AN EXPERIMENTAL SPATIAL INFORMATION SYSTEM

by

PRASHANT DINKAR VAIDYA

Thesis submitted to the Graduate Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
Computer Science and Applications

APPROVED:

Dr. L.G.Shapiro

Dr. R.M.Haralick

Dr. D.G.Kafura

January, 1982
Blacksburg, Virginia

ACKNOWLEDGEMENT

I am extremely grateful to Dr. Linda Shapiro whose guidance, assistance, and friendship made this project an informative and enjoyable experience. I would also like to acknowledge Dr. Robert Haralick for his useful and relevant comments, suggestions, and help. My sincere thanks to Gary Minden from the University of Kansas, for having designed the initial structures and the query language interpreter for this system. A discussion with him helped me understand the complex structure of the interpreter.

I would also like to address my thanks to Bary Lam, Swapan Engineer, and Steve Choquette with whom I worked. I am grateful to all the people from the Computer Science department and the Spatial Data Analysis Lab, particularly Glen Fowler, Prasanna Mulgaonkar, and Scot Krusemark who were always willing to help and open for suggestions. I would also like to acknowledge all my friends in Blacksburg whom I have grown to know and care for. They made my stay in Blacksburg a time which I will remember forever.

Finally, a word about my family and friends in India and in the U.S.A. to whom I owe the most important debt of gratitude. Although, thousands of miles apart, I could always feel their love and affection towards me. I could never

have accomplished such an important achievement in life without their constant encouragement. I dedicate this thesis to them.

TABLE OF CONTENTS

ACKNOWLEDGEMENT ii

Chapter page

I. INTRODUCTION 1

THE OUTLINE OF THE THESIS 2
DIFFERENCES BETWEEN ALPHANUMERIC AND CARTOGRAPHIC DATA 3
REPRESENTATIONS OF CARTOGRAPHIC DATA 5
THE SURVEY OF GEOGRAPHIC INFORMATION SYSTEMS . 6
THE CANADIAN GEOGRAPHIC INFORMATION SYSTEM (CGIS) 6
THE U.S. CENSUS'S DIME SYSTEM 7
THE POLYVRT SYSTEM 7
HARVARD'S GEOGRAF SYSTEM 8
WEBER'S APPROACH 9
THE GEO-DATA ANALYSIS AND DISPLAY SYSTEM . . 10
THE GEO-QUEL SYSTEM 11
SOME OTHER SYSTEMS 12
A SPATIAL DATA STRUCTURE 14

II. THE LOGICAL DATABASE STRUCTURE 19

ANALYSIS OF DATA 20
Stream Data 20
Road Data 21
PROTOTYPES 22
REGION 22
A/V REGION 23
SUBREGION ADJACENCY 23
STREAM NETWORK 24
LABELS 24
WATER STREAMS 24
STREAM 26
A/V STREAM 26
INTERSECTING STREAMS 26
ROAD NETWORK 28
ROAD 28
A/V ROAD 28
INTERSECTING ROADS 29
LABEL 29
POLYGON 31
CHAIN 31
A/V CHAIN 31
POINTS 32
POINT 32

III. DATA STRUCTURES 34

POINTERS 35
RDS_PTR 35
RELATION_PTR 35
RELATION_CELL_PTR 35
TREE_CELL_PTR 36
LIST_CELL_PTR 36
POINT_CELL_PTR 36
CHAR_STRING_PTR 36
STACK_ELEM 36
DATA TYPES FOR SPATIAL DATA STRUCTURES 37
RDS 40
RELATION LIST 41
MULTI-DIMENSIONAL RELATIONS 41
RDS 43
TREE RELATIONS 44
LIST RELATIONS 48
ATOMS 50
CHARACTER STRING 50
POINT 51
ARRAYS 51
RDS_ARRAY 51
REL_ARRAY 53
STACK_ARRAY 53
NTUPLE 53
TREE_POSITION 54
NAME_ARRAY 54
INT_ARRAY 54
DICTIONARIES 55
RDS_DICTIONARY 55
REL_DICTIONARY 56

IV. THE PHYSICAL DATABASE STRUCTURE 59

THE PHYSICAL STRUCTURE IN MEMORY 60
SPATIAL DATA STRUCTURE FOR A REGION 60
SPATIAL DATA STRUCTURE FOR A WATER STREAM . 63
SPATIAL DATA STRUCTURE FOR A STREAM 63
SPATIAL DATA STRUCTURE ROAD NETWORK 70
SPATIAL DATA STRUCTURE FOR A ROAD 70
SPATIAL DATA STRUCTURE FOR A LABEL 73
SPATIAL DATA STRUCTURE FOR A POLYGON 75
SPATIAL DATA STRUCTURE FOR A CHAIN 75
IMPLEMENTATION OF INTERNAL PHYSICAL STRUCTURE . 79
THE PHYSICAL STRUCTURE ON THE SECONDARY DEVICE 85

V. THE QUERY LANGUAGE INTERPRETER 92

THE DATA STRUCTURES FOR THE INTERPRETER 95

STACK_ELEM 96
INTEGERS 96
BOOLEAN 98
REALS 98
VOCABULARY_ARRAY 98
STACK 103
THE INTERPRETER ALGORITHM 103
THE INTERPRETER PRIMITIVES 105
STACK MANIPULATION PRIMITIVES 107
+ 107
- 108
* 108
/ 109
NOT 109
I, J, K, and L 110
RELATIONAL OPERATORS 110
RDS?, REL?, INT?, PROC?, CHAR? and DICT? 111
-NIL? 112
SWAP 113
DROP 113
DUP 114
ROT 114
-ROT 115
OVER 115
#DROP 116
#ROT 116
#-ROT 117

.STACK 118
CR (CARRIAGE RETURN) 119
(STACK) 119
PROGRAM CONTROL PRIMITIVES 120
IF 121
ELSE 121
THEN 122
DO 122
+LOOP 124
REPEAT 124
UNTIL 125
VOCABULARY MANIPULATION PRIMITIVES 126
DEFINITION 126
END_DEF 127
CONSTANT 127
VARIABLE 128
FETCH 129
STORE 129
ALLOCATE 130
FORGET 130
#VOCB 131
DATABASE MANIPULATION PRIMITIVES 131

ALLOC_RDS 132
ALLOC_REL 132
RDS_INDEX 133
REL_INDEX 134
(CATALOG) 135
(REL_CATALOG) 135
#CATALOG 136
#REL_CATALOG 136
FIND 137
LIST_RDS and LIST_REL 138
XLIST_RDS and XLIST_REL 139
REL_ATTACH 139
NT_ATTACH 140
INRELA? 140
[NAME] 141
[TYPE] 142
[DIMEN] 142
[LENGTH] 143
[USE_CNT] 143
[STRUCT] 143
[LINK] 145
[DATA] 146
RDS_DICT 146
REL_DICT 147
MISCELLANEOUS COMMANDS 147
DB_LOAD 148
DB_UNLOAD 148
INPUT> 149
***EOF 150
>OUTPUT 150
DUMP 151
" 151
GETWORD 152
DONE 152
EXAMPLES 153
Load the Database. 154
Get a Relation from the Dictionary 154
Create a New Relation 156
Attach a Relation to a Spatial Data Structure 157
Define a New Command. 158
Traverse a Relation 161

VI. THE MEMORY MANAGEMENT 167

THE ARRAYS REL_IN_CORE AND RDS_IN_CORE . . . 167
COMPUTATION OF SPACE 168
READ RELATIONS AND SPATIAL DATA STRUCTURES INTO MEMORY 170
READING A RELATION INTO MEMORY 170

READING A SPATIAL DATA STRUCTURE INTO MEMORY 172
TRANSFER OF RELATIONS AND SPATIAL DATA STRUCTURES TO THE DISK 174
TRANSFERING A RELATION TO THE DISK 175
TRANSFERRING A SPATIAL DATA STRUCTURE TO THE DISK 175
DELETE RELATIONS AND SPATIAL DATA STRUCTURES FROM MEMORY 176
DELETING A RELATION FROM MEMORY 176
DELETING A SPATIAL DATA STRUCTURE FROM MEMORY 177
THE PAGING ALGORITHM 179

VII. PERFORMANCE MEASUREMENT 185

INITIAL SET UP OF THE DATABASE 185
SET UP THE DATABASE IN MEMORY 185
TRANSFER THE DATABASE TO THE DISK 186
INDIVIDUAL PRIMITIVE OPERATIONS 187
THE OVERHEAD 187
LOAD THE DATABSE IN MEMORY 189
CREATE A RELATION OR SDS HEADER 189
SEARCH THE DICTIONARIES FOR THE GIVEN NAME 190
GET THE INDEX OF A RELATION OR SDS 191
ATTACH A RELATION TO AN SDS 192
ATTACH AN N-TUPLE TO A RELATION 193
EVALUATION OF THE PERFORMANCE OF THE SYSTEM . 193
FIND THE CPU TIME PER PAGE FAULT 206

VIII. CONCLUSIONS AND FUTURE WORK 208

BIBLIOGRAPHY 211

Appendix page

A. A 214

LIST OF FIGURES

Figure		Page
1.1	A spatial data structure representing the ..... state of Virginia	18
2.1	Prototype REGION and its relations	25
2.2	Prototypes WATER STREAMS, STREAM and their .... relations	27
2.3	Prototypes ROAD NETWORK, ROAD and their .... relations	30
2.4	Prototypes LABEL, POLYGON, CHAIN and atom POINT	33
3.1	STACK_ELEM using Pascal convention	38
3.2	Spatial data structure header RDS and .... RELATION CELL with their Pascal declarations	42
3.3	Relation header RDS, TREE_CELL and LIST_CELL ..	45
3.4	N-tuples stored in a TREE structure	47
3.5	N-tuples stored in a LIST structure	49
3.6	Data structures CHAR_STRING_NODE and POINT	52
3.7	RDS_ DICTIONARY and REL_DICTIONARY	57
4.1	Spatial data structure REGION_X and its .... relations	64
4.2	Spatial data structure WATER_STREAMS_I and	67

its relations ....

4.3 Spatial data structure STREAM_I and its 69

relations ....

4.4 Spatial data structure ROAD_NETWORK and its ... 72

relations ....

4.5 Spatial data structure ROAD_I and its relations 74

4.6 Spatial data structure LABEL_I and its 77

relations ....

4.7 Spatial data structure POLYGON_X and its 78

relations ....

4.8 Spatial data structure CHAIN_I and its 80

relations.

5.1 STACK_ELEM extended 97

5.2 Entry for command 'FIND' 100

5.3 Vocabulary entry for user defined command 102

'TEST' ....

5.4 Physical structure of relation A/V_REGION_X ... 162

6.1 The paging algorithm using Pascal-like 182

conventions ....

Chapter I

INTRODUCTION

The basic kinds of data found in maps are points, lines, and areas. This data can be stored in many forms; grid cells, list of points, list of line segments, and polygons are the common examples. The best storage representation is the one that can be accessed most efficiently for a given application.

Several geographic information systems have been in use for some time. Some of the well known systems include the Canadian Geographic Information System, the U.S. Census's DIME files, and the POLYVRT system. These systems all represent geographic data as polygons; the exact data structure used varies from system to system.

The thesis describes the implementation of yet another experimental geographic information system. It uses a formal organizational structure, called a spatial data structure, proposed by Shapiro and Haralick [SHAPL79], as its basic building block.

A spatial data structure is a set of relations used to represent a geographic entity. The components of the relations of this set may themselves be spatial data structures. This is a rich and flexible structure that can be used to

represent any kind of spatial information from high level entities such as states or cities to low level entities such as points, lines or pixels. The spatial data structure is a relational structure, and hence this system shares many characteristics of relational database systems. In particular, all of the primitive operations used in relational database systems are applicable to the relations of the spatial data structure.

1.1 THE OUTLINE OF THE THESIS

In this thesis we first discuss various representations of the cartographic data and briefly summarize several systems that are already in use. Next we define the spatial data structure and explain how geographic entities can be represented using the spatial data structures. The rest of the thesis describes the spatial information software system which has been implemented using the concept of spatial data structures.

In the description of the system we first develop the prototypes or the logical structure of the database. Next we explain the inner level data structures which are used to represent these spatial relationships. The physical data base structure, implentation details, setting up of the database, and the interpreter through which the database can

be accessed are described next. We then discuss how paging (transfering relations and spatial data structures back and forth between memory and disk) is accomplished. Finally we implement several test algorithms on which the performance of the system can be measured.

1.2 DIFFERENCES BETWEEN ALPHANUMERIC AND CARTOGRAPHIC DATA

Man has been using maps for representing land, water and sky for many years. With the help of maps he can obtain various information such as the location of a distant city, the boundary of a state, the path of a river, etc. Now automation is being employed in the processing of cartographic data to achieve cost and time efficiency. The automation of several applications, such as census work, require a computer representation of the cartographic data for processing.

Computer representation of the continuous two-dimensional features on a map is complicated by the spatial properties not found in typical alphanumeric data. For alphanumeric data there is a natural linear sequencing of alphanumeric characters over a finite range of values [HAGAP80]. The operations to be performed on the alphanumeric data are usually well defined. The range of the user's interest in an item, such as name or age, is clear to the human user. The scope of the user's interest in an item, such as last name

only, is usually specific. The maintenance of an item in an alphanumeric database is not necessarily dependent on other items in the database.

In contrast, cartographic data has no single-dimensional sorting order that defines the positional relation between all items in a map. Two items are related by radial position, but they are difficult to adequately relate to a third item. In addition to the descriptions of the features portrayed such as county names, city populations, and road surface types, there are certain spatial relationships such as distance, direction, above, below, overlap, through, adjacency, continuity, neighborhood, nearness, separates, surrounds, etc., which must be taken into consideration. The human senses spatial position, distance, and relative importance of features when looking at a map; subconsciously throwing away nonrelevant information. However, it is difficult to incorporate these spatial relationships in a computer model of the cartographic data, and requires considerable computational support. It is therefore necessary that the data structures representing the cartographic data should retain the spatial relationships between them.

1.3 REPRESENTATIONS OF CARTOGRAPHIC DATA

Most geographic information systems have been organized either as a set of polygons or as a raster of grid cells. The raster representation of the spatial data is a variation of a matrix technique, and is mainly used in image processing applications. In the raster representation of geographic data, a regular grid is placed over a map, and certain properties such as population, major crop, or land type are recorded for each grid cell. The raster representation retains the spatial relationships of the map, but the efficiency is sacrificed for some operations. Another disadvantage is that the locational accuracy is not inherrent in the size of the grid chosen.

The polygon method defines a geographic area in terms of the digital coordinates of its boundary. The polygon representation is suitable for explicitly representing region boundaries and line data such as rivers or roads. It allows a higher degree of locational accuracy. However, initial preparation and digitization of the polygon outlines are more expensive. Also, the processing data structures for polygons are more complex than the simple array structures which can be used for grids.

1.4 THE SURVEY OF GEOGRAPHIC INFORMATION SYSTEMS

In this section we briefly discuss various existing geographic information systems which are based on the polygon or raster representation of the cartographic data.

1.4.1 THE CANADIAN GEOGRAPHIC INFORMATION SYSTEM (CGIS)

The Canadian Geographic Information System (CGIS) [SWITW75] is one of the earlier and successful geographic information system. This system includes features such as a command language, an assessment language, the possibility of overlaying maps, interactive graphics, and input from a drum scanner or an X-Y digitizer. In this system, regions are represented by polygons. Two types of files are used: the image data set which contains the line segments that defines the polygons and the descriptive data set which contains the user assigned identifiers, centroids, and area for each polygon. In the image data set, a line segment points to its left and right polygons and to the next two boundary chains that continue bounding the polygons on the left and on the right. Each polygon in the image data set points to its representation in the descriptive data set and vice versa.

1.4.2 THE U.S. CENSUS'S DIME SYSTEM

In the late 1960's the U.S. Bureau of the Census developed the Dual Independent Map Encoding (DIME) concept [USBUR70] to digitize and edit city street maps. In this system, the basic element is a line segment. Each line segment is defined by two end nodes plus pointers to the polygon on the right side and the polygon on the left side of the segment. The data structure was kept simple at the expense of extra processing time for certain operations. For instance, determining what line segments share a node or finding the whole outline of a polygon requires searching the database. The structure is adequate to represent topological spatial relations between regions.

1.4.3 THE POLYVRT SYSTEM

The POLYVRT system [LABCG74] is similar to the DIME system described above. In the POLYVRT data structure, the polygons are formed by chains. A chain is a sequence of X, Y coordinates which outline the shape of the chain. A chain has two end points called nodes. A chain also has a polygon to its left and to its right associated with it as in the DIME system.

In the data structure, a polygon entry points to an ordered list of chains which compose its outline. The list of

chains for the polygon points to the chain table entry for each chain. The chain table contains a pointer to the list of coordinates forming the chain's outline, the identification of the nodes at each end of the chain, and the identification of the polygons to the segment's left and right. This feature separated the detail of the chain from its relationships with nodes and polygons. The polygon is thus established as a separate entity linked to the chains which composed it. This allowed easy maintenance and manipulation of the chains. In POLYVRT searching can take place in two directions; from chain to polygon and from polygon to chain.

1.4.4 HARVARD'S GEOGRAF SYSTEM

GEOGRAF is a system proposed by Peucher and Chrisman [PEUCT75] to handle both planar data and surface data. The system includes the concepts of (i) a least common geographic unit (an area that can not be partitioned further), (ii) a chain group (a set of chains forming a boundary of two regional units of a given polygon class), and (iii) an attribute cross-reference table. To handle surface data the system has a two part database including both a triangle data structure and a set of points that lie along lines of high information content.

The triangle structure [GOLDC76] is a low level structure used for representing surface data. A triangle represents a piece of a three dimensional surface. The vertices of the triangle are nodes containing information such as elevation and slope. An interpolation may be used with a homogeneous coordinate system based on the three nodes' sample values to obtain information about points of the surface interior to a triangle. This structure is comparable to the image data set of the CGIS.

The points in the second set consist of peaks, pits, and passes. This se corresponds to the descriptive data set of the CGIS to some extent. The GEOGRAF system is a milestone toward unification of geographic data structures [CHANS??].

All the systems described so far in this section store geographic data in vector form.

1.4.5 WEBER'S APPROACH

The advantages and drawbacks of the grid formatted data structure are almost perfectly complementary to those of the topological or polygonal data structure. Therefore Weber [WEBEW78] has proposed a combination of locational data structure and grid formatted data structure. The operations working on this data structure are defined in a hierarchical

manner, so that the transition from grid formatted to lineal (polygonal) representation is to be considered merely as the last step in a process of successive refinement defined by a nesting of squares of different sizes. Weber claims that his locational data structure is well suited for the purpose of automated cartography. However, there are no defined operations to go through this data structure and no explanation of how to store the lineal representation and the grid formatted representation.

1.4.6 THE GEO-DATA ANALYSIS AND DISPLAY SYSTEM

IBM's Geo-data Analysis and Display System (GADS) [CARLE74] is the first documented geographic database system which used the approach of relational database [CODDE70]. It is the most ambitious of the geographic information systems in terms of flexibility and interactive problem solving capability. This system has database management facilities and supports database integrity and different user views of picture database.

It combines data from different large data files to provide the user with a small combined set of data in a tabular form. The geographic data is made available to the user in the form of an 'event' database and an 'extracted' database. Geographical interpretation of the event database is accom-

plished by reference to a corresponding polygon map file. This polygon map file provides a framework or index for relating data from the event database, in terms of their spatial relationships, and thereby enabling the development of the extracted database. The extracted data can be viewed on a CRT and manipulated with a relational query language with geographic commands such as display, outline, and shade added. The user interactively tries different solutions to solve an unstructured problem, saving interim solutions as desired.

Summarizing, GADS extracts data from large databases to form a small set of polygonal features in a relational data structure. It uses the power of the query language to help users solve unstructured problems.

1.4.7 THE GEO-QUEL SYSTEM

The GEO-QUEL system developed at U.C. Berkeley is another system which uses relational database approach to manipulate georaphic data [GOA75]. The basic entity in GEO-QUEL is a map, which is a collection of points, lines, line groups (polygons) and zones (collections of polygons). A map is stored in the INGRES database system as a 9-ary relation. A query language QUEL which is similar to SEQUEL [CHAMD74]. is used to interrogate the system. The system can handle

simple queries about a map and can display information fom a map. Only Codd's conventional relational operators are included; there are no picture operators defined to handle retrieval and manipulation of pictorial entities.

Modeleski [MODEM77] proposed additional relational attributes to permit topological manipulation of geographic files. He emphasized the storage of chain files in relational structure, topological access, and the use of topological information to enhance GEO-QUEL's recognition of graph-theoretic properties of a geographic file stored in a relational form. Because of its relational nature, this system is related to the system discussed in this thesis.

1.4.8 SOME OTHER SYSTEMS

In 1978, Wagle [WAGLS78] developed a data structure based on the relational table to handle river and river basin drainage information. Using the United States Geological Servey (USGS) quad sheets as source material, he developed a system to model or compute the outline of a river, the area of a river drainage basin, the mileage distance of specific points, such as dam sites or state boundaries, from the river's mouth, and the elevation of a river at a given distance from the river's mouth.

Edwards, Durfee, and Coleman [EDWAR77] used a hierarchical polygonal data structure to represent regions that can have holes, which in turn can have holes of their own, to any level of nesting.

Hagan [HAGAP80] developed and analyzed a logical data model for cartographic features built from nodes, segments, and polygons using the owner-member concepts of the CODASYL specification. Her structure has two levels. The higher level contains the features such as lakes, roads, and towns. The lower level contains the detailed outline and position of each feature in a network of nodes, segments, and polygons. The bridge between the two levels is composition. The features are composed of the elements in the network. The network is homogeneous in the sense that the three entities relate to each other in consistent ways, that is each element of the network is dependent on the others for its existence. This ensures consistency between the features in the higher level. This two level structure has the advantage of allowing interfeature relationships to be computed at query time and network paths to be computed using direct links instead of searches. However, her analysis shows the need for a relational DBMS and CODASYL network DBMS in at least one application.

The geographic information system described in this thesis is based on a unified relational database approach suggested by Shapiro and Haralick [SHAPL79]. We start our discussion with the definition of a <u>spatial data structure</u> which is the building block of the system.

1.5 <u>A SPATIAL DATA STRUCTURE</u>

In this section we define a general spatial data structure that can be used to represent any spatial information or relational data .

An <u>atom</u> is a unit of data that will not be further broken down. Integers and characters are common examples of atoms. An <u>attribute-value table</u> A/V is a set of pairs A/V = { (a,v) | a is an attribute and v is the value associated with attribute a }. Both a and v may be atoms or more complex structures. For example, in an attribute-value table associated with a structure representing a person, the attribute AGE would have a numeric value and the attribute MOTHER might have as its value a structure representing another person.

Note : The material of this section was taken directly from [SHAPL79].

A <u>spatial data structure</u> D is a set $D = \{R1,\ldots,RK\}$ of relations. Each relation Rk has a dimension Nk and a sequence of domain sets $S(1,k),\ldots,S(Nk,k)$. That is for each $k = 1,\ldots,K$, $Rk \subseteq S(1,k) \times \ldots \times S(Nk,k)$. The elements of the domain sets may be atoms or spatial data structures. Since the spatial data structure is defined in terms of relations whose elements may themselves be spatial data structures, we call it a recursive structure. This indicates 1) that the spatial data structure is defined with a recursive definition and 2) that it will often be possible to describe operations on the structure by simple recursive algorithms.

A spatial data structure represents a geographic entity. The entity might be as simple as a point or as complex as a whole map. An entity has global properties, component parts and related geographic entities. Each spatial data structure will have one distinguished binary relation containing the global properties of the entity that the structure represents. The distinguished relation is an attribute-value table and will generally be referred to as the A/V relation. When a geographic entity is made up of parts, we may need to know how the parts are organized. Or, we may wish to store a list of other geographic entities that are in a particular relation to the one we are describing. Such a list is just a

unary relation, and the interrelationships among the parts are n-ary relations.

Let us take an example to show how the geographic entities on any map can be represented using spatial data structures. Consider the state of Virginia.

We represent the state of Virginia by a spatial data structure. In this case, the A/V relation would contain global attributes of the state such as population, area, boundary, major crop, and so on. The values of most of these attributes (population, area, major crop) are atoms. The value of the boundary attribute is a spatial data structure defining the boundary.

One obvious division of the state is into counties. A list of counties could be included as one of the relations, or it might be more valuable to store the counties in a region adjacency relation, a binary relation associating each region (county) with every other region (county) that neighbors it. Counties, of course, would also be represented by spatial data structures.

