed using the typical R-tree insertion algorithm [Gut84]. R_i is constructed from R_{i-1} by rearranging only those objects at t_i which are different from those at t_{i-1}. That is, a sub-tree of R_i can be shared with that of R_{i-1} if both cover the same set of (unchanged) objects. However, when there are changes for certain objects, these leaf nodes and their corresponding ancestor nonleaf nodes must be updated accordingly. Some entries will be deleted and others inserted. The sketch of the algorithm is listed below. We use $E.S$ and $E.P$ to refer to the spatial coverage of an entry E, and the pointer to its child node, respectively.

Algorithm 4.1 Constructing tree R_k from tree R_{k-1}

Input : a copy of tree R_{k-1};
DELET[] contains a list of entries which are in the i_{k-1} but not in i_k;
LIST[] contains a list of entries which are only in i_k.

Output : tree R_k containing all the entries in LIST[].

```
BEGIN
  FOR each entry E in DELET DO
  BEGIN
    Find the leaf node which E belongs;
    Mark all the nodes on this path dirty;
    Remove E from the leaf;
    IF the leaf has less than M/2 entries THEN mark it underflow
  END
  FOR each entry E in LIST DO
  BEGIN
     current := root;
    WHILE current is not a leaf DO
    BEGIN
      IF an entry A in current whose rectangle covers E.S THEN
        current := A.P;
      ELSE
      BEGIN
        Choose an entry A from current such that A.S needs
        least enlargement to cover E.S comparing other entries
            current := A.P;
      END
         Mark current dirty;
    END
    IF current has space THEN /* reached a leaf */
      insert E ;
    ELSE
    BEGIN
       Put E and current to STACK ;
       Mark current overflow
    END
  END
  Change those pointers pointing to undirty nodes to corresponding nodes in R(i-1);
  Remove all the undirty nodes;
  Check underflow nodes and overflow nodes;
  Reorganize the tree to balance such that each node has at least M/2 entries and
    at most M entries with the principle of marking as less nodes dirty as possible;
END
```

In the above algorithm, the rebalance of the tree is delayed until all the deletions and insertions have been completed for the new image. The reorganization of the current tree is on the local nodes, that is, the nodes are marked dirty during construction, and other nodes

which are shared by the previous tree should not be changed.

Theorem 4.1 If each image i_j, $j \geq 0$, contains k objects, tree R_{j+1} can be constructed from the balanced tree R_j using Algorithm 4.1 in O(k log k) steps and R_{j+1} is a balanced tree.

Proof Sketch:

Let D and A be the numbers of entries to be deleted and inserted, respectively. D includes only the objects in R_j but not in R_{j+1} and the objects whose shapes are changed from R_j to R_{j+1}, A includes only the objects in R_{j+1} but not in R_j and the objects whose shapes are changed from R_j to R_{j+1}, and k is the total number of objects in each picture, so that $A + D \leq 2k$. Because R_0 is a balanced tree with k entries in its leaves, if we can prove that R_1 can be constructed from R_0 in time O(k log$k$) and R_1 is a balanced tree, the rest is also proved. Let h be the height of R_0, which is in the order of O($log_m k$). The first part of Algorithm 4.1 is to remove D entries from the copy of R_0, and removing each entry requires searching from the root to a leaf node and deleting the entry from the leaf. Thus $D \times h$ is the total steps for the first part, that is, O(D $\log_m k$). The second part is to insert A entries in the tree. Each insertion will start from the root and then go to a leaf node which costs h steps. The total steps for the second part is $A \times h$, that is, O(A $\log_m k$). The third part can be done by traversing the tree in the breadth first fashion to make the undirty path to be shared with R_0. At leaf level, since i_1 also contains k objects, the overflow and underflow nodes can be rearranged within themselves, such that the resulted tree R_1 still keeps the same structure as R_0. Because R_0 is a balanced tree, therefor, so is R_1. The total steps for the third part is in O(k/m). Therefore, the time complexity is the sum of the three parts, that is, $((A+D)\log_m k + k) \leq (2k \log_m k)$ which is in the order of O(k log$k$). □

Corollary 4.1 The time complexity of constructing the *MR* -tree to index an image sequence I using Algorithm 4.1 is $O(n \times k \log k)$.

Proof Sketch:

From Theorem 4.1 we know each image can be constructed using Algorithm 4.1 in $O(k \log k)$ steps. Totally, there are n such images, therefore, the *MR* -tree construction requires $O(n \times k \log k)$ steps. □

Theorem 4.2 Given a balanced *MR* -tree, if each i_j contains k objects, the time complexity of retrieving an object by its *MBR* and t_j is at least $O(log\ k)$.

Proof Sketch:

To search through the *MR* -tree, we first locate the root of R_j which indexes i_j. If a hash table is used, the root can be found in one step. Secondly we start from the root to check each entry in a node whether it overlaps with the *MBR* and contains t_j. For each overlapping entry, the search is going further to its child node and on until it reaches a leaf. If the object is indexed by the *MR* -tree, at least one path will exist from the root to a leaf, and its length is in $O(log_m k)$. Therefore, the conclusion holds. □

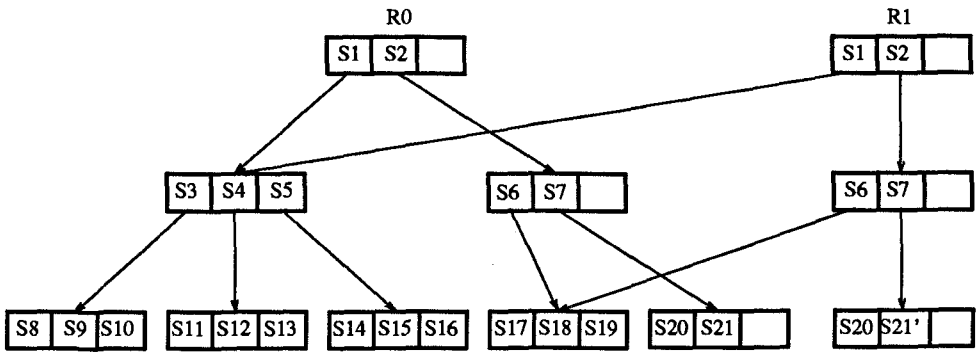

Figure 4.4. The corresponding *MR* -tree for Figure 4.2 and Figure 4.3.

For example, Figure 4.2 and Figure 4.3 show the images at t_0 and t_1, respectively. Notice that the only difference between the two images is the change in the size of the object O_4 (from O_4 to O_4'). Figure 4.4 illustrates the *MR* -tree for R_0 and R_1. Furthermore, if a new object emerges near O_4, or S_{20} increases, the corresponding tree will have very minor changes since the leaf node containing S_{21}' still has room for it.

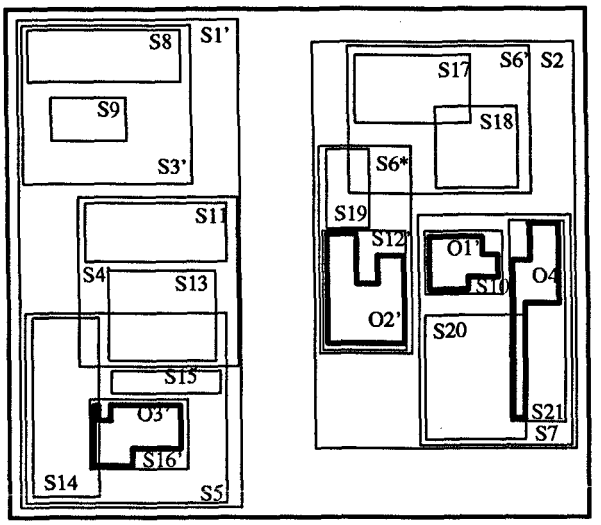

Figure 4.5. The image at time t_1 in the worst case.