Some other geographic entities that are related to a state are its highways, railroads, lakes, rivers, and mountains. Some of these entities will be wholly contained in

the state and others will cross its boundaries. One way to represent this phenomenon is to use a binary relation where the first element of each pair is a geographic entity and the second element is a code indicating whether the entity is wholly contained in the state. The spatial data structures representing the geographic entities themselves would contain more specific information about their locations.

Figure 1.1 illustrates a simplified spatial data structure containing an attribute-value table, a county adjacency relation, and a lakes relation for the state of Virginia.

The remaining chapters of this thesis describe an experimental geographic information system which has been designed and implemented using the concept of spatial data structures described in this section.

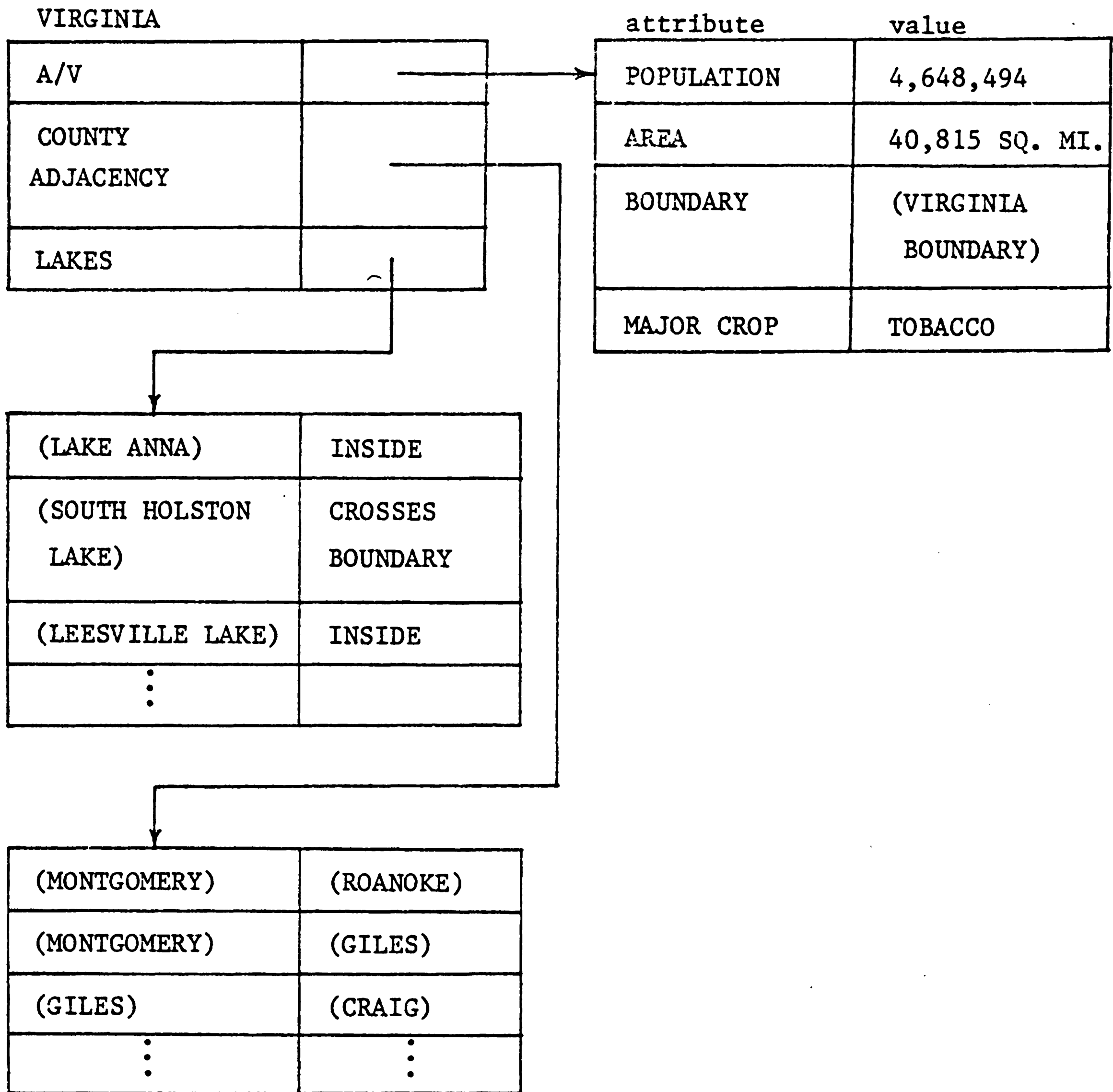


Figure 1.1 : A spatial data structure representing the state of Virginia.

Chapter II

THE LOGICAL DATABASE STRUCTURE

One objective of a database system is to systematize the access to the data elements. The first step in the implementation of any database management system is the design of its conceptual model (also known as a data model). The conceptual model is a representation of the entire information content of the database, in a form that is somewhat abstract in comparison with the way in which the data is physically stored [DATEC77]. In order to translate a model into an operational system, the model has to be described in a form which lends itself to implementation. Such a description is called a <u>schema</u> [WIEDG77]. In order to achieve data independence, these definitions must not involve any considerations of storage structure or access strategy - they must be definitions of information content only. A schema hence defines the logical structure of the database without any storage/access details.

In this chapter we develop the schema or the logical database structure for our geographic database system. The spatial data structure defined in Chapter II is the primitive or the building block of this system. A finite number of spatial data structure types are allowed in this spatial information system. Each spatial data structure belongs to

one particular type of spatial data structure and has a unique name. The schema is developed in the form of a prototype structure for each type of spatial data structure. The prototype indicates what attributes may be found in the A/V relation of this type of spatial data structure and what relations besides the A/V relation comprise the data structure.

In order to find the spatial relationships between the entities let us analyze the available data first.

2.1 ANALYSIS OF DATA

The data[1] used in this system is of two types; the stream data and the road data. The stream data consists of watershed areas, water streams, and labels while the road data consists of a road network.

2.1.1 Stream Data

A digitized map[2] of the stream data along with a description of symbols used in the map to represent various things, is shown in Appendix A. Stream data for the wat-

---

1) This data was obtained from the Dept. of Fisheries and Wildlife Science, Virginia Tech, Blacksburg, VA.
2) This map is a subset of the WATERSHED AREA map for the APPALACHIA quadrangle, located in WISE county, VA. For more information refer to the U.S.G.S. map number N3652.5 - W8245/7.5.

ershed area N3 has been used.

The region N3 is divided into three subregions F1, F2, and F3. Each F region is further subdivided into T regions, For example, region F1 is subdivided in regions T1, T2, T3, and T4. Within each T region there is a network of streams. There are two types of streams, ephemeral streams and perrinial streams. Ephemerals are further classified according to their orders as E1 (first order), E2 (second order), E3 (third order), and E4 (fourth order). Similarly perrinials are classified on the basis of order as P1, P2, P3, P4, P5, and P6. Each region is identified by a label located within that region.

It can be observed from the map that the streams do not cross the boundary of any region and thus are entirely within one particular T region. Also the streams may intersect with streams of the same type or with streams of the other type.

2.1.2 Road Data

The road data used is a subset of the road data for the entire Appalachia quadrangle which includes region N3. The road network is similar to the stream network. There are two types of roads, primary and secondary. The roads also may

intersect with roads of the same type or of a different type, but unlike streams, the roads may cross the boundaries of regions.

2.2 PROTOTYPES

From the description of data it can be observed that the basic geographic entities used in the system are regions, water streams, roads, and labels. A region in this case can be represented by a polygon which has a closed boundary, a stream or a road can be represented by a chain which is comprised of an ordered list of points and a label can be represented by a point which has coordinates. Thus we have the following high level spatial data structure types: 1) REGION, 2) WATER STREAMS, 3) STREAM, 4) ROAD NETWORK, 5) ROAD, and 6) LABEL. The low level spatial data structure types are: 1) POLYGON and 2) CHAIN. A POINT is implemented as an atom.

Now let us describe each of these prototypes in details.

2.2.1 REGION

We represent each region by a spatial data structure. Figure 2.1 illustrates the prototype REGION. Each spatial data structure of type REGION consists of four relations : i) the A/V relation, A/V REGION, ii) SUBREGION ADJACENCY,

iii) STREAM NETWORK, and iv) LABELS. For example, a region R is a spatial data structure R = { A/V REGION R, SUBREGION ADJACENCY R, STREAM NETWORK R, LABELS R }.

2.2.1.1 A/V REGION

The A/V relation has four attributes : NAME whose value is a character string representing the name of the region, AREA whose value is a number representing the area of the region, BOUNDARY whose value is a spatial data structure of type POLYGON (to be described later) representing the boundary of the region, and PARENT whose value is a spatial data structure which itself is of type REGION, representing the next immediate region which encloses the region under consideration.

2.2.1.2 SUBREGION ADJACENCY

A region may have been divided into subregions in which case these subregions are stored in a SUBREGION ADJACENCY relation. It is a binary relation associating each subregion with every other subregion that neighbors it. Both the components of each pair in the relation are spatial data struc-

tures of type REGION.

2.2.1.3 STREAM NETWORK

The relations of type STREAM NETWORK are unary relations whose components are spatial data structures of type WATER STREAMS (to be described later).

2.2.1.4 LABELS

The relations of type LABELS are unary relations whose components are spatial data structures of type LABEL (to be described later).

2.2.2 <u>WATER STREAMS</u>

Figure 2.2 illustrates the prototype WATER STREAMS. There are two types of streams, ephemerals and perrinials. WATER STREAMS therefore consists of two relations : EPHEMERALS and PERRINIALS. For example, water streams WS is a spatial data structure WS = { EPHEMERALS WS, PERRINIALS WS }. Both EPHEMERALS and PERRINIALS are unary relations whose components are spatial data structures of type STREAM (to be described next).

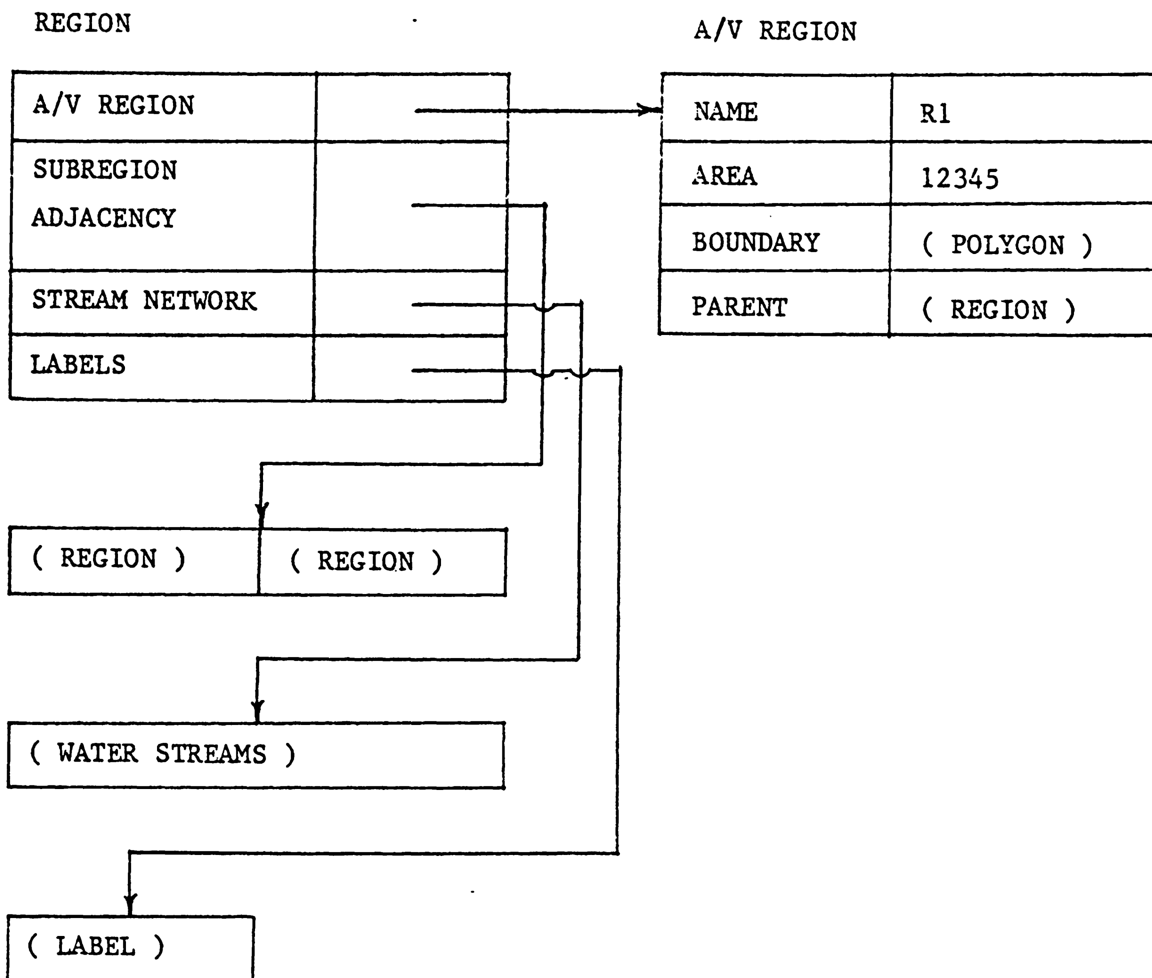


Figure 2.1 : Prototype REGION and its relations.

2.2.3 STREAM

The prototype STREAM is as shown in Figure 2.2. Each spatial data structure of type STREAM consists of two relations : an A/V relation called A/V STREAM and a binary relation INTERSECTING STREAMS. For example, a stream S is a spatial data structure S = { A/V STREAM S, INTERSECTING STREAMS S }.

2.2.3.1 A/V STREAM

The A/V relation has seven attributes : NAME, TYPE, and ORDER whose values are simple character strings representing the name of the stream, its type, and its order respectively; LENGTH, # INTERSECTING EPHEMERALS, and # INTERSECTING PERRINIALS whose values are numbers representing the length of the stream, the number of ephemerals intersecting, and the number of perrinials intersecting respectively; and COURSE whose value is a spatial data structure of type CHAIN (to be described later) representing the course of the stream.

2.2.3.2 INTERSECTING STREAMS

The relations of type INTERSECTING STREAMS are binary relations whose components represent the point of intersection, which is an atomic POINT (to be described later) and the stream intersecting at that point, which is a spatial data structure of type STREAM.

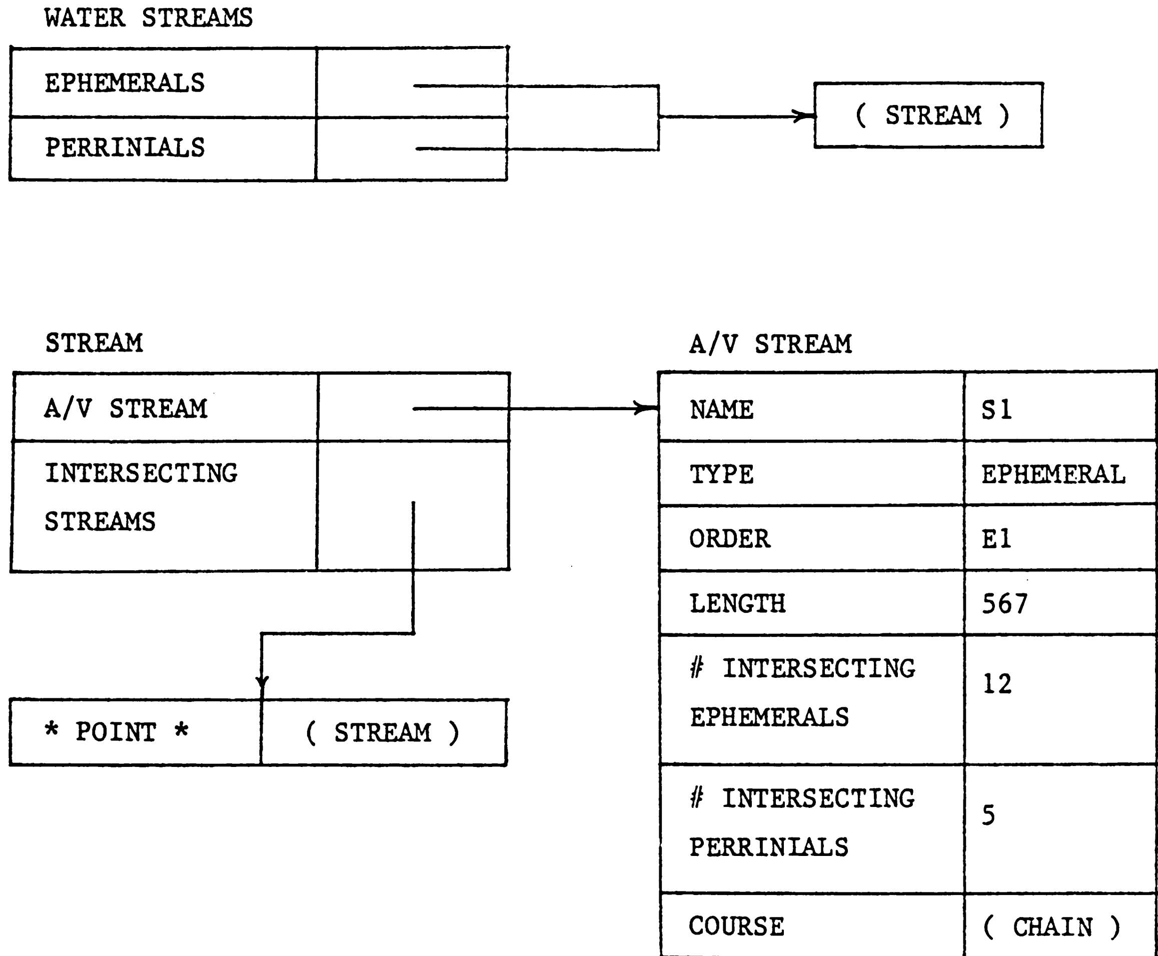


Figure 2.2 : Prototypes WATER STREAMS, STREAM and their relations.

2.2.4 ROAD NETWORK

Figure 2.3 illustrates the prototype ROAD NETWORK. There are two types of roads, primary and secondary. Each spatial data structure of type ROAD NETWORK therefore consists of two relations : PRIMARY and SECONDARY. For example, a road network RN is a spatial data structure RN = { PRIMARY RN, SECONDARY RN }. Both PRIMARY and SECONDARY are unary relations whose components are spatial data structures of type ROAD (to be described next).

2.2.5 ROAD

The prototype ROAD is as shown in Figure 2.3. Each spatial data structure of type ROAD consists of two relations : an A/V relation called A/V ROAD and a binary relation INTERSECTING ROADS. For example, road R is a spatial data structure R = { A/V ROAD R, INTERSECTING ROADS R }.

2.2.5.1 A/V ROAD

The A/V relation has six attributes : NAME and TYPE whose values are simple character strings representing the name and the type of the road respectively; LENGTH, # INTERSECTING PRIMARY, and # INTERSECTING SECONDARY whose values are numbers representing the length, the number of intersecting primary roads and the number of intersecting secondary roads

respectively; and COURSE whose value is a spatial data structure of type CHAIN (to be described later) representing the course of the road.

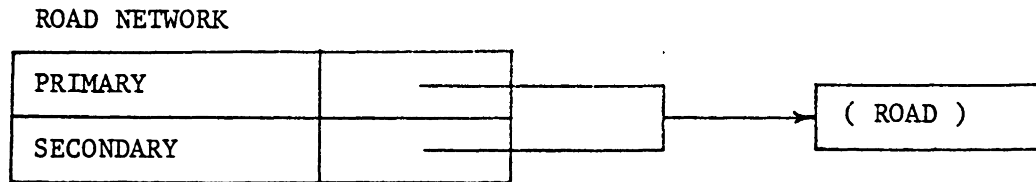
2.2.5.2 INTERSECTING ROADS

This is a binary relation whose components represent a point of intersection, which is an atomic POINT (to be described later) and a road that intersects at the point of intersection which is a spatial data structure of type ROAD.

2.2.6 LABEL

Figure 2.4 illustrates the prototype LABEL. Each spatial data structure of type LABEL consists of only one relation, an A/V relation called A/V LABEL. For example, a label L is a spatial data structure L = { A/V LABEL L }. The A/V relation in this case has two attributes, NAME whose value is a character string representing the name of the label and LOCATION whose value is an atomic POINT representing the location of the label.

After describing the types of the high level spatial data structures we now describe the low level spatial data structure types namely POLYGON and CHAIN and also the atom POINT.



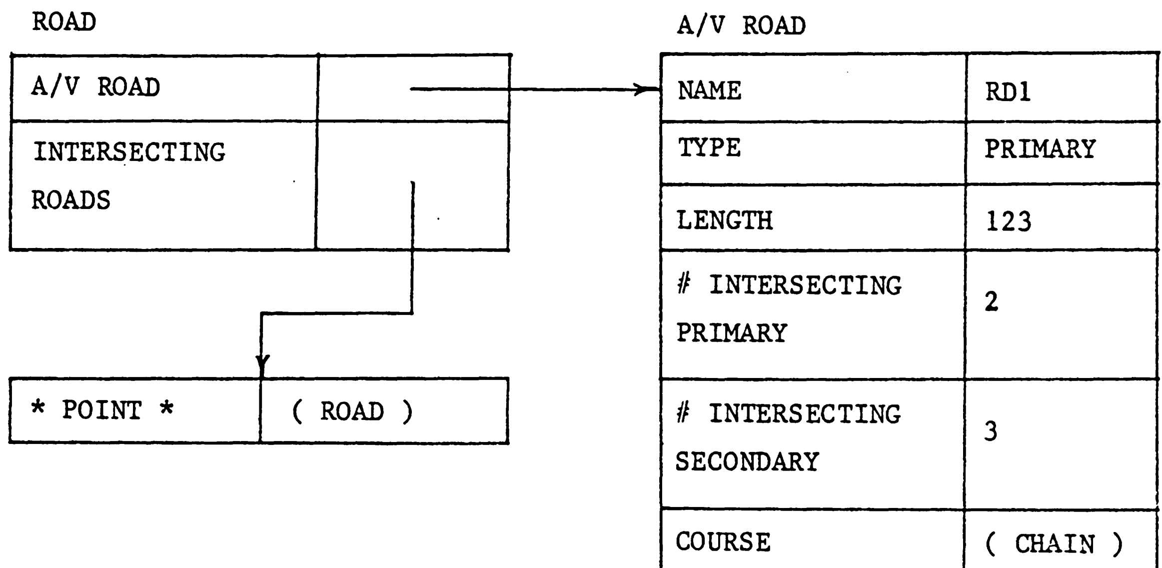


Figure 2.3 : Prototypes ROAD NETWORK, ROAD and their relations.

2.2.7 POLYGON

We represent the boundary of any region by a spatial data structure of type POLYGON. A polygon is comprised of chains. Each spatial data structure of type POLYOGN has a unary relation called CHAINS. For example, a polygon P is a spatial data structure P = { CHAINS P }. Every component of the relation CHAINS is a spatial data structure of type CHAIN (to be described next). The prototype POLYGON is shown in Figure 2.4.

2.2.8 CHAIN

The prototype for CHAIN is as shown in Figure 2.4. We represent the course of any water stream or road by a spatial data structure which is of type CHAIN [see subsection]. Each spatial data structure of type CHAIN is comprised of two relations : an A/V relation called A/V CHAIN and a relation POINTS. A chain C is thus a spatial data structure C = { A/V CHAIN C, POINTS C }.

2.2.8.1 A/V CHAIN

A chain has a region to its left and region to its right. The A/V relation therefore has two attributes : LEFT and RIGHT. The values of both these attributes are spatial data

structures of type REGION (explained in subsection 2.2.1).

2.2.8.2 POINTS

The relation POINTS is an ordered list (a binary relation) of points that define the chain.

2.2.9 POINT

POINT is an atom, a data element at the innermost level which can not be further broken down. A polygon, which represents the boundary of any region is comprised of chains. A chain, which also represents the course of a stream and a road, is in turn defined by points. A POINT is therefore an atom consisting of an ordered pair (X, Y) where X represents the latitude or the X co-ordinate and Y the longitude or the Y co-ordinate.

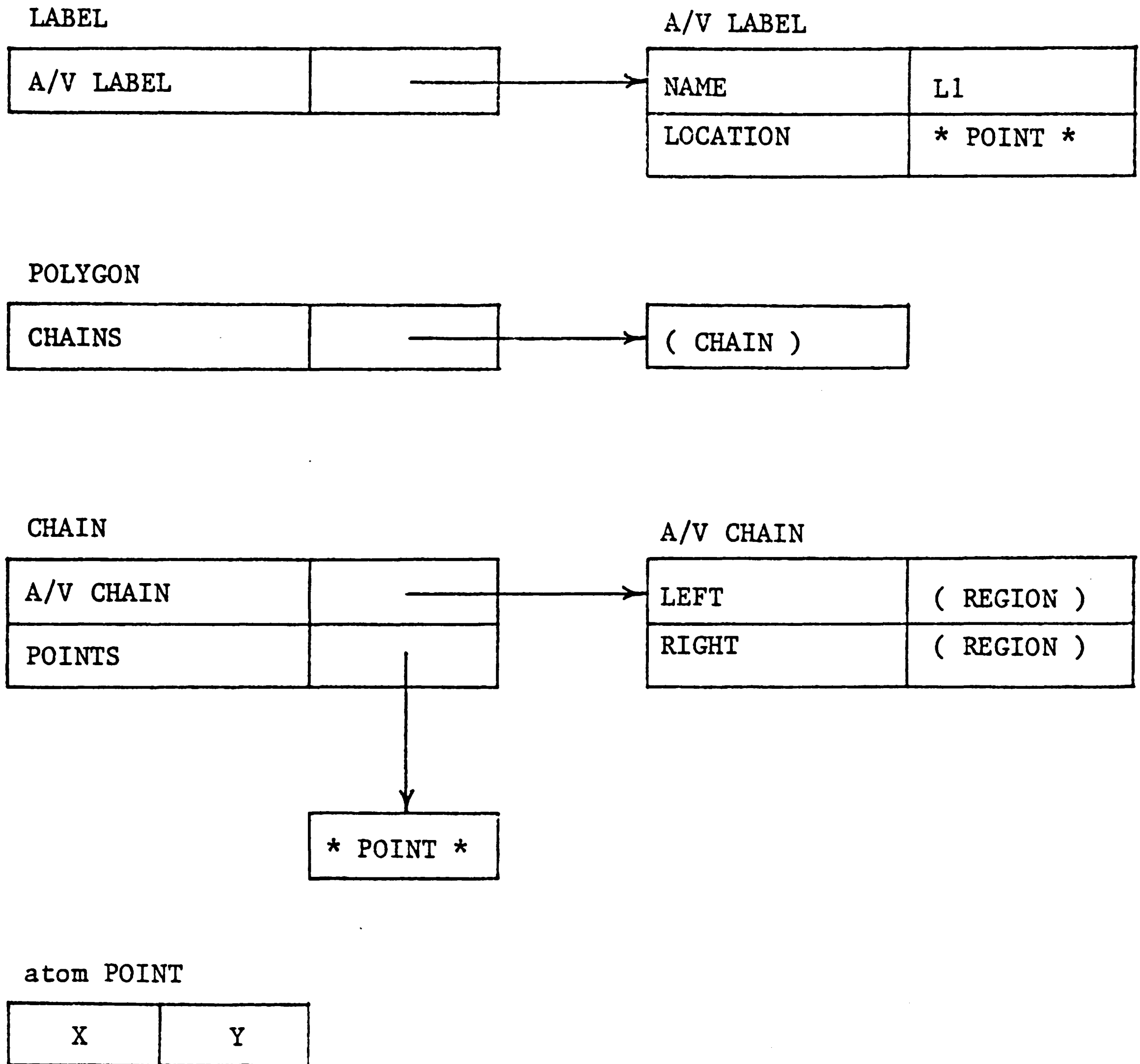


Figure 2.4 : Prototypes LABEL, POLYGON, CHAIN and atom POINT

Chapter III

DATA STRUCTURES

After designing the schema or the logical model of the database, the next step in the design of a database system is to design the physical structure of the database; that is, we must define how the data elements are physically structured according to the schema or the logical model. In order to achieve this, first of all it is necessary to define the data structures which will be used to implement the physical database structure. The data structures thus lead to the implementation of the physical model of the database from its logical model.

In chapter 3 we designed the schema or the logical database structure for our geographic information system, in the form of a prototype structure. In this chapter we describe various data structures which are used to represent the relationships between the entities. We use figures and Pascal-like declarations to describe these data types where appropriate.

The following Pascal-like convention is adopted for the use of the special notations and symbols in this chapter: A ':' has the implication ' is of type ', for example, A : B implies that A is of type B. An '| ' symbol implies a

pointer[3]. For example, A = | B implies that A is a pointer to the element B.

3.1 POINTERS

First of all we describe all the pointers which point to the actual data structures (the actual data structures will be described individually, later in this chapter).

3.1.1 RDS_PTR

RDS_PTR is a pointer to the data structure RDS.

RDS_PTR = | RDS;

3.1.2 RELATION_PTR

RELATION_PTR is also a pointer to the structure RDS.

RELATION_PTR = | RDS;

3.1.3 RELATION_CELL_PTR

RELATION_CELL_PTR points to a RELATION_CELL.

RELATION_CELL_PTR = | RELATION_CELL;

---

3) Pascal uses an 'up arrow' to indicate a pointer. An '| ' symbol was used due to the unavailability of the 'up arrow' on the print train of the printer, on which this thesis was printed.

3.1.4 TREE_CELL_PTR

TREE_CELL_PTR is a pointer to the data structure TREE_CELL.

```
TREE_CELL_PTR = | TREE_CELL;
```

3.1.5 LIST_CELL_PTR

LIST_CELL_PTR points to the structure LIST_CELL.

```
LIST_CELL_PTR = | LIST_CELL;
```

3.1.6 POINT_CELL_PTR

POINT_CELL_PTR is a pointer to the structure POINT.

```
POINT_CELL_PTR = | POINT_CELL;
```

3.1.7 CHAR_STRING_PTR

CHAR_STRING_PTR points to a CHAR_STRING_NODE.

```
CHAR_STRING_PTR = | CHAR_STRING_NODE;
```

3.2 STACK_ELEM

STACK_ELEM is our basic data element. It is a generalized data type that can hold integers; reals; pointers to spatial data structure headers, relation headers, relation cells, tree cells, list cells, point cells, and character string cells (all of these data types will be explained later).

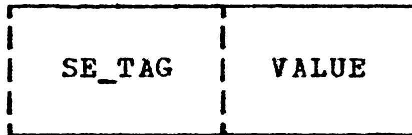
Internally, STACK_ELEM is represented as a pair consisting of a tag field, SE_TAG, indicating a data type, and a VALUE field. The VALUE may be the actual data structure, as in the case of integers, or it may be a pointer to the actual data structure, as in most other cases. Figure 3.1 illustrates the structure STACK_ELEM and its Pascal-like declaration. It also indicates the values allowed in the tag field, SE_TAG, and their implications with respect to the VALUE field.

Conceptually, we have three types of entities; spatial data structures, abbreviated as SDS's, relations, abbreviated as REL's and atoms. A spatial data structure is a set of relations. The components of these relations are either atoms or other spatial data structures. We describe the data types for representing each of these separately.

3.3 DATA TYPES FOR SPATIAL DATA STRUCTURES

A spatial data structure is composed of relations (refer to Chapter I). Each spatial data structre consists of two structures: its header, called an RDS, and a set of relations represented using a list structure, called a RELATION list.

STACK_ELEM




```
STACK_ELEM = record
                case SE_TAG : ELEM_TYPE of
                  RDS_ELEM   : (SE_RDS   : RDS_PTR);
                  REL_ELEM   : (SE_REL   : RELATION_PTR);
                  LIST_ELEM  : (SE_LIST  : LIST_CELL_PTR);
                  CHAR_ELEM  : (SE_CHAR  : CHAR_STRING_PTR);
                  INT_ELEM   : (SE_INT   : integer);
                  REAL_ELEM  : (SE_REAL  : real);
                  POINT_ELEM: (SE_POINT: POINT_CELL_PTR);
                  TREE_ELEM  : (SE_TREE  : TREE_CELL_PTR);
                  RC_ELEM    : (SE_RC    : RELATION_CELL_PTR);
              end;
```

Figure 3.1 : STACK_ELEM Using Pascal Convention

SE_TAG	VALUE
RDS_ELEM	RDS_PTR
REL_ELEM	RELATION_PTR
LIST_ELEM	LIST_CELL_PTR
CHAR_ELEM	CHAR_STRING_PTR
INT_ELEM	INTEGER
REAL_ELEM	REAL
POINT_ELEM	POINT_CELL_PTR
TREE_ELEM	TREE_CELL_PTR
RC_ELEM	RELATION_CELL_PTR

Figure 3.1 : continued

3.3.1 RDS

Each spatial data structure consists of a header represented by a structure called an RDS[4]. Figure 3.2 illustrates this structure. The RDS is a 6-tuple; NAME, TYPE, DIMENSION, LENGTH, USE_CNT, and STRUCT. The components of the RDS are described as follows: NAME is a simple character string representing the name of the SDS; TYPE is a user interpreted code for the type of the structure being pointed to by this header, which is zero for a spatial data strcuture[5]; DIMENSION is the dimension[6]; LENGTH is the number of relations in the SDS; USE_CNT is the number of other data structures that reference this SDS; and STRUCT points to the set of relations, represented in a list form (to be explained next), associated with this SDS.

Internally the RDS is a Pascal record with fields named above (refer to Figure 3.2). NAME is character string fifteen characters long; TYPE, DIMENSION, LENGTH, and USE_CNT are integers; and STRUCT is a pointer to a RELATION list, an

---

4) The same structure RDS is used to represent a relation header (to be explained later).
5) The value of TYPE is 0 if the RDS is a spatial data structure header, 1 if it is a TREE relation header, and 2 if it is a LIST relation header.
6) For spatial data structures this has little or no meaning, however it is used in relation headers (to be explained later).

internal data structure.

3.3.2 RELATION LIST

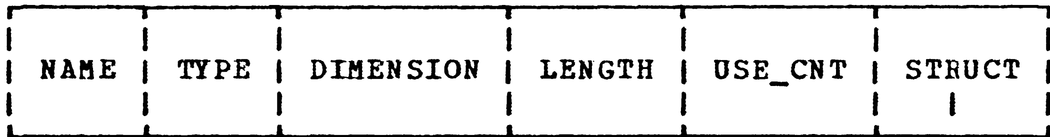
The RELATION list represents the set of relations associated with an SDS. It is composed of RELATION CELL's. Each RELATION CELL consists of two fields: RELATION, which is a pointer to a relation header and NEXT_RELATION, which points to the next RELATION_CELL. Figure 3.2 illustrates the RELATION_CELL and its Pascal-like declaration.

New relations are always attached at the begining of the RELATION CELL list, so that it is not necessary to scan the entire list. The RELATION CELL list is terminated by a nil pointer. When a relation is added to an SDS, both the LENGTH field of the SDS header and the USE_CNT field of the relation header are incremented by one.

3.4 MULTI-DIMENSIONAL RELATIONS

The RELATION data structure consists of a header, RDS, identical in structure to the SDS header described in section 4.3 and a TREE or a LIST that holds the N-tuples.

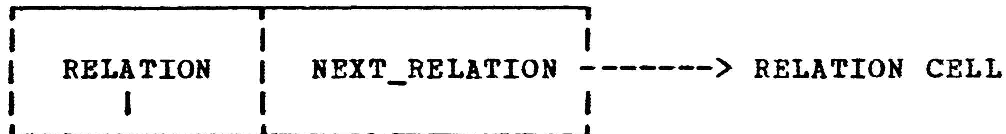
<u>Spatial Data Structure Header 'RDS'</u>



RELATION CELL list

```
RDS = record
          NAME : CHARACTER_STRING;
          TYPE : integer;
          DIMENSION : integer;
          LENGTH : integer;
          USE_CNT : integer;
          STRUCT : STACK_ELEM
      end;
```

<u>RELATION CELL</u>



Relation Header RDS

```
RELATION_CELL = record
                   RELATION : RELATION_PTR;
                   NEXT_RELATION : RELATION_CELL_PTR
                end;
```

Figure 3.2 : Spatial Data Structure Header RDS and RELATION CELL with their Pascal declarations.

3.4.1 RDS

The structure of the header is re-iterated here for reference. It has the following format: NAME represents the name of the relation; TYPE is a user interpreted code for the type of relation, 1 for the TREE relation and 2 for the LIST relation; DIMENSION represents the dimension or the order of the relation, that is, the number of components in each N-tuple of the relation; LENGTH is the number of N-tuples in the relation; USE_CNT is the number of other data structures that reference this relation; and STRUCT is a TREE or a LIST containing N-tuples for this relation. Figure 3.3 illustrates the relation header RDS. The Pascal-like declaration is as shown in Figure 3.2.

While creating a new relation, its header is created first. The NAME, TYPE, and DIMENSION are specified by the user. The LENGTH and USE_CNT are set to zero. The STRUCT is initialized to nil.

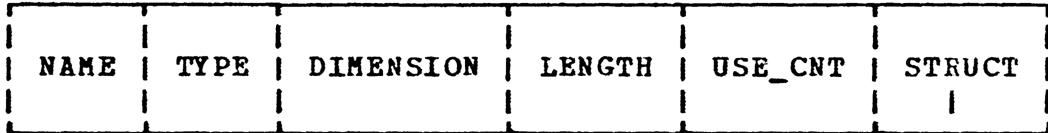
There are two types of relations: TREE relations and LIST relations, depending on whether the N-tuples are stored in a TREE or a LIST structure.

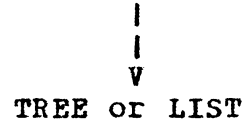
3.4.2 TREE RELATIONS

As the name implies, the relations of this type store the N-tuples in a TREE structure. The tree is N levels deep, corresponding to the dimension or the order of the relation. The first component of the N-tuple is stored on the first or the highest level. The second component is on the second level, and so on.

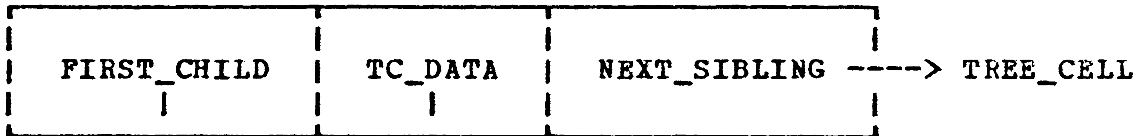
The tree is represented by a generalized tree structure [HOROE77]. Each node of the tree is called a TREE_CELL. A TREE_CELL has three fields: a data field, TC_DATA and two pointers, NEXT_SIBLING and FIRST_CHILD. The data field, TC_DATA contains the actual value of the data in case of atomic data, or it may contain a pointer to the header of another spatial data structure. The sibling pointer, NEXT_SIBLING points to the TREE_CELL at the same level and hence continues a list of TREE_CELLs on the same level. This list is terminated with a nil pointer. The child pointer, FIRST_CHILD, points to the TREE_CELL on the next level and thus continues a new list one level deeper. The child list is also terminated with a nil pointer, at each leaf of the tree. Figure 3.3 illustrates the TREE_CELL and its Pascal like declaration.

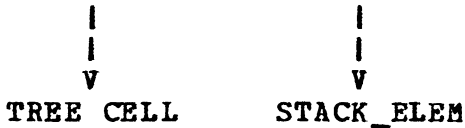
Relation Header 'RDS'





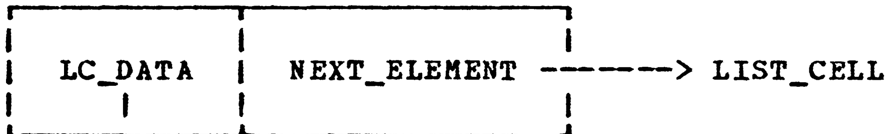
TREE CELL






```
TREE_CELL = record
                FIRST_CHILD : TREE_CELL_PTR;
                TC_DATA : STACK_ELEM;
                NEXT_SIBLING : TREE_CELL_PTR
            end;
```

LIST CELL



```
LIST_CELL = record
                LC_DATA : STACK_ELEM;
                NEXT_ELEMENT : LIST_CELL_PTR
            end;
```

Figure 3.3 : Relation Header RDS, TREE_CELL, and LIST_CELL.

The tree is structured so that all the N-tuples with the same first component share the same TREE_CELL. All the N-tuples with the same first two components share the same TREE_CELLs on the first two levels, and so forth. For example, a relation consisting of three N-tuples; (a, b, c), (a, b, d), and (b, c, d) will be stored as shown in Figure 3.4.

To facilitate searching the tree, and hence the relation, the N-tuples are stored in a lexigraphical order. The lexigraphical order is first based on the data type using the following precedence: SDS < RELATION < CHARACTER STRING < INTEGER < REAL < POINT. When two data types are the same, the ordering is based on their values. For INTEGERs and REALs, their numeric value is used. The ordering of the CHARACTER STRINGS is based on the underlying character set; it is thus character set dependent. SDS's and RELATIONs are ordered according to their names. When a new N-tuple is added to a relation, it is always added in lexigraphical order[7].

---

7) In our system it turns out that the data stored in the first component of all the N-tuples in the relation, are of same type.

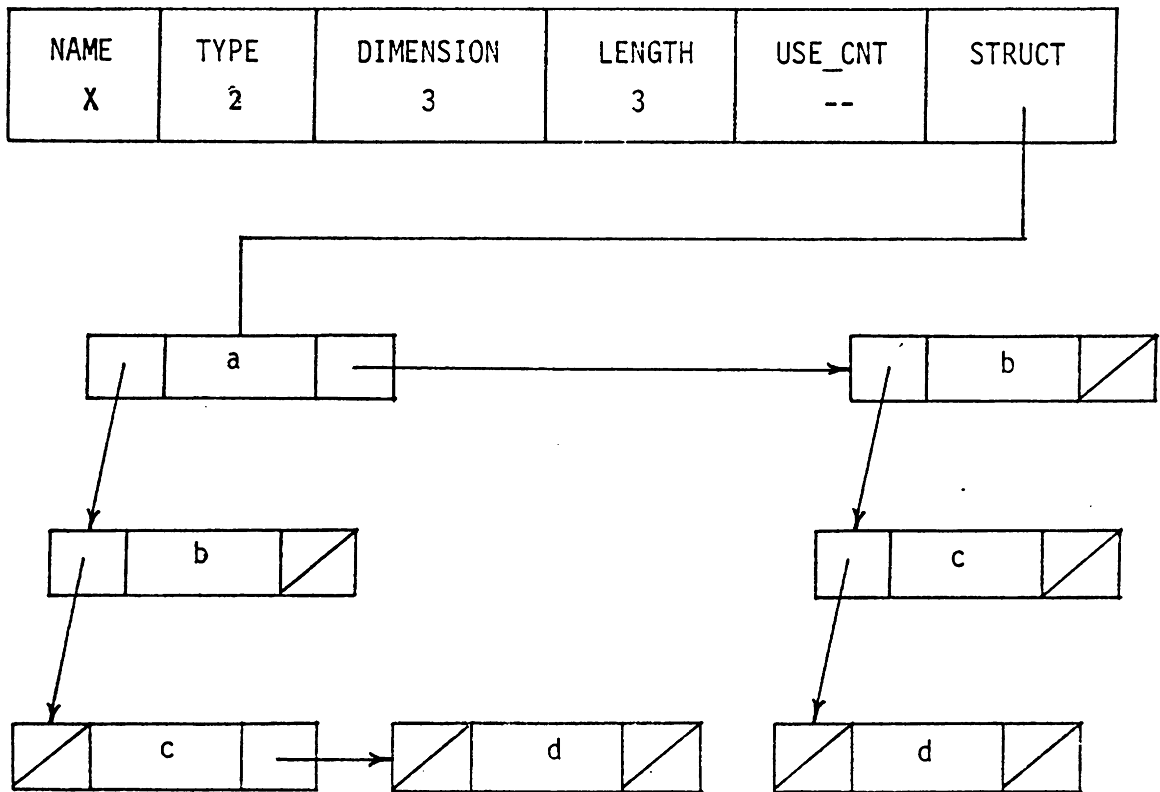


Figure 3.4 : N-tuples stored in a TREE relation.

3.4.3 LIST RELATIONS

The LIST RELATIONS store the N-tuples in a list form. These relations consists of LIST_CELLs. Figure 3.3 illustrates a LIST_CELL and its Pascal like declaration. A LIST_CELL consists of two fields: a data field, LC_DATA and a pointer, NEXT_ELEMENT. The data field, LC_DATA may contain the actual value of the data in case of atomic data, or a pointer to a SDS header in case the component is another spatial data structure, or it may point to a list in which an N-tuple is stored. The pointer NEXT_ELEMENT points to the next LIST_CELL in the list.

In a LIST RELATION there are two lists; a list of N-tuples, which grows vertically downwards and a list of components of an N-tuple, which grows horizontally. The STRUCT field of a relation header points to a LIST_CELL, whose LC_DATA field points to the first N-tuple (actually it points to the LIST_CELL containing the first component of that N-tuple) and NEXT_ELEMENT field points to the LIST_CELL pointing to the second N-tuple, and so on. Figure 3.5 shows how a relation consisting of N-tuples (a, b, c), (a, b, d), and (b, c, d) is represented as a LIST RELATION.

The order of the N-tuples stored in a LIST relation is user defined, and it is easier to insert N-tuples in a LIST

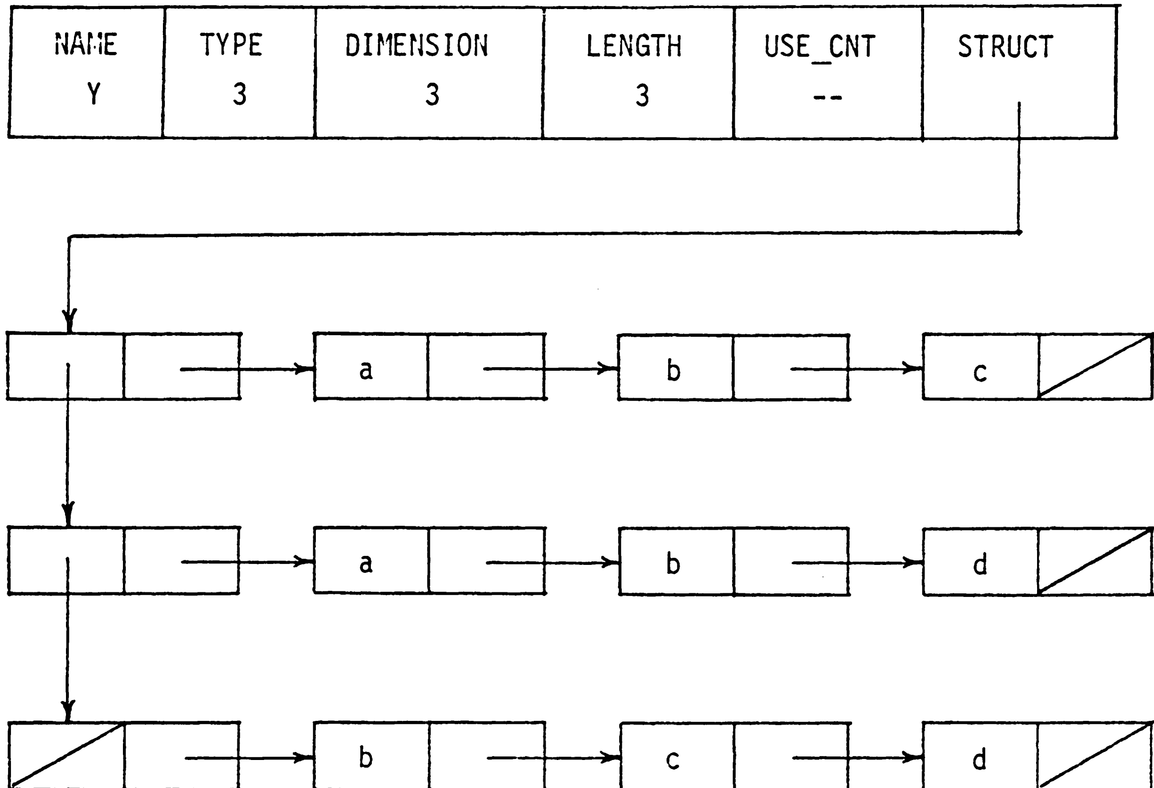


Figure 3.5 : N-tuples stored in a LIST relation.

relation. A TREE relation, on the other hand, stores the N-tuples in a lexigraphical order. This provides a quicker access when the relation is being searched for an N-tuple by its content, than a LIST relation would.

3.5 ATOMS

There are four types of atomic data elements: INTEGER, REAL, CHARACTER STRING, and POINT. INTEGERs and REALs are represented in the usual manner, by data types INTEGER, and REAL respectively. CHARACTER STRINGs and POINTs have their own individual structures.