We examine two extreme cases. First, if two images are exactly the same at time t_{i-1} and t_i, no new node is created and the same tree is assigned to both R_{i-1} and R_i. Secondly, even if there is only one out of M entries changed in each leaf node, none of the leaves and their ancestors can be shared and R_i must be a completely new tree. One such example is shown in Figure 4.5, in which, comparing with Figure 4.2, O_1 becomes O_1' and is covered by S_7 instead of S_3, O_2 is shifted to O_2' and stored in a different subtree, and O_3 is changed in size and becomes O_3'. Figure 4.6 shows the two trees, in which R_1 does not share any node with R_0.

Although it is possible to reorganize the previously established trees to make more portions shared among different images in the *MR* -tree, it is in general an NP-complete problem

to achieve the maximum memory utilization [Nis88]. Thus we will not further address this issue here.

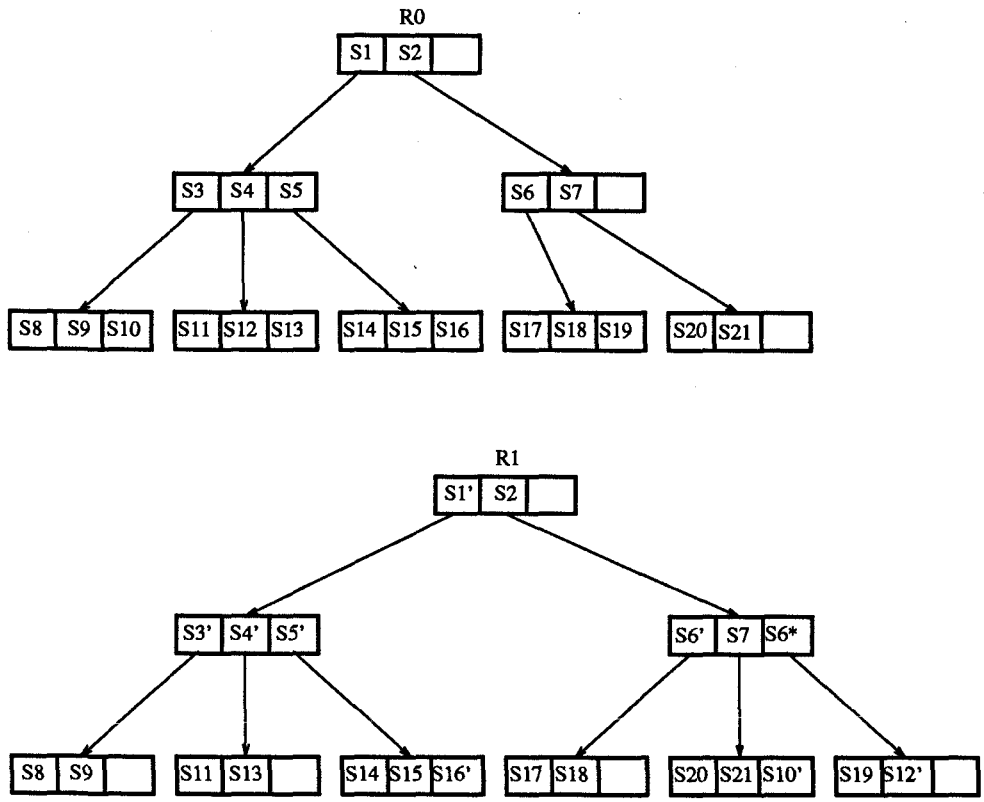

Figure 4.6. The corresponding *MR* -tree for Figure 4.2 and Figure 4.5.

4.3.3. The RT-Tree Index Structure

The cases discussed above can be handled more efficiently using another tree index structure, i.e. *RT* -tree. An *RT* -tree couples time intervals with the spatial ranges in each node of the tree so that only one index tree is maintained as opposed to the n index trees maintained by the *MR* -tree technique [Xu90]. It is also more natural using an *RT* -tree to reorganize index of the spatiotemporal attributes defined in our model when the data has already been represented as the basic spatiotemporal atoms in NF^2.

4.3.3.1. Definition of the RT-tree

An RT-tree is an improved R-tree. An RT-tree of the order M is a height-balanced tree with the index records in its leaf nodes containing entries (S, T, P), where P (pointer) points to the physical object, T represents time interval from time t_i to t_j when the object is at S, and S is the MBR of the object. All the leaf nodes are chained together to facilitate the sequential search. Each entry of a nonleaf node has the same format, (S, T, P), however, P points to a subtree whose leaves have records (S_i, T_i, P_i), such that each S_i is covered by S. Such S_i's may be overlapped among different images. Figure 4.7 shows the image which is the overlap of the images of the Figure 4.2 and Figure 4.5.

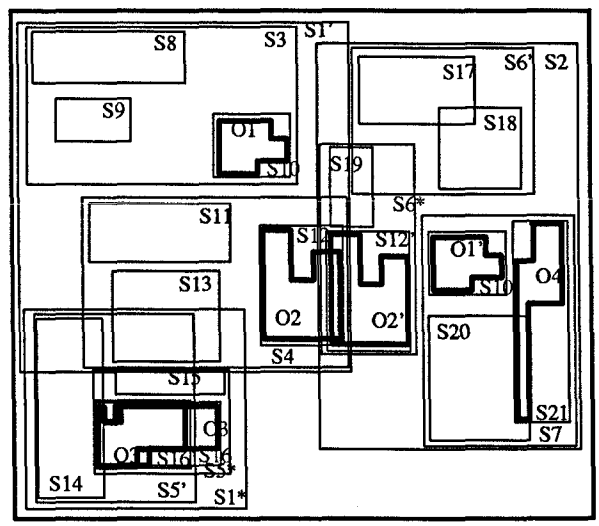

Figure 4.7. The overlap of the images shown in Figure 4.2 and Figure 4.5.

Definition 4.2 Let $h \geq 0$ be an integer, and m be a natural number, $M=2m$. An RT-tree satisfies the following conditions.

(1) Each path from the root to any leaf has the same length h; thus h is the height of the tree;

(2) Each node except the root and the leaves has at least m index record entries which point to its child nodes. The root is a leaf or has at least two entries;

(3) Each node has at most M entries;

(4) Non-leaf nodes contain index record entries of the form (S, T, P) where T is $[t_i, t_j]$, a time range, which means from time t_i to t_j, and S is a rectangle which covers all the

rectangles of the entries in the lower node pointed by P;

(5) Leaf nodes contain index record entries of the form (S , T , P) where T is the same as above, S is an *MBR* of an object, and P points to the spatial data of the object. Each leaf node has two extra entries which are used to chain its left neighboring node and right neighboring node in order to perform the sequential searches.

4.3.3.2. Construction of the RT-tree

Initially, an *RT*-tree is built in a way similar to the construction of an R-tree for the first image R_0. At time t_i when a new image R_i comes, it initiates a sequence of insertion operations. The insertion of a record (MBR_A , t_i , P) is performed as follows. First, it searches the leaf node to check whether there is an entry with the same *MBR* and the same data. If there is one, expand the time intervals of this entry; otherwise, check whether the current leaf node has free space for new entries. If it is full, it is split into two and the split process may propagate up to the root.