3.5.1 CHARACTER STRING

The character strings are used in two forms, simple and linked. The two forms are compatible. A simple character string is a sequence of not more than fifteen characters, left justified, and padded by blanks (the system does the padding). All the names of the SDS's and relations are simple character strings.

When longer character strings are needed, several simple character strings are linked together. This forms a linked character string. It consists of a list of CHARACTER_STRING_NODEs. Each CHARACTER_STRING_NODE has two fields: C_S, which is a simple character string of fifteen charac-

ters and CS_LINK, which is a pointer pointing to the next CHARACTER_STRING_NODE. Linked character strings may be used as data in other data structures. Figure 3.6 illustrates the structure of the linked character string and its Pascal declaration.

3.5.2 POINT

The structure POINT is used to represent point data. It consists of a POINT_CELL which has two fields, X and Y, both of which are of type INTEGER. X represents the x-coordinate of the point, and Y represents the y-coordinate of the point. This is as shown in Figure 3.6.

3.6 ARRAYS

In this section we describe various arrays defined in the system. These are esentially arrays of the data structures described earlier in this chapter.

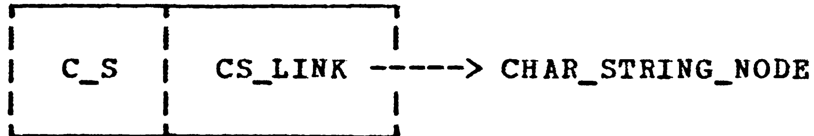
3.6.1 RDS_ARRAY

This is an array of RDS_PTRs, that is, each element of RDS_ARRAY is a pointer pointing to a spatial data structure header RDS.

RDS_ARRAY = array [1..UB] of RDS_PTR; where UB is a predefined constant, indicating the upper bound on the number of

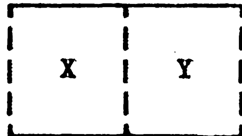
```
CHAR_STRING = packed array [ 1..15 ] of char;
```

CHAR_STRING_NODE




```
CHAR_STRING_NODE = record
                       C_S : CHAR_STRING;
                       CS_LINK : CHAR_STRING_PTR
                   end;
```

POINT




```
POINT = record
            X : integer;
            Y : integer
        end;
```

Figure 3.6 : Data Structures CHAR_STRING_NODE and POINT.

SDS's that can be formed.

3.6.2 REL_ARRAY

This is an array of REL_PTRs, that is each element is a pointer pointing to a relation header RDS.

REL_ARRAY = array [1..UB] of RELATION_PTR; where UB is a predefined constant, indicating the maximum number of relations that can be formed.

3.6.3 STACK_ARRAY

STACK_ARRAY is an array of STACK_ELEMs. An element of this array may be any one of the data types mentioned earlier, hence different data types may be stored in the same array. This array is particularly used by the query language interpreter (to be explained later). It holds the arguements of the query language commands and the results of the operations.

STACK_ARRAY = array [1..MAX_STACK] of STACK_ELEM; where MAX_STACK is a predefined constant.

3.6.4 NTUPLE

The array NTUPLE holds the components of one N-tuple. Basically, it is an array of STACK_ELEMs.

NTUPLE = array [1..UB_NTUPLE] of STACK_ELEM; where UB_NTUPLE is a predefined constant specifying the maximum number of components in an N-tuple.

3.6.5 TREE POSITION

This array holds a position in a tree. It is an array of TREE_CELL_PTRs, that is, each element of this array is a pointer pointing to a TREE_CELL.

TREE_POSITION = array [1..UB_NTUPLE] of TREE_CELL_PTR; where UB_NTUPLE is again a predefined constant specifying the maximum number of tree positions (esentially it is the maximum number of components in an N-tuple).

3.6.6 NAME ARRAY

This is an array of CHAR_STRINGs, that is, each element is a character string, fifteen characters long.

NAME_ARRAY = array [1..UB] of CHAR_STRING; where UB is the predefined upper bound.

3.6.7 INT ARRAY

This is an array of integers.

INT_ARRAY = array [1..UB] of integer;

3.7 DICTIONARIES

The system keeps two dictionaries: RDS_DICTIONARY, a dictionary of spatial data structures, and REL_DICTIONARY, a dictionary of relations. Both these dictionaries are comparable.

3.7.1 RDS DICTIONARY

This data structure represents a spatial data structure dictionary. RDS_DICTIONARY maps a name to a spatial data structure. It consists of four arrays: RDS_NAME, which is of type NAME_ARRAY, representing the names of the spatial data structures; RDS_ADDRESS, which is of type RDS_ARRAY, representing pointers pointing to the spatial data structure headers corresponding to their names; RDS_TAG, an INT_ARRAY, representing a tag to indicate whether the correponding spatial data structure is in memory or not (one if in memory and zero otherwise); and RDS_CHANGE, an INT_ARRAY, representing again a tag to indicate whether the corresponding spatial data structure has been changed or not (one if changed and zero otherwise). Figure 3.7 illustrates the structure RDS_DICTIONARY.

Spatial data structures are kept in this dictionary in a lexigraphical order based on their names. Every time a new spatial data structure is created, its entry is made in the

RDS_DICTIONARY at an appropriate place. This allows the use of a binary search mechanism to locate any spatial data structure in the dictionary. Positions of specific SDS's may change as SDS's are added or deleted from the dictionary. An integer variable NUM_RDS indicated the number of entries in the RDS_DICTIONARY.

3.7.2 REL_DICTIONARY

REL_DICTIONARY represents a relation dictionary. It maps a name to a relation. It consists of four arrays: REL_NAME, a NAME_ARRAY, representing the names of the relations; REL_ADDRESS, a REL_ARRAY, representing the pointers to the corresponding relations (actually pointers to their headers); REL_TAG, an INT_ARRAY, representing a tag that indicates whether the corresponding relation is in memory (one if in memory and zero otherwise); and REL_CHANGE, an INT_ARRAY, representing a tag to indicate whether the corresponding relation has been changed (one if changed and zero otherwise). The structure REL_DICTIONARY is as shown in Figure 3.7.

Similar to RDS_DICTIONARY, the relations stored in the REL_DICTIONARY are ordered lexigraphically based on their names. Also whenever a new relation is created, it is entered in the dictionary. This facilitates locating any

relation. Positions of specific relations may change as the relations are added or deleted from the dictionary. An integer variable NUM_REL indicates the total number entries in the REL_DICTIONARY.

RDS_DICTIONARY

RDS_NAME	RDS_ADDRESS	RDS_TAG	RDS_CHANGE

```
RDS_DICTIONARY = record
                    RDS_NAME : NAME_ARRAY;
                    RDS_ADDRESS : RDS_ARRAY;
                    RDS_TAG : INT_ARRAY;
                    RDS_CHANGE : INT_ARRAY
                 end;
```

REL_DICTIONARY

REL_NAME	REL_ADDRESS	REL_TAG	REL_CHANGE

```
REL_DICTIONARY = record
                    REL_NAME : NAME_ARRAY;
                    REL_ADDRESS : REL_ARRAY;
                    REL_TAG : INT_ARRAY;
                    REL_CHANGE : INT_ARRAY
                 end;
```

Figure 3.7 : RDS_DICTIONARY and REL_DICTIONARY.

Chapter IV

THE PHYSICAL DATABASE STRUCTURE

In the preceding chapters we defined the concept of the spatial data structure, which is the basic primitive building block of our information system. We then designed the logical model of the database, which indicates how the geographic entities and the other information can be represented in the form of spatial data structures and the relations associated with them. Next we described the various inner level data structures which are used to represent the relationships between the entities. In this chapter we describe the physical structure of the database, which is the physical representation of the logical model or schema, using the data structures.

First, we describe how the data elements are structured in memory, in accordance with the logical structure (refer to Chapter II) using the data structures defined in Chapter III. Next, we explain how the database is implemented and how all the structures and the relations are transferred to the disk from memory. We then describe the organization of the database on the secondary device, that is, how the spatial data structures and the relations are stored physically on the disk.

4.1 THE PHYSICAL STRUCTURE IN MEMORY

We have the following six high-level spatial data structure types: REGION, WATER STREAMS, STREAM, ROAD NETWORK, ROAD, and LABEL. The two low level spatial data structure types are POLYGON and CHAIN (refer to Chapter II). Each spatial data structure defined in the system belongs to one of these eight spatial data structure types and has a set of relations associated with it. In this section we describe the internal representation of the spatial data structures of each of the above mentioned types and of the relations associated with them. As usual we will use figures to provide the best illustrations.

4.1.1 SPATIAL DATA STRUCTURE FOR A REGION

A region X is represented by a spatial data structure (SDS) REGION_X which is of type REGION. X may take any one of the following values: T1, T2, ..., T11, F1, F2, F3, or N3 (refer to the desription of the data in Chapter II). Figure 4.1 illustrates the structural representation of this SDS and its relations. The spatial data structure REGION_X consists of a header record, called an RDS, and a relation list. The header record gives the general information about this SDS: its NAME, which is REGION_X; its TYPE, which is one, specifying that it is an SDS; the LENGTH, which is four

because it has four relations associated with it; its DIMENSION and USE_CNT which are zero, because they are not used for SDS's. The STRUCT field points to the relation list. The spatial data structure REGION_X has four relations associated with it; A/V REGION_X, SUBREGION ADJACENCY_X, STREAM NETWORK_X, and LABELS_X. The relation list has four RELATION_CELLs which point to (the headers of) these relations.

The A/V relation A/V REGION_X is a binary relation consisting of an attribute-value table. It is implemented as a TREE relation as shown in Figure 4.1. It consists of its header (an RDS) and a tree structure that holds N-tuples. The TYPE field of the header is set to two, specifying that it represents a TREE relation. The values of the attributes NAME and AREA are atomic, while those of BOUNDARY and PARENT are spatial data structures of type POLYGON and REGION respectively.

The relation SUBREGION ADJACENCY_X is a binary relation which associates each subregion of the region X with every other subregion that neighbors it. This relation stores the N-tuples in a TREE structure. Both the components of each N-tuple in the relation are spatial data structures of type REGION.

The relation STREAM NETWORK_X is a unary relation represented in a TREE form. Each component is a spatial data structure of type WATER STREAMS. The structural representation is as shown in Figure 4.1. The length of this relation, that is, the number of N-tuples in the relation depends on the number of T regions enclosed by the region X. This is because the streams are confined within the boundary of a T region. For example, the length of each of the relations STREAM NETWORK_T1, ..., STREAM NETWORK_T11 is one, as the T regions are not further subdivided and have only one stream network within their boundaries. The relation STREAM NETWORK_F1 has four N-tuples since the region F1 is divided into four T regions, each of which has a stream network within it. Similarly the lengths of the relations STREAM NETWORK_F2, STREAM NETWORK_F3, and STREAM NETWORK_N3 are four, three, and eleven, respectively.

The relation LABELS_X is also a unary relation represented in a TREE structure. Each component of an N-tuple in this relation, is a spatial data structure of type LABEL. This is as shown in Figure 4.1. The length of this relation depends on the number of labels in that region. For example, any T region has only one label inside it, and hence the lengths of the relations LABELS_T1, ..., and LABELS_T11 are one. Similarly the lengths of the relations LABELS_F1,

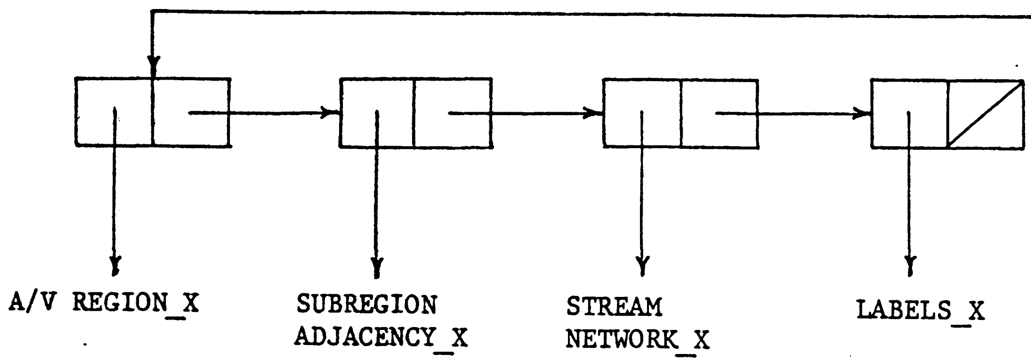
LABELS_F2, LABELS_F3, and LABELS_N3 are four, four, three, and eleven, respectively.

4.1.2 SPATIAL DATA STRUCTURE FOR A WATER STREAM

The structural representation of the spatial data structure WATER STREAMS_X, which is of type WATER STREAM is illustrated in Figure 4.2. X is the name of a region and is one of T1, T2, ..., T11. It consists of a header record (an RDS) and a relation list. Since there are two types of streams, ephemerals and perrinials, the relation list consists of two RELATION_CELLs which point to the relations (actually to their headers) EPHEMERALS_X and PERRINIALS_X respectively. Both of these relations are unary TREE relations. Each component of each N-tuple is a spatial data structure of type STREAM. The length of the relation depends upon the total number of streams.

4.1.3 SPATIAL DATA STRUCTURE FOR A STREAM

A stream I is represented by a spatial data structure STREAM_I which is of type STREAM. I may take a value between 1 and N, where N is the total number of streams in the system. Figure 4.3 illustrates the structural representation of this spatial data structure. As usual STREAM_I consists of



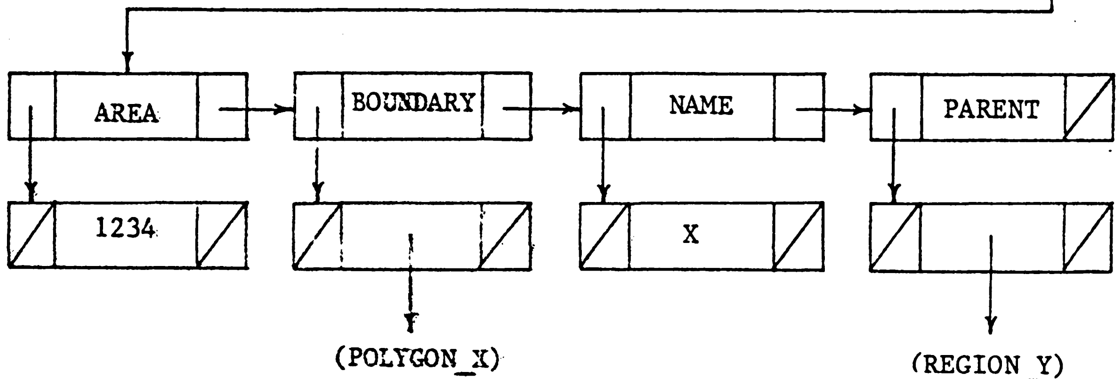
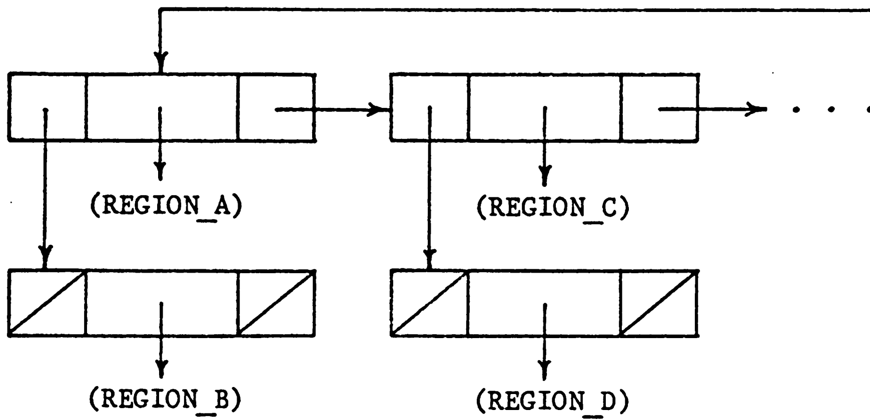


Figure 4.7 : Spatial Data Structure REGION_X and its relations.

NAME SUBREGION ADJACENCY_X	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
	2	2	n	1	



NAME STREAM NETWORK_X	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
	2	1	m	1	

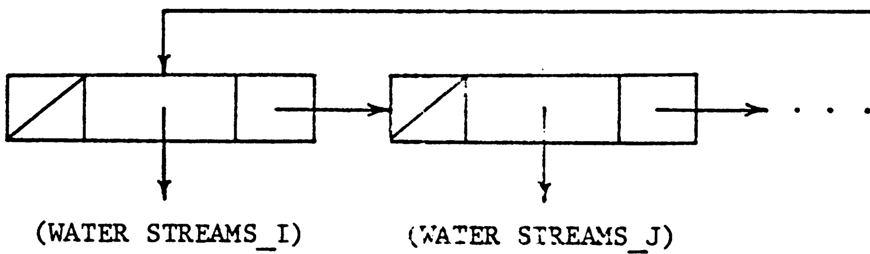


Figure 4.7 : continued..

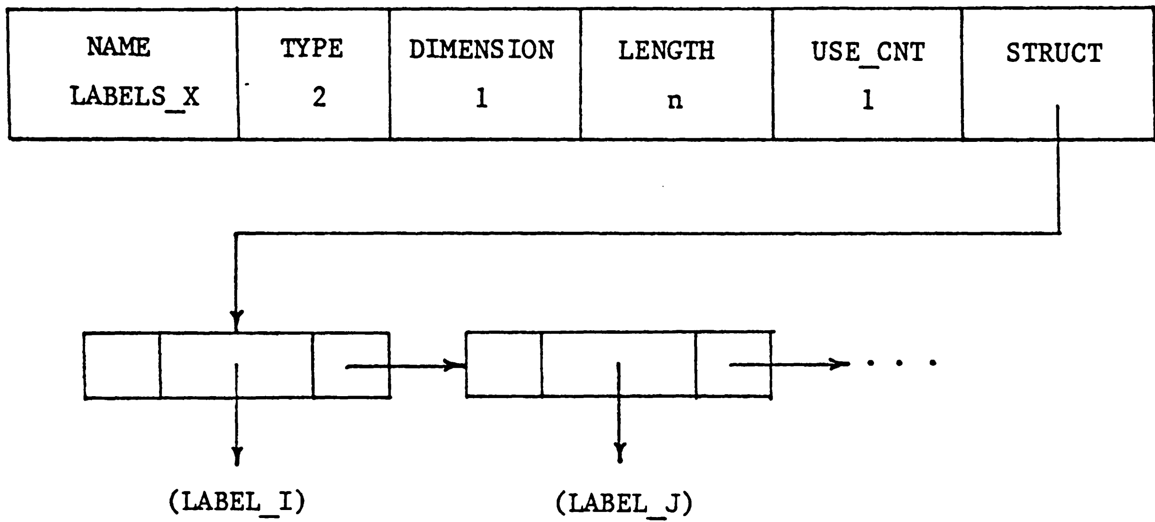
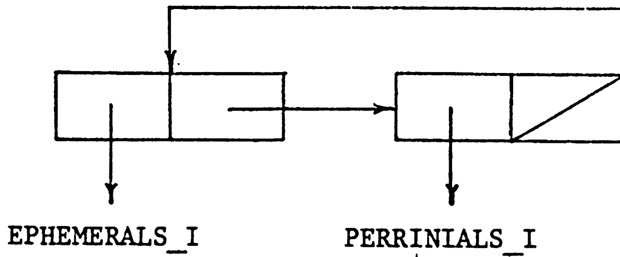
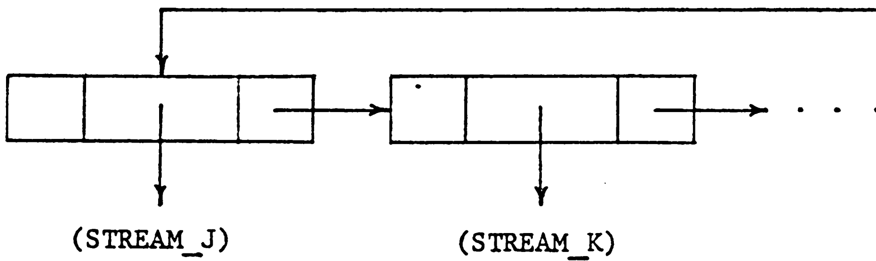


Figure 4.1 : continued..

NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
WATER STREAMS_I	1	0	n	m	



NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
EPHEMERALS_I	2	1	n	1	



NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
PERRINIALS_I	2	1	n	1	

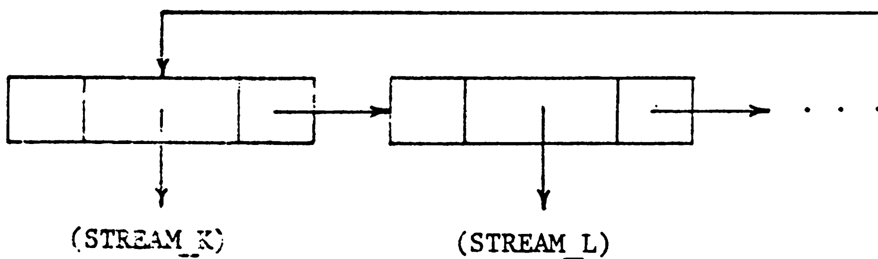


Figure 4.2 : Spatial Data Structure WATER_STREAMS_I and its relations.

a header (an RDS) and a relation list. STREAM_I has two relations associated with it, A/V STREAM_I and INTERSECTING STREAMS_I. The relation list thus has two RELATION_CELLs pointing to these relations.

The A/V relation A/V STREAM_I is an attribute-value table. It is a binary relation represented in a TREE form. The values of all but one attributes are atomic. The value of the attribute COURSE is a spatial data structure of type CHAIN.

The relation INTERSECTING STREAMS_I is a binary TREE relation. The first component of each N-tuple is atomic and represents the point of intersection. The second component is a spatial data structure of type STREAM and represents the stream intersecting at that point. It should be noted here that if there are two or more streams intersecting the given stream at the same point, they will share the same point of intersection; that is, those N-tuples will have the same first component. The length of the relation depends on the number of streams intersecting the given stream I.

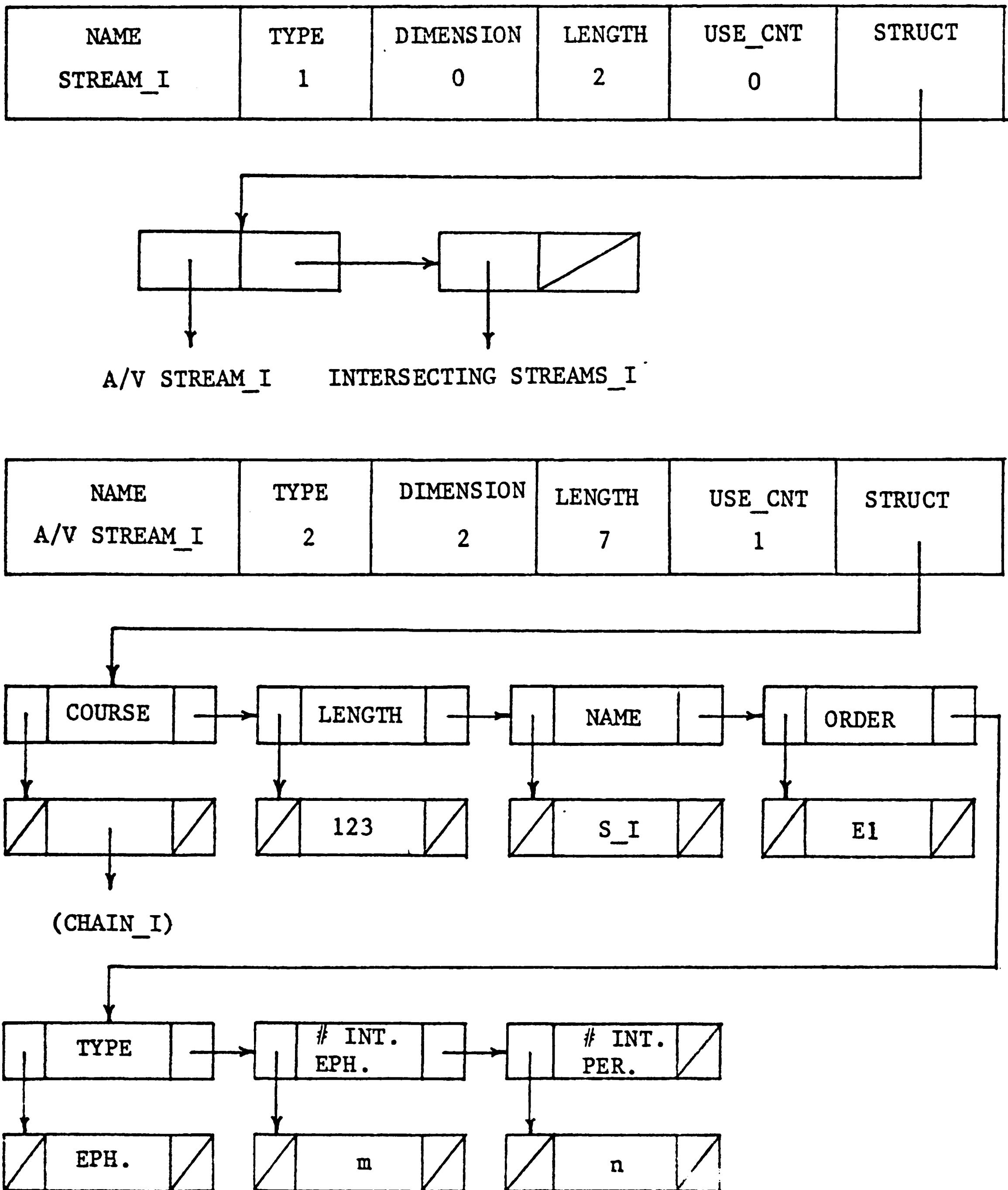


Figure 4.3 : Spatial Data Structure STREAM_I and its relations.

4.1.4 SPATIAL DATA STRUCTURE ROAD NETWORK

Figure 4.4 illustrates the internal structural representation of the spatial data structure ROAD NETWORK. This SDS has two relations associated with it (PRIMARY and SECONDARY) since the roads are of two types (refer to the description of the data in Chapter II). The relation list therefore consists of two RELATION_CELLs pointing to these two relations.

Both the relations PRIMARY and SECONDARY are unary TREE relations. The components of these relations are spatial data structures of type ROAD. The lengths of these relations depend on the number of primary and secondary roads. Figure 4.4 illustrates their structure.

4.1.5 SPATIAL DATA STRUCTURE FOR A ROAD

We represent a road I by a spatial data structure ROAD_I which is of type ROAD. The internal representation of the spatial data structure ROAD_I is similar to that of the spatial data structure STREAM_I described in the earlier subsection. This is as shown in Figure 4.5. The relation list consists of two RELATION_CELLs pointing to the two relations A/V ROAD_I and INTERSECTING ROADS_I respectively.

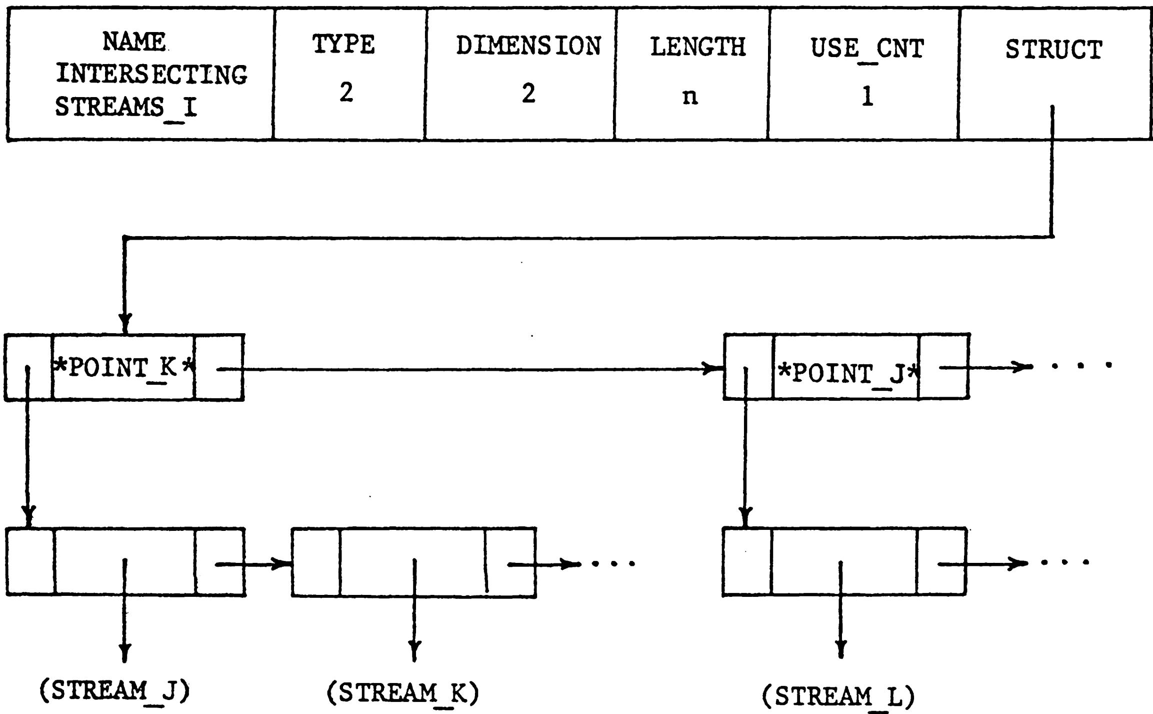
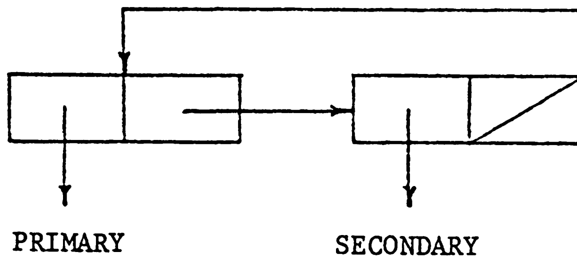
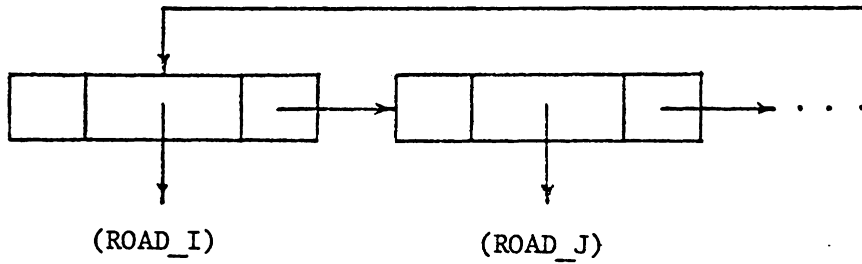


Figure 4.3 : continued..

NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
ROAD NETWORK	1	0	2	0	



NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
PRIMARY	2	1	n	1	



NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
SECONDARY	2	1	m	1	

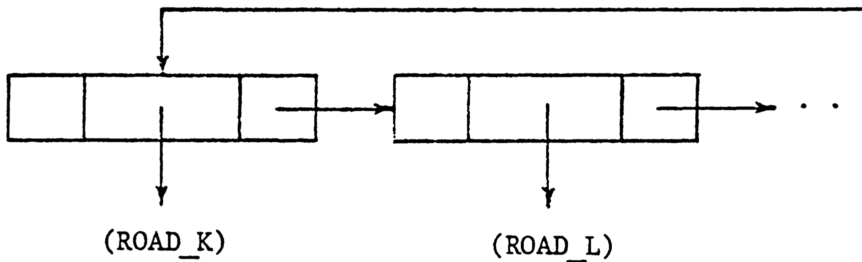


Figure 4.4 : Spatial Data Structure ROAD_NETWORK and its relations.

The A/V relation A/V ROAD_I is a binary relation. The N-tuples are stored in a TREE form. The values of all the attributes are atomic except for the attribute COURSE, whose value is a spatial data structure of type CHAIN. This relation is as shown in Figure 4.5.

The relation INTERSECTING ROADS_I is also a binary relation represented in a TREE structure. The first component is an atomic POINT representing the point of intersection, while the second component is a spatial data structure of type ROAD representing the intersecting road at that point of intersection. The length of this relation is determined by the number of roads intersecting the given road I. If two or more roads intersect the given road I at the same point, they will share the same point of intersection.

4.1.6 SPATIAL DATA STRUCTURE FOR A LABEL

A label I is represented by a spatial data structure LABEL_I which is of type LABEL. Its internal structural representation is as shown in Figure 4.6. The relation list of this SDS consists of only one RELATION_CELL since there is only one relation, A/V LABEL_I, associated with it.

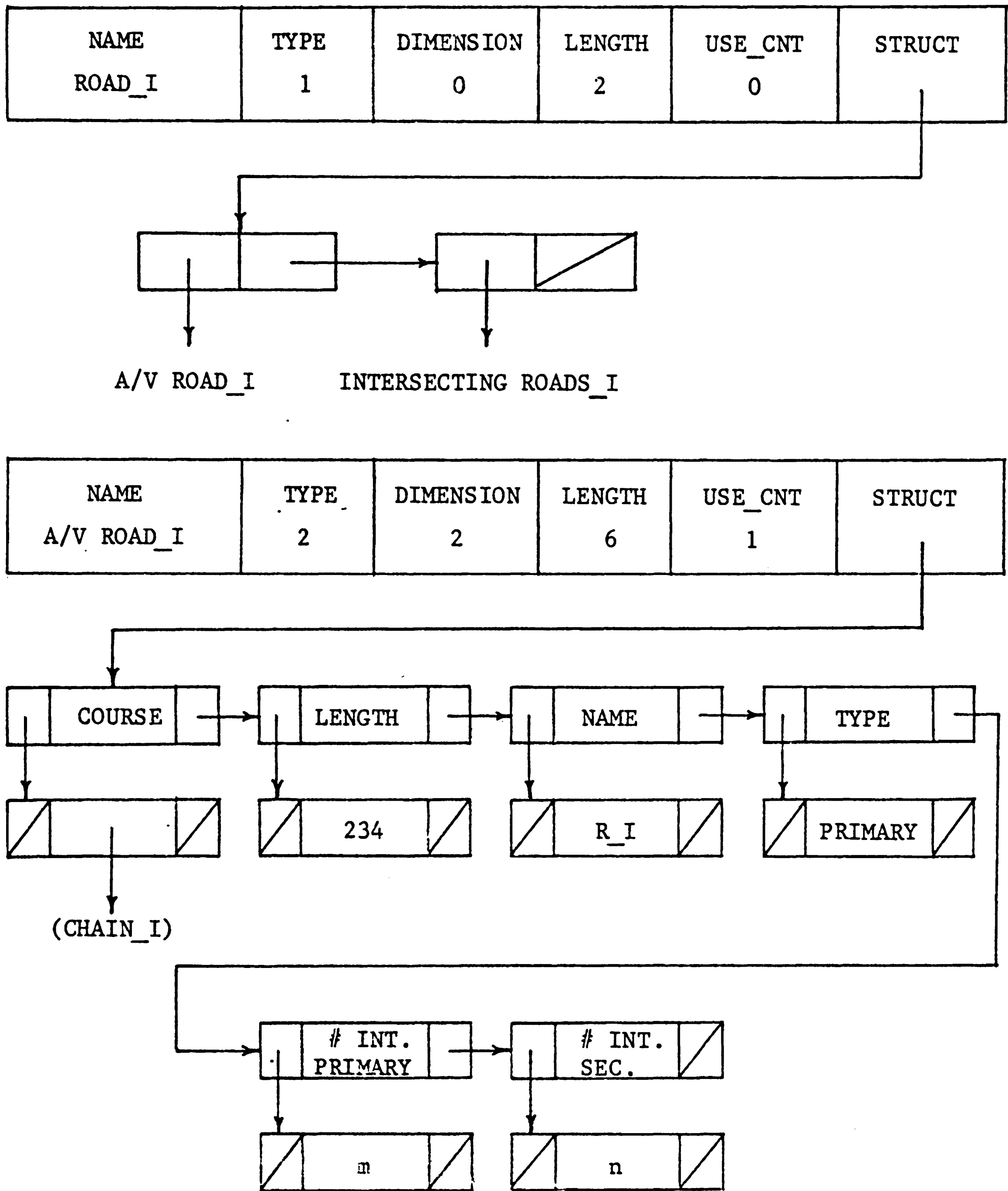


Figure 4.5 : Spatial Data Structure ROAD_I and its relations.

The A/V relation A/V LABEL_I is an attribute-value table. It is a binary relation, and the N-tuples are stored in a TREE structure. The values of both the attributes, NAME and LOCATION are atomic. The relation is represented structurally in memory, as shown in Figure 4.6.

4.1.7 SPATIAL DATA STRUCTURE FOR A POLYGON

We represent the boundary of any region X by a spatial data structure POLYGON_X, which is of type POLYGON. The relation list of POLYGON_X contains only one RELATION_CELL, as there is just one relation, CHAINS_X associated with it. This is as shown in Figure 4.7.

The relation CHAINS_I is a unary relation, represented in a TREE structure. The component of each N-tuple is a spatial data structure of type CHAIN. The length of this relation is determined by the number of chains that form that polygon.

4.1.8 SPATIAL DATA STRUCTURE FOR A CHAIN

The boundary of any region is comprised of chains. Also, the course of any stream or road is represented by a chain. A chain I is represented by a spatial data structure CHAIN_I which is of type CHAIN. Figure 4.8 describes the internal

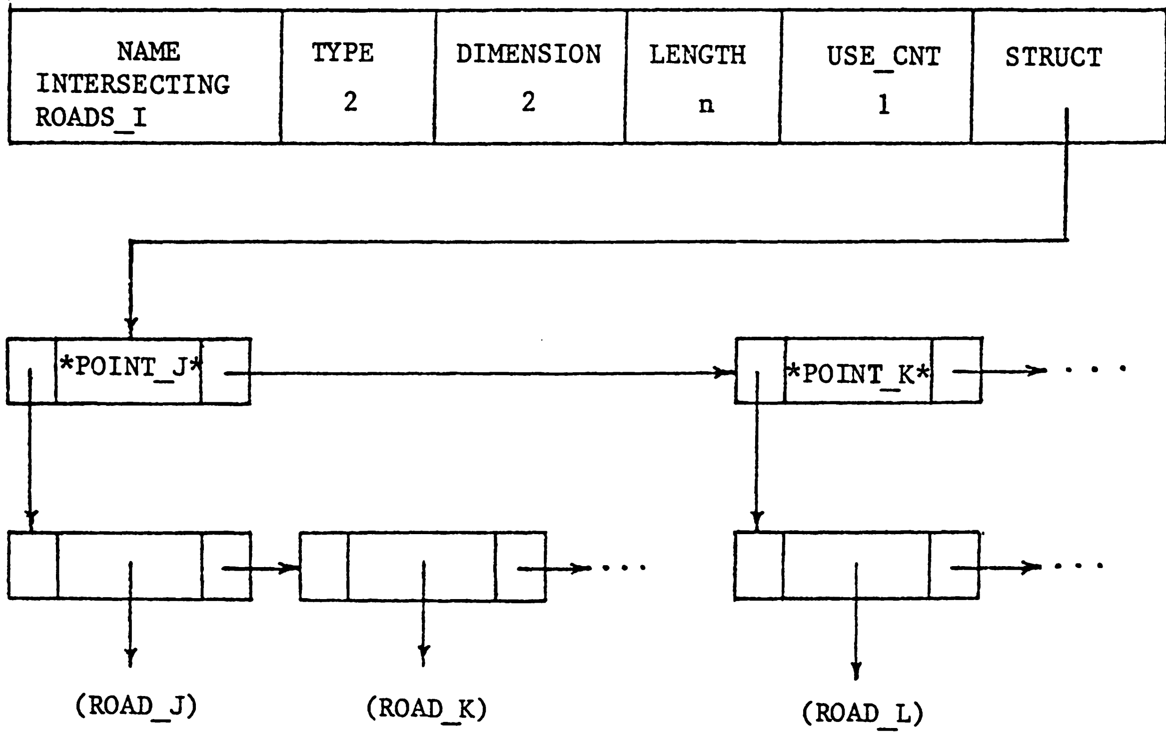


Figure 4.5 : continued..

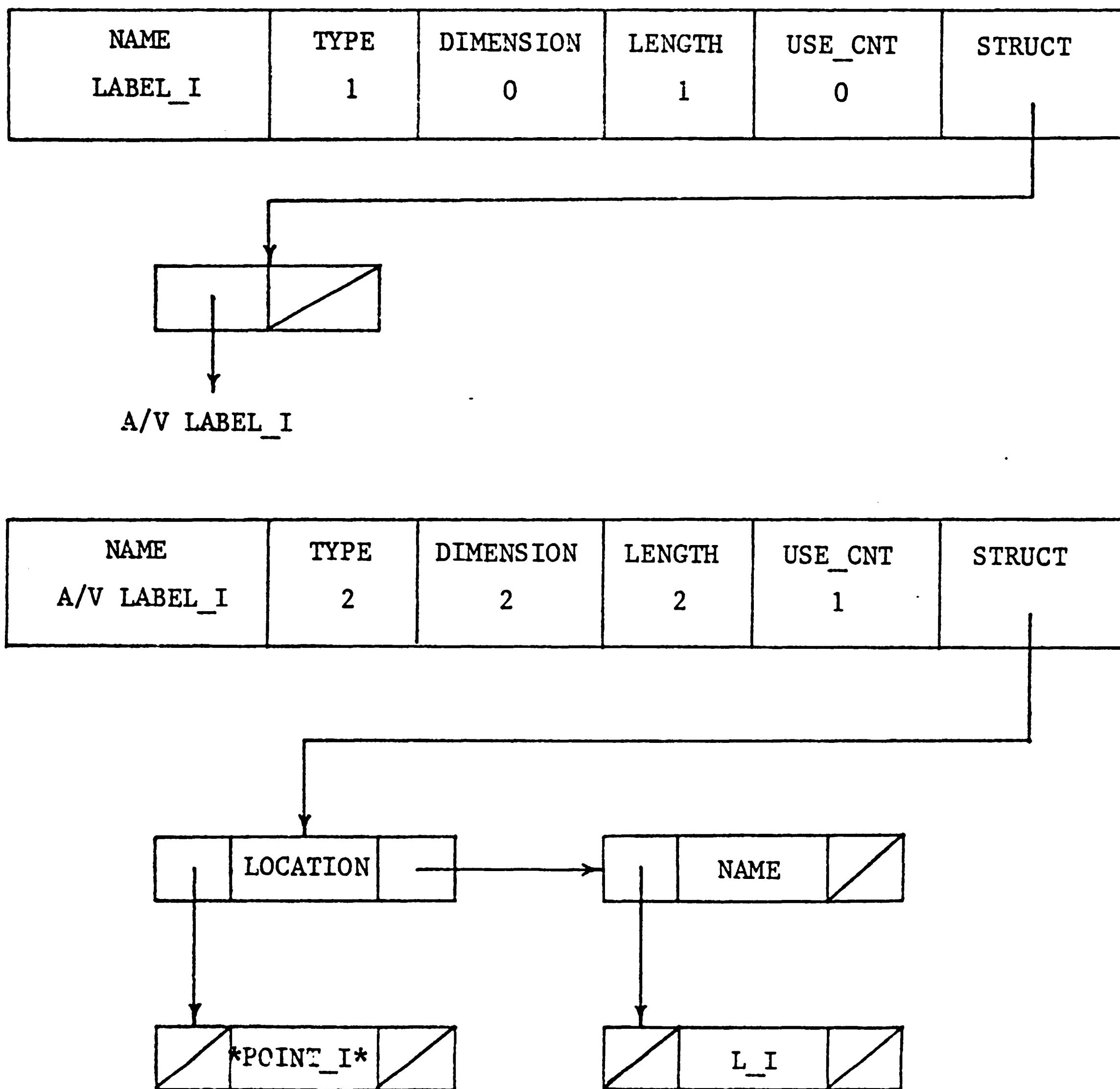


Figure 4.5 : Spatial Data Structure LABEL_I
and its relation.

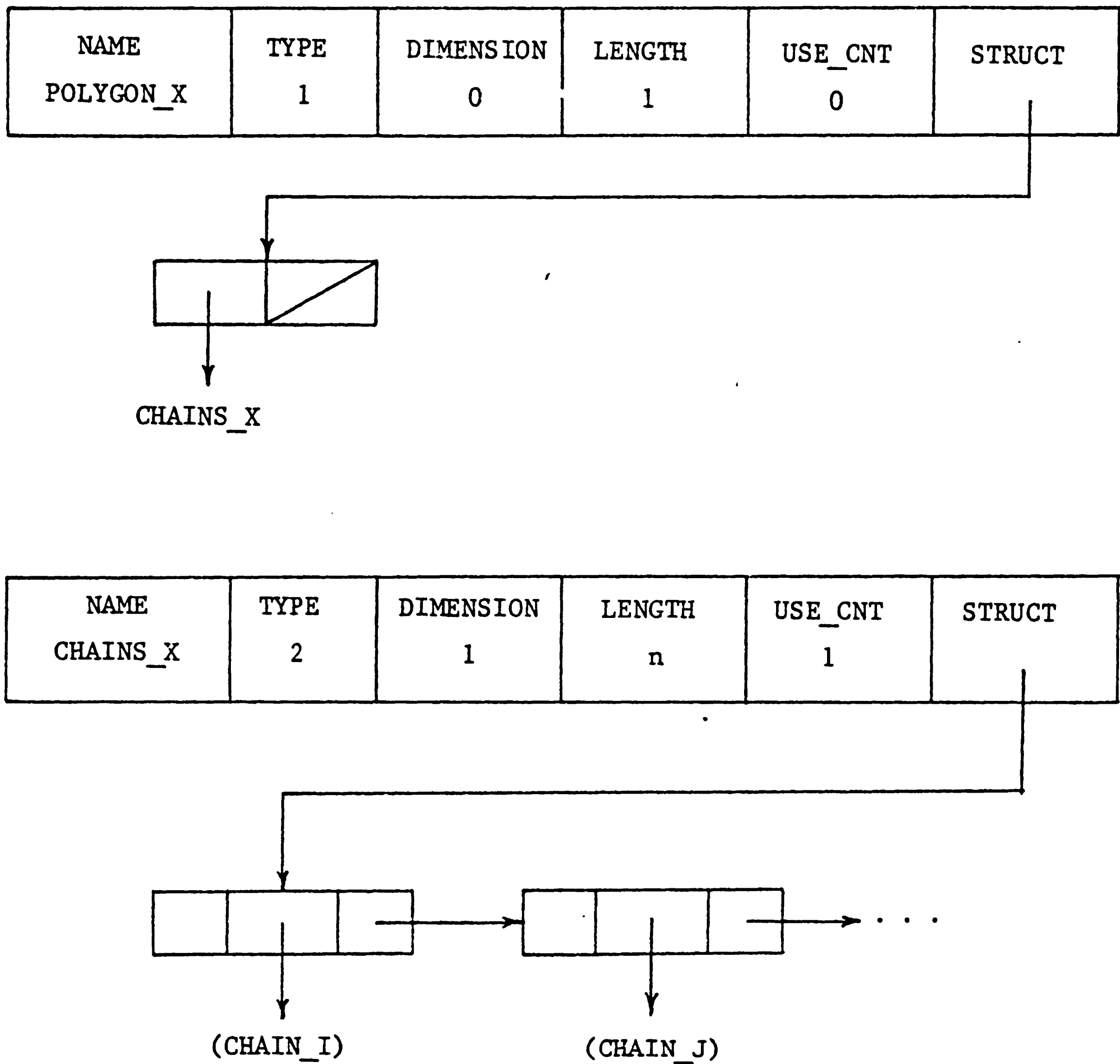


Figure 4.7 : Spatial Data Structure POLYGON_X and its relation.

structural representation of CHAIN_I. The relation list consists of two RELATION_CELLs which point to the two relations, A/V CHAIN_I and POINTS_I, associated with it.

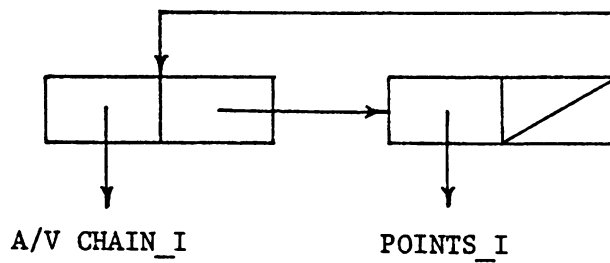
The A/V relation A/V CHAIN_I is a binary TREE relation. The values of both the attributes, LEFT and RIGHT, are spatial data structures of type REGION; and represent the regions to the left and the right of the chain, respectively.

The relation POINTS_I is a binary relation that represents an ordered list of points that define chain I. The N-tuples in this relation are stored in a list form. The first component of each N-tuple represents the X coordinate of the point, and the second component represents the Y coordinate.