Algorithm 4.2 Insert An Entry Into An *RT*-tree

Input: An *RT*-tree with the root R and an entry E (MBR_A, t_i, P) to be inserted.

Output: A balanced *RT*-tree with the root in R which contains the information about (MBR_A, t_i, P) in a leaf node.

```
BEGIN
  SW := MBR_A , TW := [ t_i , t_i ]
  LEAF_NODE = FIND_LEAF (R, SW, TW ) ;
  IF an entry E in LEAF_NODE conatains MBR_A THEN
  /* this means the object may not change its shape from last image to the current one */
  BEGIN
    Expand E.T to E.T' which includes t_i ;
    RETURN (R) ;
  END
  ELSE IF LEAF_NODE has less than M entries THEN /* the leaf node is not full */
  BEGIN
    Place the entry (SW , TW , P) in LEAF_NODE ;
    RETURN (R) ;
  END /* insert an entry into a unfull leaf */
  ELSE /* LEAF_NODE is full, has M entries */
```

```
BEGIN
   Get a new node NEW_NODE ;
   Place the entry (SW , TW , P) into NEW_NODE ;
   Split LEAF_NODE into two parts and put one part in NEW_NODE ;
   /* this split should ensure both covering rectangles minimized,
     LEAF_NODE and NEW_NODE have M/2 and M/2 +1 entries respectively. */
   Set LEAF_NODE.S , LEAF_NODE.T , NEW_NODE.S , NEW_NODE.T ;
   /* covering rectangles and time range */
   Get the father node F_NODE of LEAF_NODE ;

   WHILE F_NODE is not the root R DO
   BEGIN
      IF F_NODE has less than M entries THEN
      BEGIN
        Place the entry NEW_NODE in F_NODE ;
        RETURN (R) ;
      END
      ELSE /* F_NODE is full */
        E := NEW_NODE ; MID_NODE := F_NODE ;
            Get a new node called NEW_NODE ;
        Place the entry (E.S , E_T , E_P) into NEW_NODE ;
        Split MID_NODE into two parts and put one part in NEW_NODE ;
        /* this split should ensure both covering rectangles minimized,
        MID_NODE and NEW_NODE have M/2 and M/2 +1 entries respectively. */
        Set MID_NODE.S , MID_NODE.T , NEW_NODE.S , NEW_NODE.T ;
        /* covering rectangles and time range */
        Get the father node F_NODE of MID_NODE ;
   END
   IF F_NODE is the root R THEN
   BEGIN
      Get a new node and assign it to R ;
      Put the entry (LEAF_NODE.S , LEAF_NODE.T , LEAF_NODE) to R ;
      Put the entry (NEW_NODE.S , NEW_NODE_T , NEW_NODE) to R ;
      RETURN (R) ;
   END
  END /* insert an entry into a full leaf node */
END /* end of INSERT */
```

Procedure FIND_LEAF (R , SW , TW)

Input: An *RT*-tree with the root *R* and a search window (*SW* ,*TW*)
where *SW* is the spatial rectangle, and *TW* is the time period.

Output: The leaf nodes in the form of (*S* ,*T* ,*P*)
where *E.S* and *E.T* overlap the search window *SW* and *TW* ,

respectively.

```
BEGIN
   initialize STACK ;
   current := R ;
   IF current is a leaf node THEN RETURN (current) ;
   IF current is not a leaf node THEN
   BEGIN /* choose one entry that needs least expansion to enclose SW and TW */
      TEMP_S := 0 ; TEMP_T := 0 ;
         FOR each entry E in current node DO
         BEGIN
         enlarge E.S to E.(S+M) which encloses SW ;
         enlarge E.T to E.(T+N) which encloses TW ;
         IF M and N are smaller than TEMP_S and TEMP_T THEN
         BEGIN
             TEMP_S := M ; TEMP_T := N ; TEMP_P := E.P ;
         END
      END /* FOR chosen the entry that needs least expansion */
      call FIND_LEAF(TEMP_P, SW , TW) ;
   END /* nonleaf */
END /* FIND_LEAF procedure */
```

Lemma 4.1 Given an *RT*-tree, if it has N entries in its leaf nodes, the average time complexity of an insertion operation, to insert a new entry into the tree, is in the order of O($log\ N$).

Proof Sketch:

If an *RT*-tree has N entries in its leaves, the height H of the tree is then in the order of O($log_m N$) which is similar to a *B*-tree analysis. The first part of an insertion is to find a leaf which may either contain the entry or should hold the entry. This operation starts from the root to a leaf node, so that it visits H nodes on the path. If the leaf has already contained the entry, do nothing. Otherwise, if the leaf has room for another entry, then insert the entry. If the leaf is full, splitting might propagate to the root in the worst case where H nodes on the path to the root wll be visited. Averaging the three cases, the time complexity, therefore, is in the order of O($log\ N$). □

Theorem 4.3 The worst case time complexity of constructing an *RT*-tree for the image sequence I, each i_j containing k objects, $0 \leq j \leq n$, using Algorithm 4.2, is in the order of $O(n \times k \ log(n \times k))$.

Proof Sketch:

From Lemma 4.1, to insert one entry to an *RT*-tree reqires $O(log\ N)$ time when the tree has N entries. If all the objects are changing from one image to another in the worst case, the total number of entries to be inserted to the *RT*-tree is $n \times k$. Therefore, the time complexity of constructing the *RT*-tree for I is at worst $\log 1 + \log 2 + \cdots + \log k + \cdots + \log n \times k = (log(n \times k)!)$, i.e., $O(n \times k \ log(n \times k))$. □

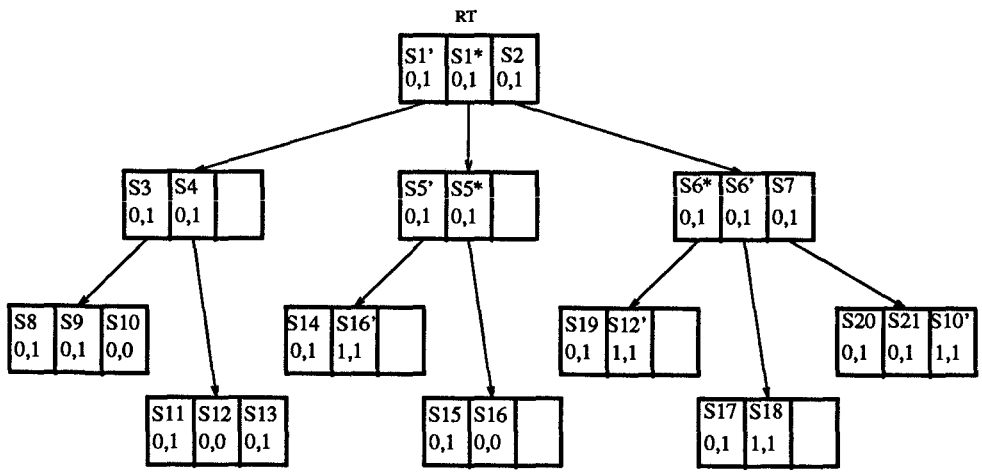

Figure 4.8. The corresponding *RT*-tree of Figure 4.7.

Notice that to make the spatial search more efficient, the selection of a nonleaf node under which the entry is to be inserted should be based on the minimal time interval and/or the least spatial coverage. Also, the covering rectangles and the time intervals of such nonleaf nodes should be updated during the insertion when expension is required. The total number of nodes, therefore, is reduced by sharing the old partitions as recorded in nonleaf nodes. That is, an *RT*-tree needs much fewer nodes than its corresponding *MR*-tree because it does not

create duplicate paths. The price paid here is that the height of an *RT*-tree may be a little higher than the corresponding *MR*-tree (since it stores some more entries than those in one image of the *MR*-tree), and each node needs some more space to store the time interval. Figure 4.8 shows the constructed *RT*-tree for Figure 4.7, which contains many overlapped partitions for images at t_0 and t_1 (represented by Figure 4.2 and Figure 4.5). Notice that the *RT*-tree (Figure 4.8) has 11 nodes while the corresponding *MR*-tree (Figure 4.6) has 17 nodes.

4.3.3.3. Retrieval of the RT-tree

The retrieval algorithm descends the tree from the root in a manner similar to an *R*-tree. More than one subtree under a node visited may need to be searched, thus it is not possible to gurrantee the worst case performance.

Algorithm 4.3 *RT*-tree Retrieval Within a Spatiotemporal Window

Input: An *RT*-tree with the root *R* and a search window (*SW* , *TW*) where *SW* is the spatial rectangle, and *TW* the is time period.

Output: All index record entries *E*s in the form of (*S* , *T* , *P*) where *E.S* and *E.T* overlap the search window *SW* and *TW*, respectively.

```
BEGIN
  found := false ;
  IF R is not a leaf node THEN
  BEGIN
    FOR each entry E in R
      IF E.S overlaps SW and E.T overlaps TW THEN
      BEGIN R := E.P ; CALL RETRIEVE ; END
    END /* end of for */
  END /* IF for nonleaf node */
  ELSE /* R is a leaf node */
  BEGIN
    FOR each entry E in R
      IF E.S overlaps SW and E.T overlaps TW THEN
      BEGIN
        found := true ; RETURN (E.S , E.T , E.P );
      END
```

```
    END /* end of for */
     IF found == false THEN RETURN ( NONE );
  END /* leaf node */
END /* RETRIEVE */
```

Theorem 4.4 Given an *RT*-tree, if *I* contains *N* different entries, the best case time complexity of retrieving an object by its *MBR* and *t* is O(*log N*).

Proof Sketch:

To search on the *RT*-tree, we start from the root to check each entry in a node whether it overlaps with the *MBR* and contains *t*. For each overlapping entry, the search goes further to its child node and on until it reaches a leaf. If the object is indexed by the *RT*-tree, at least one path will exist from the root to a leaf, and its length is in O(*log N*). Therefore, the conclusion holds. □

4.3.3.4. Node Splitting Strategies

Similar to a *B*-tree or an *R*-tree, a node split operation should be performed on an *RT*-tree node if a node is full but a new entry has to be inserted. Node splitting is a process of reorganizing the information for a node, which should group related information closely together to faciliate accessing. Since there could be many versions in one or a group of spatial objects which may differ in time, location, shape and semantics, there should be different criteria for node splitting. It is often a conflicting goal to achieve both the minimal coverage of space and the minimal coverage of time in a node splitting. We suggest three node splitting preferences which could be used independentely or combinedly to form a more sophisticated strategy.

(1) *Spatial coverage preference*: Split a node based on the minimal spatial covering rectangles. Suppose a node consists of two portions, each with a minimized spatial covering rectangle. Node split based on spatial coverage will facilitate the searches based on spatial criteria, however, it may not benefit the searches based on certain temporal criteria because the objects stored in each leaf node could belong to different images which represent different time intervals. As a result, a node covers a long time interval.

(2) *Temporal coverage preference*: Split a node based on the minimal time interval. Suppose $[t_i, t_j]$ is the time interval of a node, in which some entries have time interval $[t_i, t_k]$ and others $[t_h, t_j]$ where t_h may be less than t_k (some overlapping). Spatially, these entries could be located in the same covering rectangle. In the worst case, an entry could be in both the time intervals because it has not been changed from time t_i to t_j. This preference may facilitate searches based on certain time criteria. However, such a preference may lead to the same result as the *MR*-tree in regards to space cost if each leaf node covers one time instance only.

(3) *Semantic coverage preference*: Split a node based on the semantic knowledge about the images. A node can be split according to the meanings of different image segments. Objects in the same regions or with certain features may be grouped together or closely. This facilitates efficient data search for the frequently used semantic-oriented queries. For example, if a node contains both B.C. and Alberta (two neighboring provinces of Canada), node split could be based on the boundary line of the two provinces. A similar node splitting criteria could be based on the semantics related to time intervals. For example, if the comparison is always within a decade, the node should be split into two, in which one contains the objects in 1970's while the other contains those in 1980's.

4.4. Comparisons between the MR-Tree and the RT-Tree

We compare the performance of the two index structures, *MR*-tree and *RT*-tree, based on the storage space utilization and the processing efficiency of typical spatiotemporal queries.

4.4.1. Space Cost Analysis and Comparison

To simplify our discussion, we assume that for an *RT*-tree or an *MR*-tree of the order M, each node contains at least m $(=M/2)$ entries and at most M. Notice that in most real databases, a node is usually not full to avoid frequent tree reorganization caused by data insertions. Moreover, we assume that each image contains k objects among which there are on average x objects changed from t_j to t_{j+1} where $0 \le j < n$. That is, on average, there are $(k-x)$ objects of image j remaining the same from image j to image j+1, while there are x objects with locations or shapes changed. An entry which contains a pointer to its data is added for each changed object so that at least $N = k + nx$ entries should be in the leaf nodes to accomodate the $(n+1)$ images. To analyze the memory cost of the two tree structures, we have the following theorems.

	Best Case of RT-tree			**Worst Case of RT-tree**	
Height	Num_of_node	Num_of_entry	Height	Num_of_node	Num_of_entry
1	1	M	1	1	1
2	M	M^2	2	2	$2m$
3	M^2	M^3	3	$2m$	$2m^2$
4	M^3	M^4	4	$2m^2$	$2m^3$
...	...	...	...	...	...
h	M^{h-1}	M^h	h	$2m^{h-2}$	$2m^{h-1}$

Table 4.1 Storage space of an RT-tree: the best case vs. the worst case

Theorem 4.5 The space requirement of an *RT*-tree is from $\frac{N-1}{M-1}$ to $1+2\frac{N-2}{M-2}$, i.e., in the

order of $O(N/M)$.

Proof Sketch:

Table 1 presents the numbers of nodes and entries at each level of an *RT*-tree, similar to that of a *B*-tree [KoS86]. M^h and $2m^{h-1}$ are the best and the worst case number of entries at the bottom level (leaves) which should be able to store at least N nodes. Thus h must be greater than $\log_m N$ and less than $\log_m N - \log_m 2 + 1$, that is, in the order of $O(log_m N)$. Therefore, the number of nodes in the *RT*-tree is from $\frac{N-1}{M-1}$ $(=1+M+M^2+\cdots+M^h-1)$ to $1+2\frac{N-2}{M-2}$ $(=1+2+2m+2m^2+\cdots+2m^h-2)$, which is in the order of $O(N/M)$. □

Notice that the size of the *RT*-tree is determined by the number of the objects indexed. In constrast, the size of an *MR*-tree is dependent upon not only the number of the objects indexed but also the locality of changes.

Theorem 4.6 For the number of nodes of the *MR*-tree, the worst case is $\frac{nk}{M}+\frac{n}{M}$; the best case is $\frac{k}{M}+\frac{2xn}{M}$.