4.2 IMPLEMENTATION OF INTERNAL PHYSICAL STRUCTURE

Although, the database resides on a secondary device, it is necessary to setup the entire database in memory first, before it is transfered to the secondary device. In this section we explain how this physical structure is implemented in memory.

NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
CHAIN_I	1	0	2	0	



NAME	TYPE	DIMENSION	LENGTH	USE_CNT	STRUCT
A/V CHAIN_I	2	2	2	1	

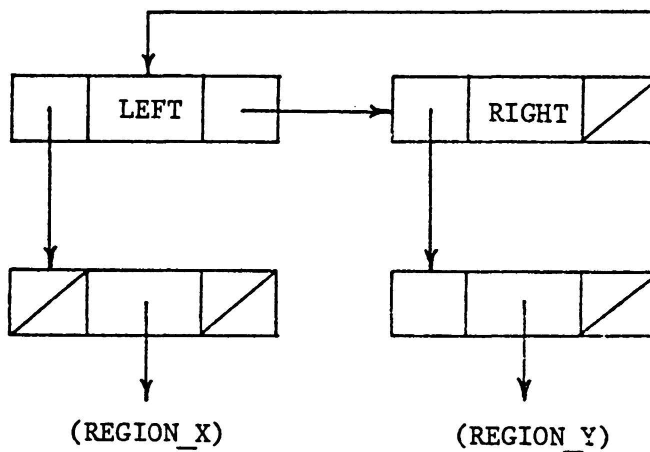


Figure 4.8 : Spatial Data Structure CHAIN_I and its relations.

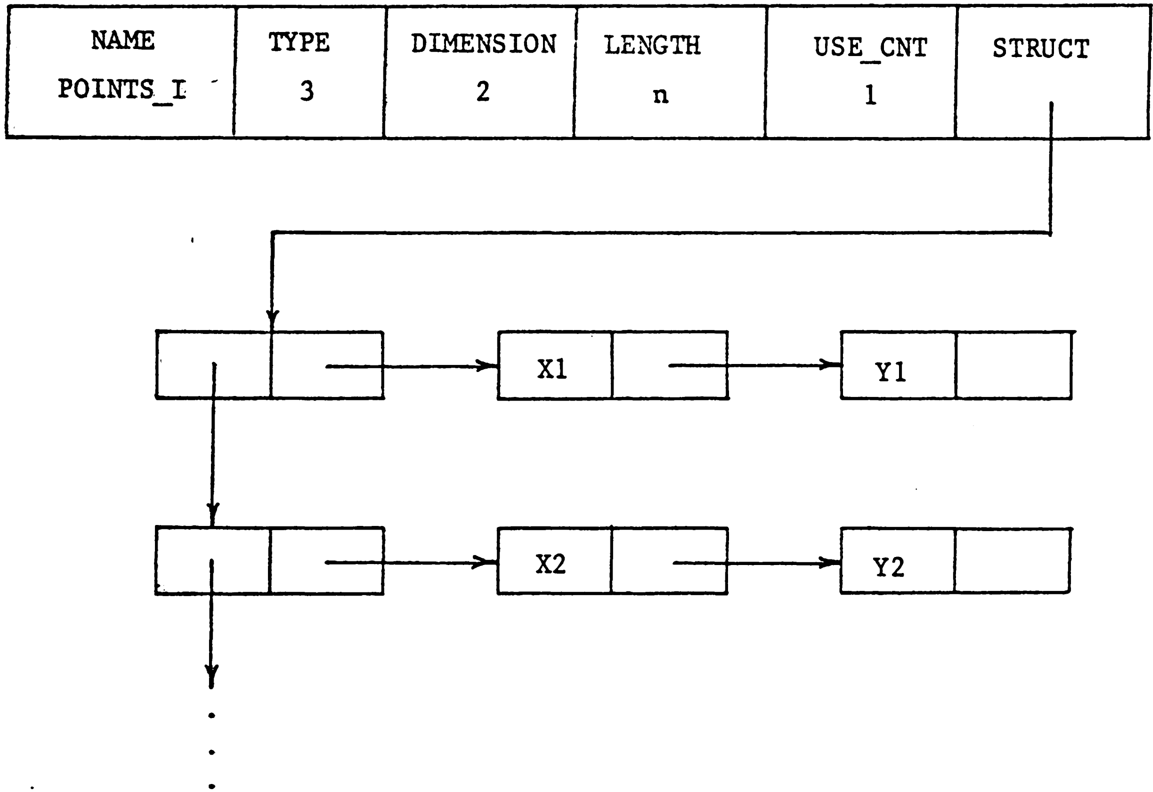


Figure 4.8 : continued..

The available geographic data, consisting of watershed areas, streams, labels, and roads (refer to Chapter II), is read from a data file into memory and structured according to the structures described in the preceding section.

The following primitive operations are used to set up this internal physical structure of the database.

1) Create a header for a spatial data structure.
2) Create a header for a relation.
3) Search the REL_DICTIONARY for a given relation name.
4) Search the RDS_DICTIONARY for a given SDS name.
5) Attach a relation to a spatial data structure.
6) Insert an N-tuple in a relation.
7) Catalog a spatial data structure in the RDS_DICTIONARY.
8) Catalog a relation in the REL_DICTIONARY.

While creating a spatial data structure, its name is read from the data file, and a header is created with the specified name. The TYPE field takes the value of one, and the DIMENSION field takes the value of zero. The LENGTH and USE_CNT fields are initialized to zero. The tag of the STRUCT field is set to RC_ELEM and its value initialized to nil. This new spatial data structure is then added to the RDS_DICTIONARY. Each relation associated with that spatial data structure is then created and attached to the spatial

data structure. When a relation is added to a spatial data structure, a relation cell, which points to the header of that relation, is linked into the list pointed to by the STRUCT field of the spatial data structure header. The new relations are always added at the begining of the relation list. Also, when a relation is attached to a spatial data structure, both the LENGTH field of the spatial data structure header and the USE_CNT field of the relation header are incremented by one.

When a new relation is created, its name is read from the data file and its header is created with the specified name. The TYPE (two for the TREE relation and three for the LIST relation) and the DIMENSION are also specified while creating the new header. The LENGTH and the USE_CNT fields are initialized to zero. The tag of the STRUCT field is set to either TREE_ELEM or LIST_ELEM depending on the type of the relation, and its value is initialized to nil.

The N-tuples are then read one by one from the data file and are inserted into the relation. In our system, each component of the N-tuple is represented by a structure called STACK_ELEM (refer to chapter 4). It is therefore necessary to set the tag field of each component of the N-tuple to an appropriate type. In the TREE relations, the N-tuples are

stored in a lexigraphical order of the type of their first component. If the two data types are the same, the ordering is based on their values. For the N-tuples having the same first m components, m = 1, ..., N-1; the ordering depends on their (m+1) st component; ordered first on their types and then on their values, if the types are same. Also, duplicates are omitted in TREE relations. The order of the N-tuples is prespecified in LIST relations.

While reading the N-tuples, if a component which is a spatial data structure is encountered, a binary search is performed in the RDS_DICTIONARY to find out whether or not that particular spatial data structure has been already created. If the spatial data structure already exists in memory, then a pointer to its header is obtained from the RDS_DICTIONARY. If it does not exist, then a new spatial data structure is created and a pointer to its header is returned as the component in that N-tuple. As in the case of a spatial data structure, whenever a new relation is created, an entry is made for that relation in the REL_DICTIONARY.

In this fashion all the spatial data structures and the relations associted with them are created and structured physically in memory.

4.3 THE PHYSICAL STRUCTURE ON THE SECONDARY DEVICE

The physical structures defined in the preceding sections are suitable for the internal manipulations. However, it is not feasible to set up these physical structures in memory, everytime the database is loaded. Also a real geographic system would be much too large to load the entire systm into memory at one time. Our experimental geographic system therefore resides on a secondary storage and parts of it are retrieved as necessary. In this section we describe the structure of the database on the secondary device and its implementation. The retrieval of this information into memory will be explained later in the succeeding chapters.

The objective of this experimental geographic system is to design structures for faster query access and efficient utilization of the storage in memory. That is, the stress is given on finding out efficient structures for the internal manipulations, and the issue of optimal record and file organization on the seconadary device is ignored[8].

---

8) The continuation of this work involves a design of a real geographic information system in which a study will be made to determine the optimal record and file organization to provide both the fast query response and the efficient storage utilization.

The physical structure of the database on the secondary storage device, which is a disk, is simple and straight forward; no elaborate record or file structure is used. Each spatial data structure and relation in the system is stored in a separate file on the disk. The corresponding names of the spatial data structures and the relations are used as the file names. Another possible file structure would be to store all the spatial data structures and relations in one file, which is divided into a number of segments having variable length, with each spatial data structure and relation occupying one segment. However, this later file organization was not used in order to avoid the complexity and overhead of memory management on the disk[9]. The advantage of keeping each structure in a separate file is that it is easier to fetch the entire structure of the SDS and the relation into memory at one time. Also when any SDS or relation is transfered back to the disk, the VAX/VMS system (on which this information system is implemented) creates a new file with the same name but with a higher version number. The files with the older version number can be deleted from the disk by simple commands. This avoids the problem of memory management on the disk.

----

9) Since this is an experimental system, a decision was made to keep the file organization simple and to let the system handle the memory management.

The data is stored sequentially within a file, and no indexed structure is used. Each data element occupies one record on a file. Since an indexed structure is not used, there is no efficient way in which any particular record can be accessed, except for sequential access or a binary search by its content. However, in our system it is not necessary to fetch any particular record in order to perform any operation on the data stored in it, because we retrieve the entire structure of the SDS and the relation; and not an individual record (This will be explained in details in the succeeding chapter). The advantage of using the sequential file organization is that the data can be stored in that order which best facilitates the forming of the internal physical structure when the SDS or the relation is retrieved and returned to memory.

A spatial data structure is stored on a disk as follows: The first five records contain the information from the spatial data structure header. The first record contains the name of the spatial data structure which is a character string. The next four records store the TYPE, DIMENSION, LENGTH, and the USE_CNT respectively; whose values are integers. Next, the names of the relations associated with the corresponding spatial data structure are stored sequentially, with one relation name in each record. For example,

a spatial data structure REGION_X is stored on the disk as follows:

REGION_X

1

0

4

0

A/V REGION_X

REGION ADJACENCY_X

STREAM NETWORK_X

LABELS_X

Similar to a spatial data structure, the first five records in a file containing a relation store the information from the relation header; that is, the first five records store the NAME, TYPE, DIMENSION, LENGTH, and USE_CNT of the relation, respectively. The N-tuples are then stored sequentially in the order in which they are structured physically in the relation. Each record contains only one component. Since a component of an N-tuple is represented by the structure STACK_ELEM (refer to Chapter III), it is necessary to store the type of the component along with its value. If the type of the component were not stored, then it would not have been possible to set the tag of the component when the relation is retrieved in memory. A component of an

N-tuple could be a pointer to another structure, as in the case of a character string or a spatial data structure. In such cases the actual data pointed to by the pointer is stored. For example, if the component is a pointer to a CHAR_STRING_NODE, then the actual character string data stored in its C_S field is stored; if the component is a pointer to a spatial data structure header, then the name of the spatial data structure is stored. The physical structure of both types of relations (TREE and LIST) is the same. The physical structure of the TREE relation A/V REGION_X is as shown below:

A/V REGION_X

2

2

4

1

CHAR -- AREA

INT -- 12345

CHAR -- BOUNDARY

RDS -- POLYGON_X

CHAR -- NAME

CHAR -- X

CHAR -- PARENT

RDS -- REGION_Y

At setup time, once the entire database structure is implemented in memory, it is transfered to the secondary device, that is to the disk. Initially, two new files are created on the disk and both the dictionaries, RDS_DICTIONARY and REL_DICTIONARY, are written onto those files. Only the RDS_ARRAY and the REL_ARRAY from those dictionaries are stored on the files, that is, only the names of the spatial data structures and the relations are stored. These names are stored sequentially on the files. For example, a portion of the file containing RDS_DICTIONARY is as follows:

```
---
REGION_F1
REGION_F2
REGION_F3
REGION_N3
---
```

These two dictionaries are not deleted from memory and are used to access each spatial data structure and relation in the system, in order to transfer them on the disk.

The RDS_DICTIONARY is then scanned sequentially and each spatial data structure entered into it is transfered to the disk. Initially a new file is created on the disk with the name of the spatial data structure as the file name. The

header record (an RDS) is then stored in that file as described above. The RELATION LIST is next traversed and the names of the relations pointed to by the RELATION_CELLs are stored in the file.

Similarly, the REL_DICTIONARY is scanned and the relations are transfered to the disk, one at a time. The same procedure is followed as in case of the spatial data structres. First, a new file is created with the corresponding name of the relation, and the header is stored in that file. The N-tuples in the relation are then accessed one by one, in the same order in which they are stored in the relation and are written onto the file. Thus the entire database is set up on the disk.

Chapter V

THE QUERY LANGUAGE INTERPRETER

In this chapter we describe the query language interpreter through which the user can interact with the database system. This query language provides an access to the information in the database via retrieval and storage operations. The credit for the design and implementation of this query language interpreter[10] (from here onwards we will refer to it as just 'the interpreter') goes to Gary Minden and his felow students from Kansas University. A major portion of this chapter was taken directly from the documentation [MINDG81] of this interpreter .

The query language is based on the language called FORTH[11] designed by Charles H. Moore. Our query language processor is designed for an interactive purpose, and a facility to use it in batch environment is not yet provided. By default, a video terminal is used as an input-output device, however, using proper commands the input-output can be performed from other text files. The interpreter designed for this query language is stack oriented and works like a calculator. The user can use a remote terminal and

---

10) This interpreter was modified and new commands were added to it in order to use it as a query language for our information system.
11) For more details about the FORTH language, please refer to [MOORC74].

communicate with the database by using a sequence of commands defined in the query language. The language has a vocabulary which defines these commands. The user can define constants, variables, and arrays; he or she can perform arithmetic and database operations, such as storage, retrieval, update, etc., using these commands. Also, the interpreter is equipped with control structures such as if-then-else, do-loop, repeat-until. In addition, the user can define his own commands, which may be a concatenation of several existing and/or user-defined commands.

The interpreter is stack oriented in the sense that most operations communicate only through a stack. The arguments of a primitive operation are first pushed onto the stack, and then the primitive operation is invoked. The results of the operation are returned to the stack. This language uses a postfix notation (also called reverse Polish notation) for all the operations.

The interpreter has a vocabulary represented by a VOCABULARY_ARRAY (to be explained later), which stores all the query language commands. The new commands and the constants, variables, and arrays, along with their values, are added to the vocabulary as they are defined by the user. The succeeding sections describe in detail, how the information is stored in the vocabulary.

The interpreter operates in one of the following two modes: EXECUTE and COMPILE. In EXECUTE mode it executes the command input by the user, while in COMPILE mode the interpreter compiles the text input by the user and adds it to the vocabulary. By default, the interpreter operates in EXECUTE mode. The mode of the interpreter is changed from EXECUTE to COMPILE and vice versa by certain commands in the query language itself.

Each command in the system has a precedence value associated with it. The precedence value of a command can be either zero, one, or two. Similarly, the interpreter has a state value which determines whether it is in EXECUTE or COMPILE mode. The state of the interpreter is changed (either increased or decreased) by certain commands in the language. Initially, the interpreter in invoked with a state zero and remains in that state until a command is invoked which raises the state value. Every time a command is invoked, its precedence is compared with the current state value of the interpreter. If the precedence of the command is greater than or equal to the current state of the interpreter, the interpreter remains in the EXECUTE mode and the command is executed. Otherwise, its mode is changed to COMPILE in which the text input by the user is compiled into the vocabulary. The interpreter remains in the COMPILE mode

until a command is encountered which reduces the state of the interpreter and puts it back into EXECUTE mode.

Each command in the query language is implemented as a Pascal routine. The entire interpreter is implemented using a CASE control structure [JENSK74] in Pascal. The CASE statement consists of code for every command in the system. The corresponding Pascal code is executed when any command is invoked. Each command has a procedure element called PROC_ELEM (to be explained later) associated with it. This procedure element corresponds to the CASE LABEL in the Pascal CASE statement. The procedure element of the current command is used as a CASE SELECTOR to execute the appropriate code.

Before we continue our discussion on the interpreter, we need to describe the data structures used by the interpreter and the representations of the data types.

5.1 THE DATA STRUCTURES FOR THE INTERPRETER

The interpreter supports and utilizes all the data structures described in Chapter IV. The additional data structures designed solely for the purpose of the interpreter and the data types whose representations are different than those described in Chapter IV are described here.

5.1.1 STACK_ELEM

The structure of our basic element, STACK_ELEM, is extended to include the elements of type PROC_ELEM, VOCB_ELEM, DICT_ELEM_0, DICT_ELEM_1, and DICT_ELEM_2. A data element of type PROC_ELEM has an integer value which represents the process element value associated with any primitive in the language. The process element represents a CASE LABEL in the CASE statement memorysponding to any primitive in the language. A data element of type VOCB_ELEM also has an integer value and represents an index to any element in the vocabulary. An element of type DICT_ELEM_0, DICT_ELEM_1, or DICT_ELEM_2 is a pointer to a CHAR_STRING_NODE which stores the actual interpreter command. The type of the command is determined by its precedence value (DICT_ELEM_0 for precedence zero, DICT_ELEM_1 for precedence one, and DICT_ELEM_2 for precedence two). The Figure 5.1 illustrates the Pascal-like declaration of the extended STACK_ELEM structure (refer to Figure 3.1 for the original Pascal-like declaration of STACK_ELEM).

5.1.2 INTEGERS

Integers are represented by the user as character strings of decimal digits with a plus and/or a minus sign. The plus sign is optional. The interpreter scans the integer charac-

```
STACK_ELEM = record
               case SE_TAG : ELEM_TYPE of
                 RDS_ELEM   : (SE_RDS   : RDS_PTR);
                 REL_ELEM   : (SE_REL   : RELATION_PTR);
                 LIST_ELEM  : (SE_LIST  : LIST_CELL_PTR);
                 CHAR_ELEM  : (SE_CHAR  : CHAR_STRING_PTR);
                 INT_ELEM   : (SE_INT   : integer);
                 REAL_ELEM  : (SE_REAL  : real);
                 POINT_ELEM: (SE_POINT: POINT_CELL_PTR);
                 TREE_ELEM  : (SE_TREE  : TREE_CELL_PTR);
                 RC_ELEM    : (SE_RC    : RELATION_CELL_PTR);
                 PROC_ELEM  : (SE_PROC  : integer);
                 VOCB_ELEM  : (SE_VOCB  : integer);
                 DICT_ELEM_0, DICT_ELEM_1, DICT_ELEM_2
                            : (SE_DICT  : CHAR_STRING_PTR);
               end;
```

Figure 5.1 : STACK_ELEM EXTENDED.

ter string and converts it to binary for internal representation. All the integer manipulations are performed in binary.

5.1.3 BOOLEAN

Boolean values are used in several places to determine the program flow. Internally, the system uses the integer data type to represent the boolean value. A value zero is equivalent to the boolean value 'false'. A value not equal to zero (usually one) is equivalent to the value 'true'. The boolean values can not be stored explicitely in other data structures; however, their integer representations can be stored.

5.1.4 REALS

The interpreter does not support the data type REAL.

5.1.5 VOCABULARY ARRAY

The VOCABULARY_ARRAY represents the vocabulary of the query language interpreter. It is an internal array of STACK_ELEMs holding definitions for the query language and user defined commands, constants, variables, and arrays. The maximum size of the VOCABULARY_ARRAY is specified when the Pascal program is compiled.

Initially, when the interpreter is invoked the built-in commands in the query language are read from a data file along with their procedure elements and precedence values, and are stored in the VOCABULARY_ARRAY. Each command in the system is stored in the vocabulary as follows. The tag of the next available element of the VOCABULARY_ARRAY is set to either DICT_ELEM_0, DICT_ELEM_1, or DICT_ELEM_2 depending on the precedence value of the command, either zero, one, or two respectively. The value of this array element is a pointer to a CHARACTER_STRING_NODE which contains the command. The tag of the next element in the array is set to PROC_ELEM and its value is set to the procedure element value associated with the command. The definition of the command is terminated by a reference to a query language command END_DEF (to be explained later). The next element in the VOCABULARY_ARRAY contains a reference to the command END_DEF. Its tag is set to PROC_ELEM, and it has a value equal to the procedure element value of the command END_DEF. Figure 5.2 illustrates an entry in the VOCABULARY_ARRAY for the built-in command FIND which has a precedence zero and a procedure element value 82.

All the user defined commands are stored in the vocabulary as follows. Initially, a reference to a CHAR_STRING_NODE which contains the name of the command is

TAG	VALUE	
---	---	
DICT_ELEM_0	--------->	'FIND'
PROC_ELEM	82	
PROC_ELEM	15	(REF. TO END_DEF)

Figure 5.2 : ENRTY FOR THE COMMAND 'FIND'

stored in the VOCABULARY_ARRAY, with the tag of that particular element set to DICT_ELEM_0 (all the user defined commands have the lowest precedence which is zero). The new command is created using a built-in command DEFINITION (to be explained later) which raises the state value of the interpreter. All the following text, input by the user, is compiled and stored in the vocabulary until a command END_DEF is encountered which again lowers the state value of the interpreter and puts it back into EXECUTE mode. Thus the array elements following the reference to the name of the new command contain references to the text, input by the user. They might be references to character or integer data; references to predefined constants or variables; pointers to spatial data structures, relations or any other structures supported in the system; or references to the existing commands in the vocabulary (these commands might be the built-in commands in the system or the user-defined commands). Figure 5.3 illustrates an entry in the vocabulary for the user defined command TEST, defined as follows: " TEST" DEFINITION X Y + END_DEF

The constants and the variables defined by the user are entered in the OCABULARY_ARRAY in a way similar to the built-in commands. The only difference is that instead of the procedure element values in case of built-in commands,

The command is " TEST" DEFINITION X Y + END_DEF where DEFINITION, +, and END_DEF are built-in language commands, and X and Y are predefined variables.

TAG	VALUE	EXPLANATION
DICT_ELEM_0	" TEST "	THE COMMAND. PRECEDENCE 0.
VOCB_ELEM	<INDEX OF X>	REFERENCE TO THE VARIABLE X.
VOCB_ELEM	<INDEX OF Y>	REFERENCE TO THE VARIABLE Y.
PROC_ELEM	<INDEX OF +>	REFERENCE TO THE COMMAND +
PROC_ELEM	<INDEX OF END_DEF>	REFERENCE TO THE COMMAND END_DEF

Figure 5.3 : VOCABULARY ENTRY FOR USER DEFINED COMMAND 'TEST'.

the values of constants or variables are stored. The constants and variables are also terminated by a reference to the built-in command END_DEF.

5.1.6 STACK

The STACK is an internal array of STACK_ELEMs, used to hold the arguments and results for processing. The STACK supports all the data structures. The data are pushed onto the STACK by referencing them. Referencing an integer by entering it in the input stream pushes the binary value on the STACK. Naming a defined spatial data structure, pushes a pointer to its header onto the STACK. The commands for the stack manipulation operations are described in later sections.

All the other data structures and their representations are the same as described in Chapter IV.

5.2 THE INTERPRETER ALGORITHM

In this section we briefly explain the interpreter algorithm.

Initially, the interpreter is invoked with its state set to zero. The input from the terminal is scanned and the string entered by the user, which is delimited by a blank,

is stored in a CHAR_STRING_NODE. A binary search is then carried out in the RDS_DICTIONARY to check if the given string is the name of any spatial data structure in the system. If not, a search is performed in the REL_DICTIONARY to find if it is a relation name. If a name is found in any one of the above dictionaries, then a pointer to the corresponding structure, either a spatial data structure header or a relation header, is returned to the stack. If both the searches fail, then the vocabulary is searched to find out if the input string is a command defined in the query language. If the string is not a valid command, it is checked to determine whether or not it is a valid integer. If it is an integer, then the character string is converted into the binary integer and pushed onto the stack. A failure of this test causes an error message to be printed on the terminal.

If the input string is a valid command in the language, and has a precedence greater than or equal to the current interpreter state, then the command gets executed. It should be noted that initially the state of the interpreter is zero, and there are certain commands in the system which raise the state of the interpreter. If the precedence value of the command is lower, then the text following the command is compiled into the vocabulary until an another command is encountered which lowers the state of the interpreter and puts it back into execute mode.