Proof Sketch:

The tree for image at t_0 is a complete *R*-tree whose number of nodes should be in the order of $O(k/M)$ and whose height should be in the order of $O(log_M k)$. The number of nodes in the remaining trees is dependent on the order of insertions and concrete data. However, it is not difficult to study two extreme cases (the best and the worst cases). The worst case happens when the updates are uniformly distributed in the search area as a new image is inserted,

while the best case happens when the changes are clustered in a few nodes. Let L be the number of the leaf nodes in R_0. In the worst case, x varying objects in the current image are scattered evenly corresponding to the objects in l $(=\min(x, L))$ leaf nodes of the previous tree. Thus l new leaves plus their ancestor nonleaf nodes up to the root should be created for the new index tree. If $x \geq L$, the tree is a complete R-tree where no subtrees are shared with the first one. Thus the total number of nodes of an MR-tree is about $\frac{nk}{M} + \frac{n}{M}$, where $\frac{n}{M}$ is the nodes for n roots, which is much worse than the RT-tree memory ultilization. In the best case, the previous versions of all the changed objects happen to be stored in the neighbouring leaves such that other (unaffected) nodes can be shared, Since each such tree (from the second on) requires at least $\frac{x}{M} + \frac{x}{M^2} + \frac{x}{M^3} + \cdots + 1 \approx \frac{2x}{M}$ nodes, the total number of nodes of the MR-tree in the best case is $\frac{k}{M} + \frac{2xn}{M}$. □

Corollary 4.2 In the best case, the memory ultilization of the MR-tree is close to that of a corresponding RT-tree; in the worst case, the memory ultilization of the MR-tree is much worse than the RT-tree.

4.4.2. Time Cost Analysis and Comparison

Suppose the size of each node is one page. Then the time cost is the total number of pages accessed (the total number of nodes visited) in the retrieval of the inquired spatial objects. An RT-tree search starts at the root, following the entry (S, T, P) of a nonleaf node where S locates the search rectangle and T is a time t_i or a time interval $[t_i, t_j]$ until a leaf is reached. The cost of the search time depends on the height of the RT-tree and the overlap-

pings of nonleaf nodes. The spatial and temporal overlapping of nodes, depending on the distribution and sizes of objects, is hard to decide. In the following analysis, we assume the *RT*-tree and the *MR*-tree have the same overlappings in their nodes so that the heights of the two trees are the major factor of time complexity.

We examine the worst case for an *RT* tree. Suppose the *RT*-tree stores the objects each of which is changed from one image to another. Thus the total number of leaf entries to be stored should be $n \times k$, which requires an *RT*-tree of height $\log_m n \times k = \log_m n + \log_m k$, where m is the number of entries in each node. Since $\log_m n$ is less than 1 if the number of images is less than the number of entries in a node, the *RT*-tree should have alomst the same height as an R_j in the corresponding *MR*-tree. In an average case when the probability of an object changing is p where $0 < p < 1$, the total leaf entries in an *RT*-tree should be $p \times n \times k$. Hence the height of the *RT*-tree is defined by the number of different entries.

An *MR*-tree search starts by finding the root R_j of a subtree using the specified t and then searches for the spatial objects. If the roots, R_j, $0 \le j \le n$, of the subtrees, are organized into a balanced tree structure, the *MR* will be the root of subtrees, R_j, $0 \le j \le n$. From *MR* to an R_j, the height is $O(log_m n)$ if the tree is in the order of m. The height of an R_j, $0 \le j \le n$, is $O(log_m k)$ from the Theorem 4.2. Totally, the *MR*-tree has the height $O(\log_m n + \log_m k)$. The height of an *MR*-tree, different from that of an *RT*-tree, is purely defined by the number of objects in each image. Even when few things change, the number of trees with the same height is the same instead of being decreased. If a retrieval has a time interval from t_y to t_z, then there are $(z - y + 1)$ trees (i.e., R_j, for $j = y, y + 1, ..., z$, in the *MR*-tree). The cost of accesses should be $(z - y + 1)$ multiplied by the height of the *MR*-tree. In such a case, the *MR*-tree is much worse than the *RT*-tree in performance. Only if a query is about a specific

time instance, then only one tree R_j is visited.

Corollary 4.3 In some cases, when few objects change and most objects are stable, the *RT*-tree is a good choice. Instead, when most objects in an image changes from time to time and queries are about a whole space at specific time instances instead of time intervals, the use of the *MR*-tree will speed up the search process.

4.5. Spatial Data Structures

As we stated in the previous sections, geographical data is usually irregular in shapes which require variable length structures to store. Broadly, two types of representations to describe the positional extent of spatial objects are vector-based using coordinates and raster-based using grid cells. In vector format, a polygon is described by a series of lines or points, a line is also described by a series of points and a point is described either by absolute coodinates or by relative coordinates. Raster format using scan lines or image pixels is usually used for region object representation in terms of a regular grid cell or variable-sized cells. Arguments have been made in favor of either the raster or the vector format [Bur86]. However, particular formats and format conversions are not our major interest, rather we are more concerned with how these representations are related to database storage structures.

In our extended relational database, the spatial data is grouped into objects and each object has different versions so that its storage is different from organizing a thematic map or an image as one unit, especially for raster representations. In the consequent sections we discuss the data structures used for representing a spatial attribute of one object.

4.5.1. Data Structure Using Raster Representation

One of the typical raster representations is the quadtree structure which is variable resolution arrays that allow a region to be split up into parts, or to contain holes without difficulty. Quadtrees are mostly used to represent binary images or region data. As shown in Figure 4.9, a region is regularly decomposed into quadrants and a subspace may be further divided into four equal parts until a quadrant is either completely contained in an object region or completely outside [Bur86]. The smallest quadrant may be the single pixel. Each node of the tree could be represented as a record with six fields, of which four fields are pointers to the sons, one is a pointer to the father and the last encodes the color, i.e., black, white, or grey, of the node. The grey nodes are internal nodes. The region then is composed of those black nodes.

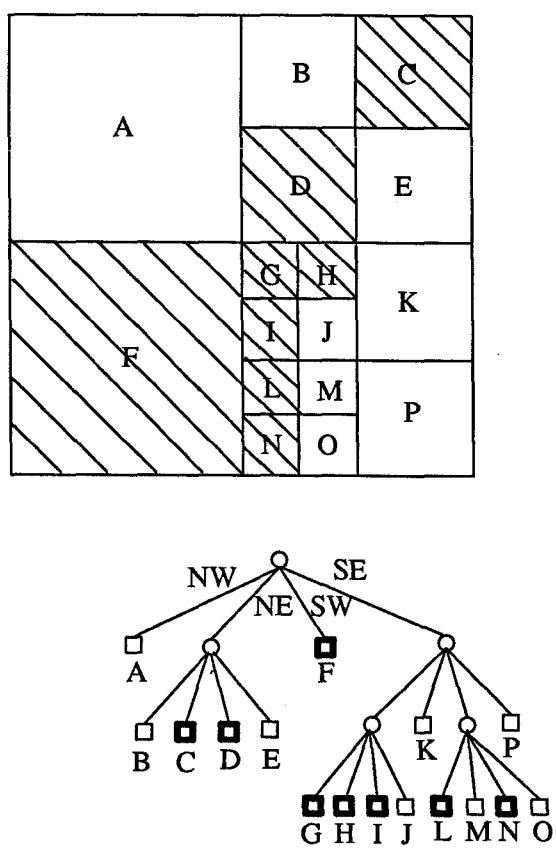

Figure 4.9 A simple region on a rasterized map and the corresponing quadtree.

A thematic map or an image may contain more than one object. For example in Figure 4.10, a map is composed of four objects, namely O_1, O_2, O_3, and O_4. The index structure such as the *RT*-tree developed in last section can be used for indexing the four objects where each leaf node contains a pointer to the object's data set. In raster based representation, one object region has two parts. One is the position of the object in the map and the other is the relative region data itself. As an example, for object O_3 in figure 4.10 , its *MBR* upleft coner is located at (8,8) in the map and its data is orgnized into a quadtree corresponding to the tree in Figure 4.9.

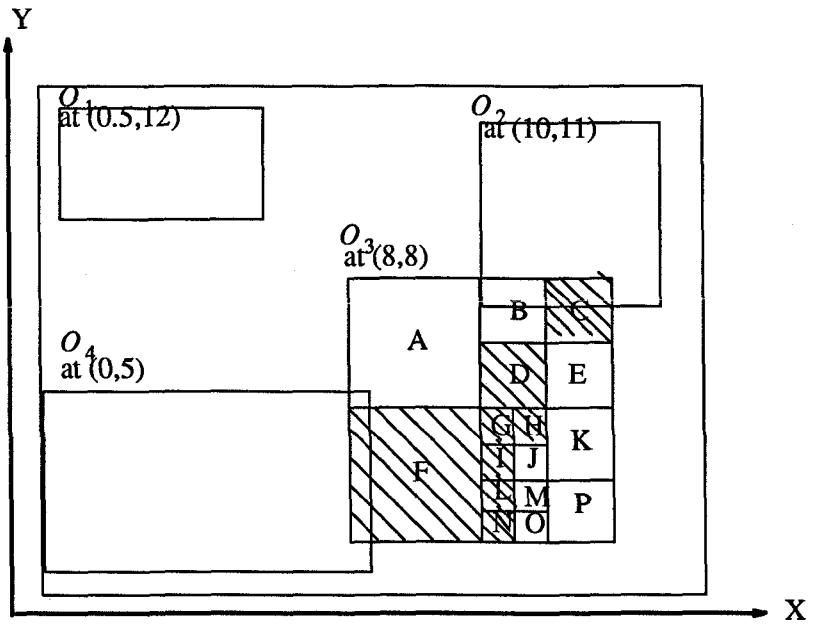

Figure 4.10 A rasterized map.

When object O_4 increases at another time, say, at t_1, as Figure 4.11 illustrates, the corresponding quadtree is constructed similarly. Notice that the two trees in Figure 4.9 and Figure 4.11 have much in common in this case such that sharing at storage level can be achieved by sharing common quadrants. Figure 4.12 demonstrates the possiblity. For a moving object, say O_3 shifted from (8,8) to (9,6), as Figure 4.13 shows, it only needs to record its *MBR* position and the second part can be completely shared with its previous version.

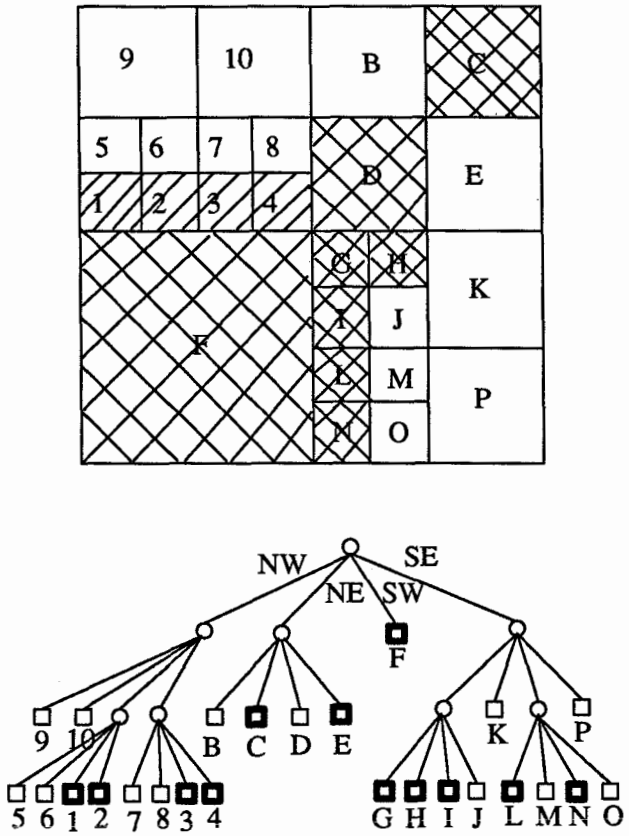

Figure 4.11 The region at t_1 and the corresponding quadtree.

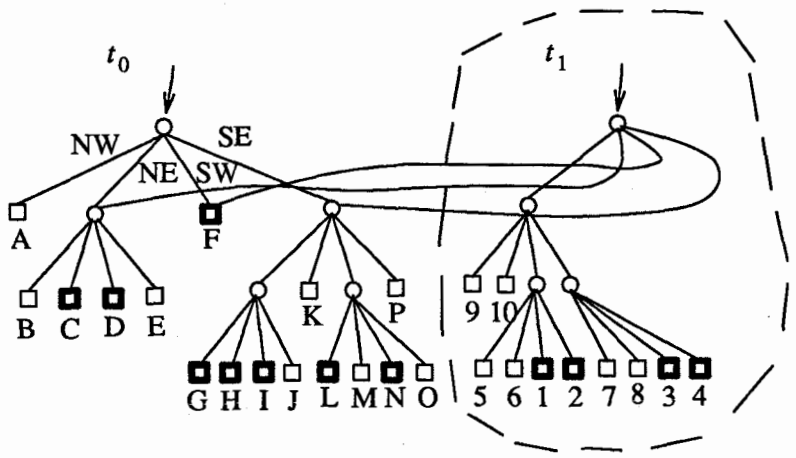

Figure 4.12 Two quadtrees.

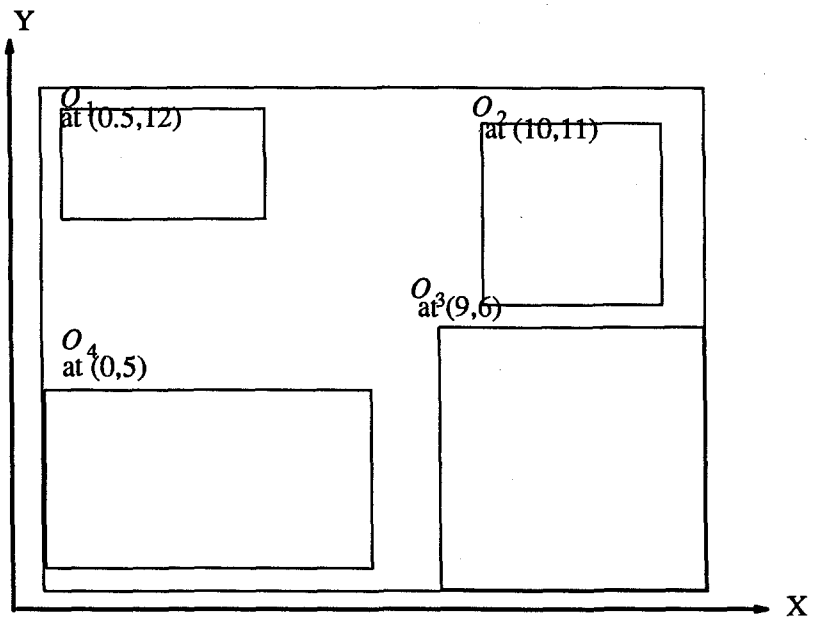

Figure 4.13 The map at time t_1.