5.3 THE INTERPRETER PRIMITIVES

As described earlier, each primitive operation in the interpreter is implemented as a Pascal routine. The arguments to a primitive are first placed onto the stack before invoking the primitive. The results of the primitive are similarly returned to the stack and can be used as the arguments to the succeeding primitives. For this discussion, the primitives are divided into five categories: stack manipulation, program control, vocabulary manipulation, data structure processing, and miscelleneous. The following notation is adopted for the description of each primitive.

<DS> indicates a data structure of any type, that is, a STACK_ELEM. A data structure consists of a tag field and a value field. Where necessary, this is indicated by a pair, (<TAG>,<VALUE>).

<DSn> indicates an n'th <DS>.

<INT> indicates an integer data structure.

<INTn> indicates an n'th <INT>.

<T/F> indicates a boolean data structure.

<STRING> indicates a character string.

<RDS> indicates a pointer to a spatial data structure header.

<REL> indicates a pointer to a relation header.

NAME(<RDS>/<REL>) indicates the name of an SDS or relation.

TYPE(<RDS>/<REL>)	indicates the type of an SDS or relation.
DIMEN(<RDS>/<REL>)	indicates the dimension of an SDS or a relation.
LENGTH(<RDS> or <REL>)	indicates the length of an SDS or a relation.
USE_CNT(<RDS> or <REL>)	indicates the use count of an SDS or a relation.
<PTR>	indicates a pointer of any type.
VOCABULARY[<INT>]	indicates the contents of the VOCABULARY_ARRAY at location <INT>.
RDS_DICT [<INT>]	indicates the contents of the RDS_DICTIONARY at location <INT>.
REL_DICT [<INT>]	indicates the contents of the REL_DICTIONARY at location <INT>.
STACK [<INT>]	indicates the stack entry at location <INT>.
SP	indicates the current top of stack pointer.
TOS	indicates the current top of stack item, that is, it is equivalent to STACK [SP].
NOS	indicates the next item below TOS on the stack, that is, STACK [SP-1].
---	indicates the contents of the stack that are not affected by the processing described.

VP	indicates the current top of vocabulary.

The format in which the primitives are described is as follows. First, the function of the command is explained. Next, the stack contents before and after invoking the command are described. It should be noted here that the command itself is not entered into the stack, but is used in our notation as a separator between the stack contents before and after its execution. Following the stack contents, the syntax of the command and the actual operations that take place when the command is invoked are described.

5.3.1 STACK MANIPULATION PRIMITIVES

The primitives described in this section are general purpose primitives for manipulating the stack and doing simple arithmetic operations. Within this section the words are grouped according to general usage.

5.3.1.1 +

+	Add top two integers on the stack.
Stack Contents:	<INT1> <INT2> + <INT1+INT2>
Syntax:	<INT1> <INT2> +
Operation:	STACK[SP-1] := STACK[SP] + STACK[SP-1]
	SP := SP - 1

This command adds the top two integers on the stack. If one of the items is not of integer type, an error message is printed and the stack is not modified.

5.3.1.2 -

-	subtract top two integers on the stack.
Stack Contents:	<INT1> <INT2> - <INT1-INT2>
Syntax:	<INT1> <INT2> -
Operation:	STACK[SP-1] := STACK[SP-1] - STACK[SP]
	SP := SP-1

This words subtracts TOS from NOS and leaves the result on the stack. If one of the items is not of integer type, an error message is printed and the stack is not modified.

5.3.1.3 *

*	Multiply top two integers on the stack.
Stack operations:	<INT1> <INT2> * <INT1*INT2>
Syntax:	<INT1> <INT2> *
Operation:	STACK[SP-1] := STACK[SP] * STACK[SP-1]
	SP := SP-1

This command multiplies the top two integers on the stack. If one of the items is not an integer, an error message is printed and the stack is not modified.

5.3.1.4 /

/	Divide the top two integers on the stack.
Stack Contents:	<INT1> <INT2> / <INT1/INT2>
Syntax:	<INT1> <INT2> /
Operation:	STACK[SP-1] := STACK[SP-1] / STACK[SP] SP := SP-1

This command divides NOS by TOS and leaves the quotient on the stack. If one of the items is not of integer type, an error message is printed and the stack is not modified.

5.3.1.5 NOT

NOT	Complement the boolean value on the TOS.
Stack Contents:	<T/F> NOT <T/F>
Syntax:	<T/F> NOT
Operation:	if (TOS = 'true') then STACK[SP] := 'false' else STACK[SP] := 'true'

The TOS is interpreted as a boolean value and the complement of this value is returned on the TOS.

5.3.1.6 I, J, K, and L

I, J, K, and L — Push DO-+LOOP indices onto the stack.

Stack Contents: --- I <INT> (or J, K, or L)

Syntax: I (or J, K, or L)

Operation: SP := SP+1

STACK[SP] := current

The commands I, J, K, and L are used to obtain the current DO-+LOOP indices when within an iterative branch (see DO and +LOOP primitives). I always obtains the index of the inner most DO-+LOOP control structure. J obtains the index of the next outermost DO-+LOOP control structure, and so forth. Results are unpredictable if an outer index is referenced but the DO-+LOOPs are not nested to that level. For example, if DO-+LOOPs are nested two levels deep and a reference is made to K or L, the results are undefined.

5.3.1.7 RELATIONAL OPERATORS

=, -=, >=, <=, >, and < — Relational operators.

Stack Contents: <DS1> <DS2> ROP <T/F>

(where ROP is one of the above.)

Syntax: <DS1> <DS2> ROP

(where ROP is one of the above.)

Operation: if (NOS ROP TOS)

```
        then STACK[SP-1] := 'true'
        else STACK[SP-1] := 'false'
    SP := SP-1
```

These commands compare the top two data structures on the stack. Most of the data structure types are supported. If the data structures are in the relation specified, a boolean value of 'true' is returned, otherwise, a value of 'false' is returned. The comparison is made between the data types based on the following order: SDS < RELATION < LIST < STRING < POINT < INTEGER < REAL < MAP LIST < TREE. If the data types match, a comparison is made between the data values. For INTEGER, REAL, and STRING types the arithmetic or the character string comparison is made. For SDS and RELATION type the NAMEs of these data structures are compared as strings. The comparisons between LISTs, MAPs and TREEs are currently invalid.

5.3.1.8 RDS?, REL?, INT?, PROC?, CHAR? and DICT?

RDS?, REL?, INT? PROC?, CHAR?, and DICT?	Test the TOS data type.
Stack Contents:	<DS> TT <DS> <T/F> (where TT is one of the above.)
Syntax:	<DS> TT

(where TT is one of the above.)

Operation: SP := SP+1

```
if ( type(TOS) = TT )
   then STACK[SP] := 'true'
   else STACK[SP] := 'false'
```

These commands test the TOS for the indicated data type. If the type corresponds to the words used, a boolean value of 'true' is returned to TOS, otherwise, a 'false' value is returned. Note that the data structure is not removed from the stack and becomes the NOS after execution.

5.3.1.9 -NIL?

-NIL? Compare the TOS to the nil pointer.

Stack Contents: <PTR> -NIL? [<PTR>] <T/F>

Syntax: <PTR> -NIL?

Operation:

```
if ( TOS = nil )
   then STACK[SP] := 'false'
   else SP := SP + 1
        STACK[SP] := 'true'
```

The comand -NIL? compares the TOS to the nil pointer. If the TOS pointer is nil, the pointer is dropped and a boolean value of 'true' is returned. If the pointer is not nil, the pointer is retained and a value of 'false' is pushed onto

the stack. This command is normally used to determine the end of a linked data structure.

5.3.1.10 SWAP

SWAP	Swap positions of the top two items on the stack.
Stack Contents:	<DS1> <DS2> SWAP <DS2> <DS1>
Syntax:	<DS1> <DS2> SWAP
Operation:	TEMP := STACK[SP] STACK[SP] := STACK[SP-1] STACK[SP-1] := TEMP

The command SWAP changes the positions of the top two items on the stack. All the data types are supported.

5.3.1.11 DROP

DROP	Remove the top item from the stack.
Stack Contents:	<DS> DROP ---
Syntax:	<DS> DROP
Operation:	SP := SP-1

This command deletes the top item from the stack.

5.3.1.12 DUP

DUP Duplicate the TOS.

Stack Contents: <DS> DUP <DS> <DS>

Syntax: <DS> DUP

Operation: SP := SP+1

STACK[SP] := STACK[SP-1]

This command makes a copy of the TOS and pushes the copy onto the stack. All the data types are supported.

5.3.1.13 ROT

ROT Rotate the top three items on the stack.

Stack Contents: <DS1> <DS2> <DS3> ROT <DS2> <DS3> <DS1>

Syntax: <DS1> <DS2> <DS3> ROT

Operation: TEMP := STACK[SP-2]

STACK[SP-2] := STACK[SP-1]

STACK[SP-1] := STACK[SP]

STACK[SP] := TEMP

The command ROT moves the item below the NOS to TOS. Both the original TOS and NOS move down in the stack.

5.3.1.14 -ROT

-ROT	Reverse rotate the top three items on the stack.
Stack Contents:	<DS1> <DS2> <DS3> -ROT <DS3> <DS1> <DS2>
Syntax:	<DS1> <DS2> <DS3> -ROT
Operation:	TEMP := STACK[SP] STACK[SP] := STACK[SP-1] STACK[SP-1] := STACK[SP-2] STACK[SP-2] := TEMP

This command is a complement of ROT. The TOS is placed below the original NOS, the original item below the NOS moves to NOS, and the original NOS becomes the TOS.

5.3.1.15 OVER

OVER	Copy NOS and push it onto the stack.
Stack Oprations:	<DS1> <DS2> OVER <DS1> <DS2> <DS1>
Syntax:	<DS1> <DS2> OVER
Operation:	SP := SP+1 STACK[SP] := STACK[SP-2]

This command makes a copy of the NOS and pushes it onto the stack to become the new TOS. All the data types are supported.

5.3.1.16 #DROP

#DROP	Remove the indicated number of items from the stack.
Stack Contents:	<DSn> --- <DS1> <INT> #DROP ---
Syntax:	<DSn> --- <DS1> <number of items to drop> #DROP
Operation:	SP := SP-STACK[SP]-1

#DROP removes the number of items indicated by the TOS, but not counting the TOS, from the stack. Thus several DROP operations can be performed by using this command.

5.3.1.17 #ROT

#ROT	Rotate top n items on the stack.
Stack Contents:	<DSn> ... <DS1> <INT> #ROT <DSn-1> ... <DS1> <DSn>
Syntax:	<DSn> ... <DS1> <number of items to rotate> #ROT
Operation:	TEMP := STACK[SP-<INT>] STACK[SP-<INT>] := STACK[SP-<INT>+1] STACK[SP-<INT>+1] := STACK[SP-<INT>+2] -- -- STACK[SP-1] := TEMP SP := SP-1

This command rotates the top n items of the stack, but not including the TOS which specifies n. The item at SP-<INT> becomes the new TOS.

5.3.1.18 #-ROT

#-ROT Reverse rotate top n items on the stack.

Stack Contents: <DSn> ... <DS1> <INT> #-ROT <DS1> <DSn> ... <DS2>

Syntax: <DSn> ... <DS1> <number of items to rotate> #-ROT

Operation:

```
TEMP := STACK[SP-1]
STACK[SP-1] := STACK[SP-2]
STACK[SP-2] := STACK[SP-3]
              --
              --
STACK[SP-<INT>] := TEMP
SP := SP-1
```

This command reverse rotates the top n items of the stack, but not including the TOS which specifies the value n. The item at SP-1 becomes the new TOS and the original NOS is positioned at SP-<INT>.

5.3.1.19 .

.	Print the TOS.
Stack Contents:	<DS> ---
Syntax:	<DS to output> .
Operation:	WRITE (STACK[SP])
	SP := SP-1

This word outputs the TOS to the current output file and removes the TOS from the stack. For INTEGERs, REALs, and STRINGS their values are printed. For SDSs and RELATIONs, their names are output. For MAPs, POINTs, LISTs and TREEs, only the corresponding data type is printed.

5.3.1.20 .STACK

.STACK	Output the entire stack.
Stack Contents:	--- .STACK ---
Syntax:	--- .STACK
Operation:	for I := SP downto 0 do
	WRITE (STACK[I])

The contents of the entire stack are output to the current output file. No modification of the stack takes place.

5.3.1.21 CR (CARRIAGE RETURN)

CR	Output an end of line.
Stack Contents:	--- CR ---
Syntax:	CR
Operation:	WRITELN

This word outputs an end of line (normally CARRAIGE RETURN and LINE FEED) to the current output file. No modification of the stack takes place.

5.3.1.22 (STACK)

(STACK)	Obtain the contents of the indexed location on the stack.
Stack Contents:	--- <INT> (STACK) STACK[<INT>]
Syntax:	<index in the stack> (STACK)
Operation:	STACK[SP] := STACK[SP-<INT>-1]

This command uses the TOS as an index into the stack. The TOS has an index -1 and the NOS has an index 0. The TOS is replaced with the contents of the stack location indexed. This command is convenient for fetching data deep in the stack.

5.3.2 PROGRAM CONTROL PRIMITIVES

The primitives described in this section control the sequence of program execution. There are four basic control structures: sequential, conditional, loop, and repeat-until. Note that there are no labels and that it is not possible to use a GOTO statement in this programming system. The sequential execution is the normal mode and needs no further explanation. The control strutures are limited in how they may be combined. Essentially, these control structures determine whether portions of a program should be executed or not. We call these portions the 'branches'. In general, a control structure must occur entirely within a branch of another control structure. For example, the following is an illegal structure because the DO-+LOOP does not occur entirely within a branch of the IF-THEN-ELSE construct (all the individual structures are explained later).

IF --- DO --- ELSE --- +LOOP --- THEN

The following is a valid control structure.

IF --- DO --- +LOOP --- ELSE --- IF --- ELSE --- THEN --- THEN

Now, we describe various program control primitives.

5.3.2.1 IF

IF — Perform a conditional branch based on a boolean value.

Stack Contents: <T/F> IF ---

Syntax: <T/F> IF --- <'true' branch> ---
[ELSE --- <'false' branch> ---] THEN

Operation: if (TOS = 'true')
  then SP:=SP-1; <execute 'true' branch>
  else SP:=SP-1; <execute 'false' branch>
SP:=SP-1

The command IF tests the TOS and if it has the boolean value 'true', the code between the IF and ELSE or THEN (if there is no ELSE) is executed. When the TOS has the value 'false', the code following the ELSE, if it exists, is executed. The IF-ELSE-THEN structures may be nested as long as the entire structure occurs entirely within one branch or the other.

5.3.2.2 ELSE

ELSE — Mark the end of the 'true' branch in a conditional structure.

Stack Contents: --- ELSE ---

Syntax: (See IF command above)

The command ELSE denotes the end of the 'true' branch in a conditional control structure and the begining of the 'false' branch. If the 'true' branch is taken by the corresponding IF, execution continues following the correponding THEN. No stack operations are done by ELSE.

5.3.2.3 THEN

THEN — Terminate an IF-ELSE-THEN structure.

Stack Contents: --- THEN ---

Syntax: (See the command IF above)

The command THEN terminates an IF-ELSE-THEN control structure. No stack operations are performed.

5.3.2.4 DO

DO — Begin an iterative control structure.

Stack Contents: <INT> <INT> DO --- <INT> +LOOP ---

Syntax: <upper limit> <initial index> DO
--- <iterative branch> ---
<index increment> +LOOP

Operation:
```
if ( TOS < NOS )
  then SP:=SP-2;<execute iterative branch>
  else SP:=SP-2;<continue execution
       following corresponding +LOOP>
```

The command DO initiates an iterative control structure. If the TOS is less than NOS, the branch between the DO and the corresponding +LOOP (the command +LOOP and the index increment will be explained later) is executed. Otherwise, execution continues following the +LOOP. Note that DO performs its test before the iterative branch is executed. Hence, if TOS is greater than or equal to NOS, the execution continues immediately following the +LOOP. DO-+LOOPs may be nested to at the most five levels. If DO-+LOOPs are nested, an entire DO-+LOOP structure must be within one branch.

For example, a Pascal-like loop

```
for I := 1 to 10 do begin
   ---
   ---
end;
```

will be implemented as follows:

INPUT	EXPLANATION
1	The lower bound of the loop.
10	The upper bound of the loop.
DO	The starting of the loop..
---	The text within the loop.
---	The text within the loop.
1	The increment of the loop.
+LOOP	The Termination of the loop.

5.3.2.5 +LOOP

+LOOP — Terminate iterative branch and increment the index.

Stack Contents: <INT> +LOOP ---

Syntax: <index increment> +LOOP (See DO above)

Operation: SP:=SP+1

STACK[SP] := <inner most DO-+LOOP index>
+ NOS

STACK[SP-1] := <upper limit for DO-+LOOP>

<goto corresponding +DO>

The command +LOOP terminates an iterative control structure. During the execution, the TOS is added to the current DO-+LOOP index. The upper limit and the index are pushed onto the stack in the format that DO expects them. As described above, DO then decides whether to continue execution of the iterative branch or to pass control to the code following the +LOOP.

5.3.2.6 REPEAT

REPEAT — Mark the begining of a REPEAT-UNTIL control structure.

Stack Contents: --- REPEAT ---

Syntax: --- REPEAT --- <T/F> UNTIL ---

The command REPEAT marks the begining of a REPEAT-UNTIL control structure. Execution continues until the corresponding UNTIL is encountered at which time, the TOS is tested. If the TOS has the boolean value of 'false', control returns to the corresponding REPEAT. Otherwise, it continues following the UNTIL. REPEAT does not modify the stack nor does it make an entry into the vocabulary.

5.3.2.7 UNTIL

UNTIL — Mark the end of a REPEAT-UNTIL structure.

Stack Contents: <T/F> UNTIL ---

Syntax: (See REPEAT above)

Operation:

```
if ( TOS = 'false' )
  then <return to corresponding REPEAT>
  else <continue execution following
             UNTIL>
SP:=SP-1
```

UNTIL terminates a REPEAT-UNTIL control structure. If TOS has the value 'false', control is returned to the corresponding REPEAT. Otherwise, the execution continues following the UNTIL.

5.3.3 VOCABULARY MANIPULATION PRIMITIVES

The primitives described in this section add new words to the vocabulary or access parts of the vocabulary.

5.3.3.1 DEFINITION

DEFINITION	Define a new word.
Stack Contents:	<STRING> DEFINITION ---
Syntax:	<NAME> DEFINITION --- <previous words> --- END_DEF
Operation:	VP:=VP+1; VOCABULARY[VP]:=(DICT_0, <STRING>); STATE := COMPILE

DEFINITION causes a new vocabulary entry to be made with the name given on the stack. The precedence of the new word is set to zero, the lowest possible precedence. DEFINITION changes the precedence of the interpreter to COMPILE mode so that words with lesser precedence, that is zero, are compiled into the vocabulary, rather than executed immediately. The compilation continues until the interpreter precedence is again lowered, normally by END_DEF. Unpredictable results may occur if DEFINITIONs are nested. In short, the DEFINITION-END_DEF construct provides a user defined macro facility.

5.3.3.2 END_DEF

END_DEF	End the definition of new word or terminate the execution of a word.
Stack Contents:	--- END_DEF ---
Syntax:	(See DEFINITION above)
Operation:	if (STATE = COMPILE)

```
if ( STATE = COMPILE )
   then VP := VP + 1
        VOCABULARY[VP]:=(PROC,END_DEF)
        STATE := EXECUTE
   else <terminate execution of current
         word>
```

END_DEF terminates the current definition and makes an entry, pointing to itself, into the vocabulary when in compile mode. END_DEF has a precedence equal to the interpreter's compile mode, so that it is executed even when the interpreter is in COMPILE mode. When the defined word is executed and the END_DEF word is referenced, execution of the current word terminates and control is returned to the word that invoked the current word.

5.3.3.3 CONSTANT

CONSTANT	Define the named constant.
Stack Contents:	<DS> <STRING> CONSTANT ---
Syntax:	<DS> <NAME> CONSTANT

Operation: VP:=VP+1;VOCABULARY[VP]:=(DICT_0,<STRING>

VP:=VP+1;VOCABULARY[VP]:=(PROC,VAR)

VP:=VP+1;VOCABULARY[VP]:=<DS>

This command enters into the vocabulary a named constant. The constant may be of any data structure type. When the constant is referenced in subsequent words, its value is pushed onto the stack. The constant values may not be changed onced the constants have been defined.

5.3.3.4 VARIABLE

VARIABLE Define a named variable.

Stack Contents: <DS> <STRING> VARIABLE

Syntax: <DS> <NAME> VARIABLE

Operation: VP:=VP+1;VOCABULARY[VP]:=(DICT_0,<STRING>)

VP:=VP+1;VOCABULARY[VP]:=(PROC,VAR)

VP:=VP+1;VOCABULARY[VP]:=<DS>

The command VARIABLE defines a named variable. When that name is referenced, its address in the vocabulary is returned to the stack. The address is an <INT> data type and so can be manipulated with the arithmetic operators. The contents or the value of the variable can be obtained using a command FETCH and a value can be assigned to a variable using a command STORE. A primitive ALLOCATE (to be explained

later) may be used to allocate additional space in the vocabulary to the variable, that is, define a linear array.

5.3.3.5 FETCH

FETCH	Push contents of the vocabulary location onto the stack.
Stack Contents:	<INT> FETCH VOCABULARY[<INT>]
Syntax:	<INT> FETCH
Operation:	STACK[SP] := VOCABULARY[<INT>]

The command FETCH interprets the TOS as an index into the vocabulary. The contents of the corresponding vocabulary location replace the TOS.

5.3.3.6 STORE

STORE	Place NOS into the vocabulary indexed by TOS
Stack Contents:	<DS> <INT> STORE ---
Syntax:	<DS> <INT> STORE
Operation:	VOCABULARY [<INT>] := <DS> SP := SP - 2

The TOS is interpreted as an index into the vocabulary. The NOS is placed into the vocabulary at the location indexed.

5.3.3.7 ALLOCATE

ALLOCATE — Reserve space in the vocabulary.

Stack Contents: <INT> ALLOCATE ---

Syntax: <INT> ALLOCATE

Operation:
```
VP := VP + <INT>
SP := SP - 1
```

The number of elements indicated by the TOS is reserved in the vocabulary. ALLOCATE is normally used following a VARIABLE definition to reserve additional storage for the variable. The additional storage can be referenced by performing arithmetic operations on the vocabulary address obtained when a VARIABLE is referenced.

5.3.3.8 FORGET

FORGET — Remove the most recent part of vocabulary.

Stack Contents: <STRING> FORGET ---

Syntax: <NAME> FORGET

Operation:
```
if ( <NAME> in VOCABULARY )
   then (let VOCABULARY[<INT>] := <NAME>)
      VP := VP - <INT>
SP := SP - 1
```

The given <NAME> is looked up in the vocabulary and if found, all the words defined after the word, including itself, are removed from the vocabulary. The <NAME> must be the name of a word definition.

5.3.3.9 #VOCB

#VOCB	Return the aount of space used in the vocabulary.
Stack Contents:	--- #VOCB <INT>
Syntax:	#VOCB
Operation:	SP := SP + 1 STACK [SP] := VP

The amount of space used in the vocabulary is returned on the TOS. This is useful when scanning the vocabulary is necessary.

5.3.4 DATABASE MANIPULATION PRIMITIVES

The primitives described in this section are used for a specific purpose. The user can interact with our database system using these primitive operations. These operations help the user to access the information stored in the database, that is, to create, build, scan, and query the spatial data structures and the relations in the system.

5.3.4.1 ALLOC_RDS

ALLOC_RDS	Allocate and initialize the spatial data structure header.
Stack Contents:	<INT1> <INT2> <STRING> ALLOC_RDS <RDS>
Syntax:	<TYPE> <DIMEN> <NAME> ALLOC_RDS
Operation:	SP := SP-2 STACK[SP] := <RDS>

ALLOC_RDS allocates a new spatial data structure header, called an RDS, with the specified name, and fills it with the information provided. The TYPE for a spatial data structure is one and the DIMENSION is zero. The LENGTH and USE_CNT are set to zero and the STRUCT is set to nil. The STRUCT type is set to SE_RC (a pointer to a RELATION LIST). The name and the pointer to this SDS header are entered into the RDS DICTIONARY. Finally, the pointer to the SDS header is returned on the stack.