Such arrangement clusters all the versions of one object together. This clustering can produce significant performance in processing queries on an object history.

4.5.2. Data Structure Using Vector Representation

Vector representation of an object is an attempt to represent the object as exactly as possible. The coordinate space is assumed to be continuous, not quantized as with raster space, allowing all positions, lengths, and dimensions to be defined precisely. For example, the same object O_1 in Figure 4.9 can be represented more compressedly and precisely as illustrated in Figure 4.14. One typical and simple vector representation is that a region is represented by polygons each of which is then represented as a series of points. More sophisticated methods have been suggested in order to avoid redundancy and to detect complex or nested polygons. Using a vector representation method, all the objects (polygons) in an image at one time instance are easily clustered together by network linkages [Bur86]. Queries about relation-

ships among objects in one plane benifit from the clustering. However, queries about an object history may be costly processed since the versions of an object are stored further apart. That is, more disk accesses are required in order to get the necessary information about an object.

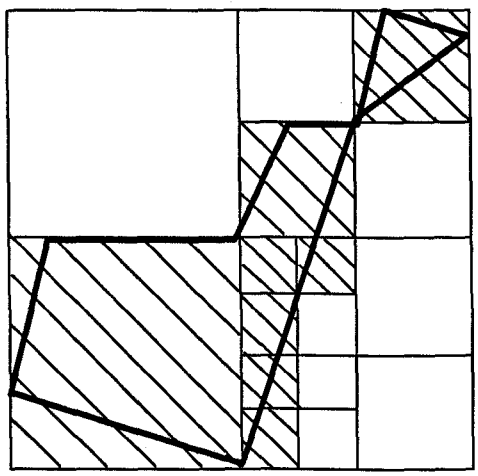

Figure 4.14 Represent an object as a polygon.

In our spatiotemporal databases, again, there are relative and absolute coordinates to represent a polygon. Relative representation is good when objects do not change their shapes or geometric definations, but only change their topological relations, such as moving objects. The polygon representation differs from the quadtree method, in that the former is hard to achieve the sharability among the several versions of an object; while the later may be less precise or space consuming.

CHAPTER 5

CONCLUSIONS

5.1. Contributions of This Thesis

In this thesis we have proposed a spatiotemporal database management system (SDBMS) based on an extended relational data model. At representation level, a spatiotemporal relation differs from a conventional relation in several ways. First we have extended the attribute domain to include such basic data types as *OID*, *POINT*, *LINE*, and *REGION* in relations. The basic construct of an attribute is the atom which consists of a temporal component and a value from an attribute domain, where the temporal component specifies the duration of the value. Objects are arranged in NF^2 in order to provide users a clear view of objects and life spans of their various attributes. Moreover, relational algebraic operations have been defined on our database model. We have also discussed the interface and extended the retrieval statement of SQL query language to query on our databases.

At physical level, we have studied indexing structures, multiple *R*-trees and *RT*-tree, for spatiotemporal domains based on the conceptual and complexity analysis. An *RT*-tree incorporates temporal information with spatial objects in index nodes, which represents an elegant merge of multiple *R*-trees with different time-stamps. Therefore, it saves storage space and improves the performance.

5.2. Future Research

The implementation of the whole spatiotemporal database management system concerns query parsing, optimization, buffer management, and concurrency control which should be studied based on our extended relational model. Specifically, efficient algorithms for spatial functions should be studied based on computational geometry theory and practice. Moreover, the performance of the *RT*-tree in experimental spatiotemporal databases and its comparison with other index structures are also interesting topics for our future study since these index methods are also applicable to other databases such as object-oriented spatial database systems or other kinds of geographic information systems.

APPENDIX

SYNTAX FOR STSQL

This syntax covers the examples used in Chapter 3. The meta-symbols used are '::=' and '|' and '{' and '}'. The brackets are used to enclose optional items.

STSQL Constructs:

SELECT column1, ..., column n
FROM tablename, ... , tablename
WHERE conditions
WHEN conditions

SELECT UNARY_FUNCTION(column k), BINARY_FUNCTION(column i, column j) , ...
FROM tablename, ... , tablename
WHERE conditions
WHEN conditions

SELECT column1,..., column n
FROM tablename, ... , tablename
WHERE conditions
AND column i (comparison or spatial predicate)
(subquery)
WHEN column j (comparison or spatial predicate)
(subquery)

STSQL Syntax for Retrieval:

<query>::= <a-query> {<time-spec>} {<con-spec>}
<a-query>::= SELECT <att-spec> FROM <relations>

<att-spec>::= <attributes> | <functions>
<attributes>::= <attr> | <attr>, <attributes>
<attr>::= <geo-attr> | <a-attr>
<geo-attr>::= <POINT> | <EXT>
<EXT>::= <LINE> | <REGION>
<REGION>::= <AREA> | <PGON>
<a-attr>::= <INT> | <REAL> | <STRING>
<functions>::= <function> | <function>, <functions>

<function>::= <aggregate> (<attr>) |
<unary-fun> |
<binary-fun> |
<high-fun> |
<temp-fun>
<aggregate>::= <AVERAGE> | <COUNT> | <MINIMUM> | <MAXIMUM> |
<NEAREST> | <FURTHEST> | <SUM>
<unary-fun>::= <LENGTH> (<LINE>) |
<DIAMETER> (<REGION>) |
<PERIMETER> (<REGION>) |
<AREA> (<REGION>) |
<CENTER> (<REGION>)

<binary-fun>::= <INTERSECTION> (<EXT> , <EXT>) |
<DISTANCE> (<POINT> , <POINT>) |
<UNION> (<REGION> , <REGION>) |
<DIFFERENCE> (<REGION , <REGION>) |

<temp-fun>::= <TIME> {(<attr>)}
<high-fun>::= <MOVING_DIR> (<geo-attr>) |
<MOVING_SPEED> (<geo-attr>) |
<CHANGING_RATE> (<geo-attr>) |
<INCREASED (<REGION>) |
<REDUCED (<REGION>)

<t_dom>::= <attr>.TIME | <t_value>
<t_value>::= <t_point> | <t_range> | (<t_value>, <t_value>) |
expression with +, - operations on <t_value>
<t_point>::= PRESENT | time value in chronons
<t_range>::= (<t_point>, <t_point>)
<t_unit>::= YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

<relations>::= <relation> | <relation>, <relations>
<relation>::= existing table name

<time-spec>::= AT <t_point> |
IN <t_point> |
DURING <t_point>, <t_point> |
AFTER <t_point> |
BEFORE <t_point> |
SINCE <t_point> |
WHEN conditions

<t_point>::= PRESENT | time value in Chronons
<t-conditions>::= <t-cond> | <t-cond> AND/OR <t-conditions>
<t-cond>::= <t_dom> comp <t_dom> | <t_dom> <time-spec>

<con-spec>::= WHERE <conditions>

<conditions>::= <a-condition> | <s-condition> AND/OR <s-conditions>
<a-condition>::= <a-attr> <comp> <attr> |
<a-attr> <comp> <query> |
<function> <comp> <attr> |
<function> <comp> <function> |
<geo-attr> <s-predicate> <geo-attr> |
<h-predicate> (<geo-attr>)

<s-predicate>:= IS_INSIDE |
INTERSECTS |
IS_NEIGHBOR_OF |
IS_COVERED_BY |
NOT_COVERED_BY |
IS_NORTH_OF |
...
WITHIN

<h-predicate>::= <IS_MOVED> (<geo-attr>)> |
<IS_INCREASED> (<REGION>) |
<IS_REDUCED> (<REGION>) |
<IS_CHANGED> (<REGION>)

<comp>::= < | = | > | <= | >= | !=
<range>::= (<num-value>, <num-value>)
<num-value>::= INT_value | REAL_value
<num>::= <num-value> | <geo-num>

REFERENCES

[Ari86] G. Ariav, A Temporally Oriented Data Model, *ACM Transactions on Database Syastems*, Vol. 11, No. 4, December 1986, 499-527.

[BaA90] A. Basu and R. Ahad, An SQL-based Query Language For Networks of Relations, *SIGMOD Record*, Vol. 19, No.1, Mach 1990.

[BLW88] D. S. Batory, T. Y. Leung and T. E. Wise, Implementation Concepts for an Extensible Data Model and Data Languagee, *ACM Transactions on Database Systems*, Vol. 13, No. 3, September 1988, 231-262.

[BaM72] R. Bayer and E. McCreight, Organization and Maintainance of Large Ordered Indexes, *Acta Informatica*, Vol. 1, No. 3, 1972, 137-189.

[Bou85] P. Boursler, Image Data Bases: A Status Report, *Proceedings of IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Miami Beach, Florida, November 18-20, 1985, 359-366.

[Bur86] P. A. Burrough, Principles of Geographical Information System for Land Resources Assessment, in *Monographs on Soil and Resources Survey*, P. H. T. Beckett (ed.), Oxford Science Publications, No. 12, 1986.

[CDR86] M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, Object and File Management in the EXODUS Extensible Database System, *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, August, 1986, 91-100.

[CDF86] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson and E. Shekita, Architecture of EXODUS Extensible DBMS, *Proceedings of International Workshop of Object-Oriented Database System*, Asilomar, CA, 1986, 52-65.

[ClM90] T. Y. Cliff and R. R. Muntz, Query Processing Temporal Databases, *Proceedings of Sixth Internal Conference on Data Engineering*, Los Angeles, California, USA, Feb. 5-9, 1990, 200-208.

[ClW83] J. Clifford and D. S. Warren, Formal Semantics for Time in Databases, *ACM Transactions on Database Systems*, Vol. 8, No. 2, June 1983, 214-254.

[Cod70] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, *Communication ACM*, 13, 1970, 377-387.

[Cod79] E. F. Codd, Extending the Relational Database Model to Capture More Meaning, *ACM Transactions on Database Systems* , Vol. 4, No. 4, December 1979, 397-434.

[CoH79] J. Cohen and T. Hichey, Two Algorithms for Detertermining Volumes of Convex Polyhedra, *Journal of ACM*, Vol. 26, No. 3, July 1979, 401-414.

[Dit86] K. R. Dittrich, Object-Oriented Database Systems: The Notion and the Issues, *Proceedings of the International Workshop on Object-Oriented Database Systems*, California, September 1986, 2-6.

[EgF88] M. J. Egenhofer and A. U. Frank, Towards a Spatial Query Language User Interface Considerations, *Proceedings of the 14th VLDB Conference*, Los Angeles, California, 1988, 124-133.

[FSR87] C. Faloutsos, T. Sellis and N. Roussopoulos, Analysis of Object-Oriented Spatial Access Methods, *Proceedings of 1987 ACM-SIGMOD Conference on Management of Data*, San Fransisco, CA, May 1987, 426-439.

[FeP86] M. Feuchtwanger and T. K. Poiker, Infological and Datalogical Spatial Models, *Proceedings of the workshop on Digital Mapping and Land Information*, University of Calgary, Calgary, Alberta, Canada, April 1986.

[GaV85] S. K. Gadia and J. H. Vaishnav, A Query Language for a Homogenious Temporal Database, *ACM Symposiumm on Principle of Database Systems*, New York, 1985, 51-56.

[GCK89] G. Gardarin, J. P. Cheiney, G. Kiernan, D. Pastre and H. Stora, Managing Complex Objects in an Extensible Relational DBMS, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989, 55-65.

[Gut89] R. H. Guting, Gral: An Extensible Relational Database System for Geometric Applications, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989, 33-44.

[Gut84] A. Guttman, R-Tree: A Dynamic Index structure for Spatial Searching, *Proceedings of 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984, 47-57.

[HaO88] A. Hafez and G. Ozsoyoglu, The Partial Normalized Storage Model of Nested Relation, *Proceedings of the 14th VLDB Conference*, Los Angeles, California, 1988.

[HSW89] A. Henrich, H. Six and P. Widmayer, The LSD tree: spatial access to multidimensional point and non-point objects, *Proceedings of the Fifteenth International Conference on Very large Data Bases*, Amsterdam, 1989, 45-53.

[Jag90] H. V. Jagadish, Spatial Search with Polyhedra, *Proceedings of Sixth Internal Conference on Data Engineering*, Los Angeles, California, USA, Feb. 5-9, 1990, 311-319.

[JoC88] T. Joseph and A. F. Cardenas, PICQUERY: A High Level Query Language for Pictorial Database Management, *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, May 1988, 675-681.

[KeW87] A. Kemper and M. Wallrath, An Analysis of Geometric Modeling in Database Systems, *ACM Computing Surveys*, Vol. 19, No. 1, March 1987, 47-91.

[KoS86] H. F. Korth and A. Silberschatz, Database System Concepts, *McGraw-Hill*, 1986.

[LaC88] G. Langran and N. R. Chrisman, A Framework for Temporal Geographic Information, *CARTOGRAPHICA*, Vol. 25, No. 3, 1988, 1-14.

[Mit89] B. Mitschang, Extending the Relational Algebra to Capture Complex Objects, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989, 297-305.

[Nis88] N. Nishimura, Complexity Issues In Tree-Based Version Control, *Technical Report 212/88*, Department of Computer Science, University of Toronto, June 1988.

[RKS88] M. A. Roth, H. F. Korth and A. Silberschatz, Extended Algebra and Calculus for Nested Relational Databases, *ACM Transactions on Database Systems*, Vol. 13, No. 4, December 1988, 389-417.

[RoF88] N. Roussopoulos and C. Faloutsos, An Efficient Pictorial Database System for PSQL, *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, May 1988, 675-681.

[Sam84] H. Samet, The Quadtree and Related Hierarchical Data Structures, *ACM Computing Survey*, Vol. 16, No. 2, June 1984.

[Sam88] H. Samet, Hierarchical Representations of Collections of Small Rectangles, *ACM Computing Survey*, Vol. 20, No. 4, December 1988, 271-309.

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos, The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects, *Proceedings of the 13th VLDB Conference*, Brighton, England, 1987, 3-11.

[SiW88] H. Six and P. Widmayer, Spatial Searching in Geometric Databases, *Proceedings of IEEE 4th International Conference on Data Engineering*, Los Angeles, CA, February 1988, 496-503.

[Sno87] R. Snodgrass, The Temporal Query Language TQuel, *ACM Transactions on Database Systems* , Vol.12, No. 2, June 1987, 247-298.

[TaG89] A. U. Tansel and L. Garnett, Nested Historical Relations, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, Vol. 18, No. 2, June 1989, 284-293.

[TsZ84] S. Tsur and C. Zaniolo, An Implementation of GEM — Suporting a Semantic Data model on a Relational Back-End, *Proceedings of 1984 ACM-SIGMOD Conference on Management of Data*, May 1984.

[Xu90] X. Xu and J. Han, RT-tree: The Index Structure for Spatiotemporal Databases, *The 4th International Symposium on Spatial Data Handling*, Zurich, Switzerland, July 1990.