5.3.4.2 ALLOC_REL

ALLOC_REL	Allocate and initialize a relation header
Stack Contents:	<INT1> <INT2> <STRING> ALLOC_REL <RDS>
Syntax:	<TYPE> <DIMENSION> <NAME> ALLOC_REL
Operation:	SP := SP-2 STACK[SP] := <REL>

The command ALLOC_REL allocates a relation header with the specified name, type, and dimension and fills it with the information provided. The type for a TREE relation is two and that for a LIST relation is three. The LENGTH and USE_CNT are set to zero, and the STRUCT is set to nil. The STRUCT type is set to either SE_TREE for a TREE relation, or to SE_LIST for a LIST_RELATION. That is, a pointer to a TREE_CELL or a LIST_CELL respectively. The relation is entered into the REL_DICTIONARY and finally, the pointer to the relation header is pushed onto the stack.

5.3.4.3 RDS_INDEX

```
RDS_INDEX             Get the index of an <RDS> specified by
                      the given name.
Stack Contents:       <STRING> RDS_INDEX (<INT> or <STRING>)
                      <T/F>
Syntax:               <STRING> RDS_INDEX
Operation:            if <STRING> in RDS_DICT
                         then STACK[SP] := <index of <RDS>>
                              SP := SP+1
                              STACK[SP] := 'true'
                         else SP := SP+1
                              STACK[SP] := 'false'
```

This command searches the RDS_DICTIONARY for the specified name. If a spatial data structure is found with the specified name, its index is added to the stack, and the status of 'true' is returned to the top of the stack. If the spatial data structure is not found, then the character string is retained on the stack and a status of 'false' is returned to the top of the stack.

5.3.4.4 REL_INDEX

REL_INDEX — Get the index of a <REL> specified by the given name.

Stack Contents: <STRING> REL_INDEX (<INT> or <STRING>) <T/F>

Syntax: <STRING> REL_INDEX

Operation:

```
if <STRING> in REL_DICT
   then STACK[SP] := <index of <REL>>
        SP := SP+1
        STACK[SP] := 'true'
   else SP := SP+1
        STACK[SP] := 'false'
```

This command searches the REL_DICTIONARY for the specified name. If a spatial data structure is found with the specified name, its index is added to the stack, and the

status of 'true' is returned to the top of the stack. If the spatial data structure is not found, then the character string is retained on the stack and a status value of 'false' is returned to the top of the stack.

5.3.4.5 (CATALOG)

(CATALOG)	Access the entries in the RDS_DICTIONARY.
Stack Contents:	<INT> (CATALOG) <RDS>
Syntax:	<RDS_DICT index> (CATALOG)
Operation:	STACK[SP] := RDS_DICT[STACK[SP]]

The content of the RDS_DICTIONARY entry indexed by the TOS is returned to the TOS. This is a pointer to a spatial data structure header.

5.3.4.6 (REL_CATALOG)

(REL_CATALOG)	Access the entries in the REL_DICTIONARY.
Stack Contents:	<INT> (REL_CATALOG) <REL>
Syntax:	<INT> (REL_CATALOG)
Operation:	STACK[SP] := REL_DICT[STACK[SP]]

The content of the REL_DICTIONARY entry indexed by the TOS is returned to the TOS. This is a pointer to a relation header.

5.3.4.7 #CATALOG

#CATALOG — Return the number of entries in the RDS_DICTIONARY.

Stack Contents: --- #CATALOG <INT>

Syntax: #CATALOG

Operation:

```
SP := SP+1
STACK[SP] := <number of entries in the
                     RDS_DICTIONARY>
```

The number of entries in the RDS_DICTIONARY is pushed onto the stack.

5.3.4.8 #REL_CATALOG

#REL_CATALOG — Return the number of entries in the REL_DICTIONARY.

Stack OPeration: --- #REL_CATALOG <INT>

Syntax: #REL_CATALOG

Operation:

```
SP := SP+1
STACK[SP] := <number of entries in the
                     REL_DICTIONARY>
```

The number of entries in the REL_DICTIONARY is pushed onto the stack.

5.3.4.9 FIND

FIND — Obtain an <RDS>, <REL> or a <VOCB> specified by the given name.

Stack Contents: <STRING> FIND <RDS> or <REL> or <VOCB> or <STRING> <T/F>

Syntax: <STRING> FIND

Operation:

```
case ( STACK[SP] ) in
   RDS_DICT   : STACK[SP] := <RDS>
                SP := SP + 1
                STACK[SP] := 'true'
   REL_DICT   : STACK[SP] := <REL>
                SP := SP+1
                STACK[SP] := 'true'
   VOCABULARY : SP := SP+1
                STACK[SP] := 'true'
   otherwise    SP := SP+1
                STACK[SP] := 'false'
```

Given a name, this command first searches the RDS_DICTIONARY for a spatial data structure with the specified name. If found, a pointer to its header is returned to tne NOS and a boolean value 'true' is returned in the TOS. Otherwise, the REL_DICTIONARY is searched next. If a relation with the specified name is found in the REL_DICTIONARY,

a pointer to its header is returned in the NOS and a boolean value of 'true' is pushed onto the stack. If the specified name is not a valid relation name, the VOCABULARY is searched. If the given name is a valid command in the VOCABULARY, a boolean value of 'true' is pushed onto the stack. Otherwise, a boolean value of 'false' is pushed onto the stack and the specified name is retained on the stack at the NOS.

5.3.4.10 LIST_RDS and LIST_REL

LIST_RDS and LIST_REL	List the header of a spatial data structure and a relation respectively.
Stack Contents:	<RDS> LIST_RDS --- or <REL> LIST_REL ---
Syntax:	<RDS> LIST_RDS or <REL> LIST_REL
Operation:	<List the header pointed to by TOS> SP := SP-1

The contents of the spatial data structure header and a relation header are output to the current output text file, using the commands LIST_RDS and LIST_REL respectively. The output gives the label and value of each field of the header. The type of the header, spatial data structure or relation, is also output.

5.3.4.11 XLIST_RDS and XLIST_REL

XLIST_RDS or XLIST_REL	List the header and the contents of a spatial data structure and a relation respectively.
Stack Contents:	<RDS> XLIST_RDS --- or <REL> XLIST_REL ---
Syntax:	<RDS> XLIST_RDS or <REL> XLIST_REL
Operation:	<List the SDS or the relation pointed to by the TOS> SP := SP-1

The commands XLIST_RDS and XLIST_REL output to the current text file the contents of a specified spatial data structure and a relation, respectively, along with their headers. If the structure is a spatial data structure, the name of each relation associated with it is printed. If the structure is a relation, the components of each N-tuple in the relation (only the values and not the types) are printed.

5.3.4.12 REL_ATTACH

REL_ATTACH	Attach a relation to a spatial data structure.
Stack Contents:	<REL> <RDS> REL_ATTACH ---
Syntax:	<REL> <RDS> REL_ATTACH

Operation: <Attach the relation pointed to by NOS to the SDS pointed by TOS.>

The relation pointed to by the NOS is attached to the spatial data structure pointed to by the TOS. The relation is attached as the first relation in the RELATION LIST for the spatial data structure.

5.3.4.13 NT_ATTACH

NT_ATTACH Attach an N-tuple to a relation.

Stack Contents: <DSn> --- <DS1> <REL> NT_ATTACH ---

Syntax: <DSn> --- <DS1> <REL> NT_ATTACH

The N-tuple from the STACK[SP-DIMEN(<REL>)-1] through the NOS is attached to the relation pointed by the TOS. The size of the N-tuple is determined by the DIMENSION field of the relation header. The N-tuple is inserted in the relational tree structure or the relational list structure, as described in Chapter IV. Both N-tuple and the relation pointer are dropped from the stack.

5.3.4.14 INRELA?

INRELA? Check if an N-tuple is in a specified relation.

Stack Contents: <DSn> --- <DS1> <REL> INRELA? <T/F>

Syntax: <DSn> --- <DS1> <REL> INRELA?

Operation: if (<the N-tuple is in <REL>)

then STACK[SP-DIMEN(<REL>)-1]:='true'

else STACK[SP-DIMEN(<REL>)-1]:='false'

SP := SP-DIMEN(<REL>)-1

The relation pointed to by the TOS is scanned for the N-tuple below it on the stack. The size of the N-tuple is determined from the DIMENSION field of the relation header. If the N-tuple is found, a boolean value of 'true' is returned in the TOS. Otherwise, a value of 'false' is returned. The N-tuple and the relation pointer are removed from the stack in either case.

5.3.4.15 [NAME]

[NAME] Return the name of a spatial data structure or a relation.

Stack Contents: <RDS> or <REL> [NAME] <STRING>

Syntax: <RDS> or <REL> [NAME]

Operation: STACK[SP] := NAME(<RDS> or <REL>)

This command returns in the TOS, the name of the spatial data structure or the relation pointed to by the TOS.

5.3.4.16 [TYPE]

[TYPE] Return the type of the given structure,
1 for a spatial data structure,
2 for a relational tree structure.
3 for a relational list structure.

Stack Contents: <RDS> or <REL> [TYPE] <INT>

Syntax: <RDS> or <REL> [TYPE]

Operation: STACK[SP] := TYPE(<RDS> or <REL>)

The type of the specified spatial data structure or the relation, pointed to by the TOS, is returned in the TOS.

5.3.4.17 [DIMEN]

[DIMEN] Return the dimension of a given spatial data structure or a relation.

Stack Contents: <RDS> or <REL> [DIMEN] <INT>

Syntax: <RDS> or <REL> [DIMEN]

Operation: STACK[SP] := DIMEN(<RDS> or <REL>)

The dimension of the given SDS or the relation pointed to by the TOS is returned in the TOS.

5.3.4.18 [LENGTH]

[LENGTH]	Return the length of a given spatial data structure or a relation.
Stack OPerations:	<RDS> or <REL> [LENGTH] <INT>
Syntax:	<RDS> or <REL> [LENGTH]
Operation:	STACK[SP] := LENGTH(<RDS> or <REL>)

This command returns in the TOS the length of the spatial data structure or the relation pointed to by the TOS.

5.3.4.19 [USE_CNT]

[USE_CNT]	Return the use count of a given spatial data structure or a relation.
Stack Contents:	<RDS> or <REL> [USE_CNT] <INT>
Syntax:	<RDS> or <REL> [USE_CNT]
Operation:	STACK[SP] := USE_CNT(<RDS> or <REL>)

The use count or the number of references to the given spatial data structure or the relation pointed to by the TOS is returned in the TOS.

5.3.4.20 [STRUCT]

[STRUCT]	Return the structure pointed to by the given pointer, specified by the TOS.
Stack Contents:	<PTR> [STRUCT] <PTR>

Syntax: <PTR> [STRUCT]

Operation: STACK[SP] := < data structure of <PTR> >

The data structure pointed to by the specified pointer is returned in the TOS. This command is applicable only to the data types that support lower level structures. The following table describes the effect of this operation on the allowable data structures.

ARGUMENT	RESULT	OPERATION
<RDS>	<RC>	<RDS>@.STRUCT
<REL>	<TREE> or <LIST>	<REL>@.STRUCT
<TREE>	<TREE>	<TREE>@.FIRST_CHILD
<LIST>	<LIST>	<LIST>@.LC_DATA (FOR A VERTICLE LIST)
<RC>	<REL>	<RC>@.RELATION

If the argument is of type <TREE> the CHILD pointer of the node is returned. In the case of <LIST>, the next LIST_CELL in the vertical list is returned (refer to Chapter IV). If the argument is not one of the above types, then an error message is printed on the user's terminal, and the argument is retained on the stack. The current version of the interpreter code does not check for a nil pointer; and it is the user's responsibility to ensure that a nil pointer is not given as an argument to this operation.

5.3.4.21 [LINK]

[LINK]	Advance one node in a linked list.
Stack Contents:	<PTR> [LINK] <PTR>
Syntax:	<PTR> [LINK]
Operation:	STACK[SP] := <next data structure in the linked list.>

The next data structure on the current level is returned in the TOS. This command is applicable only to those data structures which support linking nodes of the same type together, that is, which can grow on the same level. The following table describes the effect of this command on the allowable data types.

ARGUMENT	RESULT	OPERATION
<RC>	<RC>	<RC>@.NEXT_RELATION
<TREE>	<TREE>	<TREE>@.NEXT_SIBLING
<LIST>	<LIST>	<LIST>@.NEXT_ELEMENT
<STRING>	<STRING>	<STRING>@.CS_LINK

The error checking for a nil pointer is not yet provided, and it is again the user's responsibility to ensure that a nil pointer is not used as an argument for this command. If the argument is not one of the above types, then an error message is output to the user's terminal and the argeument is retained on the stack.

5.3.4.22 [DATA]

[DATA] Return the data from the node pointed to by the TOS.

Stack Contents: <PTR> [DATA] <DS>

Syntax: <PTR> [DATA]

Operation: STACK[SP] := < data at <PTR> >

The data at the node pointed to by the TOS is returned on the stack at TOS. The following data types are supported as arguments for this command: <TREE>, <LIST>, and <POINT>. All other data types cause an error message to be printed on the user's terminal and the argument is retained on the stack. Error checking for a nil pointer as an argument is not yet provided.

5.3.4.23 RDS_DICT

RDS_DICT Print the specified portion of the RDS_DICTIONARY.

Stack Contents: <INT1> <INT2> RDS_DICT ---

Syntax: <INT1> <INT2> RDS_DICT

Operation: for I := <INT1> to <INT2> do

writeln (name of RDS_DICT[I]);

This command prints a portion of the RDS_DICTIONARY specified by the NOS and the TOS on the user's terminal. Only the names of the spatial data structures from the dictionary are printed.

5.3.4.24 REL_DICT

REL_DICT	Print the specified portion of the REL_DICTIONARY.
Stack Contents:	<INT1> <INT2> REL_DICT ---
Syntax:	<INT1> <INT2> REL_DICT
Operation:	for I := <INT1> to <INT2> do writeln (name of REL_DICT[I])

This command prints the portion of the REL_DICTIONARY indicated by the NOS and the TOS on the user's terminal. Only the names of the relations from the dictionary are printed.

5.3.5 <u>MISCELLANEOUS COMMANDS</u>

In this section we describe the commands that are included in the system for the convenience of the user, and the commands for loading and unloading of the database.

5.3.5.1 DB_LOAD

DB_LOAD — Load the database into memory from the disk.

Stack Contents: --- DB_LOAD ---

Syntax: DB_LOAD

This command loads the database from the disk into memory and gives access to the information in it to the user. The operation performed by this command is to read into memory the names of all the spatial data structures and relations from the RDS_DICTIONARY and the REL_DICTIONARY, respectively, residing on the disk. It also initializes the other fields in those dictionaries, which are in memory, to either nil or zero, depending on their types. Finally a message is printed on the user's terminal, indicating that the geographic database is loaded. This command depends on the VAX/VMS implementation of the Pascal system.

5.3.5.2 DB_UNLOAD

DB_UNLOAD — Unload the database from memory.

Stack Contents: --- DB_UNLOAD ---

Syntax: DB_UNLOAD

This command unloads the databse from the memory. The user can no longer access the database once this primitive

is invoked. This command writes both the dictionaries (only the names of the SDS's and relations) from memory on the disk. It then transfers the existing spatial data structures and relations in memory to the disk. Only those SDS's and relations are required to be transfered to the disk whose structures have been changed, that is if any relation is added to or deleted from any spatial data structure, or any N-tuple is inserted, updated or deleted from any relation. This command also depends on the VAX/VMS implementation of the Pascal system.

5.3.5.3 INPUT>

INPUT>	Begin input from a specified text file.
Stack Contents:	<STRING> INPUT> ---
Syntax:	<file description> INPUT>

This command suspends the input from the current source and opens the named file for continuing the input stream. The input continues from the new file until the word ***EOF is encountered at which time the input file is closed. The input then continues from the interactive terminal. Initially, before accessing the specified file, a test is performed to check whether or not a file with a specified name exists. If the specified file does not exists, an error mes-

sage is printed. The system currently does not support nesting of the input files. This command depends on the VAX/VMS implementation of the Pascal system.

5.3.5.4 ***EOF

***EOF	Denote the end of an input file.
Stack Contents:	--- ***EOF ---
Syntax:	***EOF

When this command is interpreted, the current input file is closed and the input is continued from the interactive terminal.

5.3.5.5 >OUTPUT

>OUTPUT	Open a specified text file for an output.
Stack Contents:	<STRING> >OUTPUT ---
Syntax:	<file name> >OUTPUT

This commands searches the directory for the given file name and opens the file, if it exists, for an output. If the specified file does not exist in the directory, a new file is created with the given name and is opened for an output. If the given name is 'TERMINAL', the current text file, if not an interactive terminal, is closed; and the terminal becomes the output device.

5.3.5.6 DUMP

DUMP Print a portion of the VOCABULARY.

Stack Contents: <INT1> <INT2> DUMP ---

Syntax: <start index> <end index> DUMP

Operation: for I := <INT1> to <INT2> do

writeln (VOCBULARY[I])

The contents of the VOCABULARY from the start location to the end location are output to the current output text file. This is a convenient way of examining the contents of the VOCABULARY when de-bugging the programs.

5.3.5.7 "

" Compile a new character string.

Stack Contents: " <STRING> " <STRING>

Syntax: " <STRING> "

The command " causes the input character string to be scanned until another " is encountered. The intervening characters are returned as a string. Note that " is a word and must be followed by a delimiting blank. The delimiting blank is not a part of the character string. Any further blanks will be part of the character string upto, but not including, the ". For example, some valid character strings

are: " PRASHANT", " REGION N3". If the interpreter is in EXECUTE mode, the character string is pushed onto the stack. If the interpreter is in COMPILE mode, the character string is entered in the vocabulary.

5.3.5.8 GETWORD

GETWORD — Compile the next character string word.

Stack Contents: GETWORD <STRING> <STRING>

Syntax: GETWORD <STRING>

This command scans the input text file for the next word and returns it as a character string on the TOS.

5.3.5.9 DONE

DONE — Terminate the intepreter session.

Stack Contents: --- DONE

Syntax: DONE

The current session is terminated and the control is returned to the timesharing system that invoked the interpreter.

5.4 EXAMPLES

In this section we use examples to show how the interpreter primitives described earlier can be used to perform certain database operations.

Each example shown below uses one or more query language commands. For each command, first the arguments (if there are any) are pushed onto the stack, and then the command is invoked. For some commands the results of the previous commands are used as arguments. Each line of input text is shown under the column INPUT. A brief explanation of each line of input text is shown under the column EXPLANATION. Each command in the query language is marked by an asterisk. The arguments are shown as they are entered by the user. The input character strings are enclosed between a pair of double quotes. Note that, double quotes (") is a valid command in the language. Finally, at the end of the example, the contents of the stack resulting from the execution of the command/s are displayed. Both the type and the value of each element on the stack are shown in the format 'TYPE-- VALUE'. The stack grows upward; that is, the first element represents the top of stack. The commands for displaying the entire stack or displaying the top of the stack are not explicitely shown, but this can be achieved using the commands .STACK and '.', respectively.

5.4.1 Load the Database.

The following example loads the database into memory from the disk and gets some general information about the database, such as number of relations and spatial data structures in the database.

INPUT	EXPLANATION
* DB_LOAD	Load the database into memory from disk.
* #CATALOG	Number of entries in the RDS_DICTIONARY.
* #REL_CATALOG	Number of entries in the REL_DICTIONARY.

The stack contents are:

INT-- 1004	Number of relations in the database.
INT-- 508	Number of spatial data structures in the database.

5.4.2 Get a Relation from the Dictionary

This example shows how a pointer to a relation or a spatial data structure with a specified name can be put on the stack, if it exists in the database. We will search for an existing relation A/V_REGION_X in the system.

INPUT	EXPLANATION
" A/V_REGION_X"	Name of the relation to be searched.
* FIND	Search the REL_DICTIONARY for the given

name.

The stack contents are:

INT-- 0 or 1	The status of the search; 1 if found and 0 if not found.
REL-- A/V_REGION_X or	A pointer to the relation header if the status is found.
CHAR-- A/V_REGION_X	The character string is retained if the status is not found.

Given a name of a spatial data structure, this example searches the RDS_DICTIONARY for that spatial data structure. If the SDS is found in the dictionary, its index in the dictionary is put on the top of the stack. This index is then used to fetch a pointer to its header.

INPUT	EXPLANATION
" REGION_X"	Name of the spatial data structure.
* RDS_INDEX	Command to get the index of the SDS.
* DROP	Drop the status of the search.

The index of the spatial data structure REGION_X is now on the top of the stack. We assumed that the spatial data structure with the given name is present in the system.

* (CATALOG)	The command for getting the SDS header.

The top of the stack is:

RDS-- REGION_X A pointer to the SDS header for REGION_X.

5.4.3 Create a New Relation

The following example creates a new binary relation having a tree structure and inserts N-tuples into it.

INPUT	EXPLANATION
2	Dimension of the relation.
2	Type of the relation.
" A/V_REGION_Y"	Name of the relation to be created.
* ALLOC_REL	Command to create the relation.

The top of stack now contains a pointer to the header of this new relation.

INPUT	EXPLANATION
" AREA"	First component of the N-tuple.
12345	Second component of the N-tuple.
* ROT	Rotate the top three elements on the stack to get the pointer to the relation header on the top of stack.
* NT_ATTACH	Command to attach the N-tuple to the relation.

At the end of these operations the stack is empty because the pointer to the relation header is dropped after the N-tuple is attached to it.

5.4.4 Attach a Relation to a Spatial Data Structure

This example attaches a relation A/V_REGION_X to a spatial data structure REGION_X. We assume that the relation AV_REGION_X and the spatial data structure REGION_X are present in the system.

INPUT	EXPLANATION
" A/V_REGION_X"	Name of the relation to be attached.
* FIND	Get the pointer to the relation header.
* DROP	Drop the top of stack which is the status of the search.
" REGION_X"	Name of the spatial data structure.
* FIND	Get the pointer to the SDS header.
* DROP	Drop the top of the stack.
* REL_ATTACH	Attach the relation pointed to by the next of stack to the spatial data structure pointed to by the top of stack.

The stack is empty after these operations.

5.4.5 Define a New Command.

The following example shows how a new command can be created in the system. We will use this new command in our later examples. The operation performed by this new command is to print a relation or a spatial data structure with the specified name, if it is present in the system. The name of the relation or SDS is stored in a predefined variable 'XYZ'. The value stored in the variable 'XYZ' is assumed to be of type CHAR_STRING.

INPUT	EXPLANATION
" PRINT_REL"	Name of the new command.
* DEFINITION	The command to start the definition of new command. This command increases the state of the interpreter and puts it into COMPILE mode.
XYZ	Variable specifying the name of the relation.
* FETCH	Get the name on the top of the stack.
* FIND	The command to search the relation in the dictionary.

The top of stack now contains the status of the search. If the TOS is true, then a pointer to the header of a relation or a spatial data structure is next on the stack.

* IF   Begining of the IF-THEN construct.

                        If the TOS is 'true', then perform the following operations.

Now check if the TOS is a pointer to a relation header or a spatial data structure header. Then invoke a proper command to print that relation or SDS.

* RDS?   Check if the TOS is a pointer to an SDS header.

The TOS now contains a value 'true' or 'false'.

* IF   Begining of the IF-ELSE-THEN construct. If the TOS is 'truè', then the code between IF and ELSE is executed. Otherwise, code between ELSE and THEN is executed.

                        --<Begining of the 'true' branch>--

The TOS is a pointer to a spatial data structure header.

* XLIST_RDS   Print the spatial data structure pointed

to by the TOS.

* ELSE --<End of the 'true' branch>--

--<Begining of the 'false' branch>--

The TOS is a pointer to a relation header.

* XLIST_REL Print the relation pointed to by the TOS.

* THEN --<End of the 'false' branch>--
End of the IF-ELSE-THEN construct.

* THEN End of the IF-THEN construct.

* END_DEF The command to end the definition of the new command. This command is executed even in COMPILE mode and reduces the state of the interpreter.

At the end of these operations the stack is empty. The command PRINT_REL is entered in the VOCB_DICTIONARY.

5.4.6 Traverse a Relation

This example shows how a relation can be traversed to get the individual components of the N-tuples in the relation on the stack. In the example, a relation A/V_REGION_X is used. The relation A/V_REGION_X is a binary relation and the N-tuples are stored in a tree structure in this relation. The physical structure of the relation A/V_REGION_X is redrawn in Figure 5.4 for reference. We use a REPEAT-UNTIL construct to traverse the entire relation. Each iteration accesses one N-tuple and displays the indiviual components in it. In the example, the stack contents only during the first iteration of the REPEAT-UNTIL loop are illustrated.

INPUT	EXPLANATION

First let us define a variable X and initialize it to zero.

0	Initial value of the variable.
" X"	Name of the variable.
* VARIABLE	The command to create a variable.
" A/V_REGION_X"	Name of the relation to be traversed.
* FIND	Search the relation name in the dictionary and get a pointer to its header.
* DROP	Drop the status of the search to get the

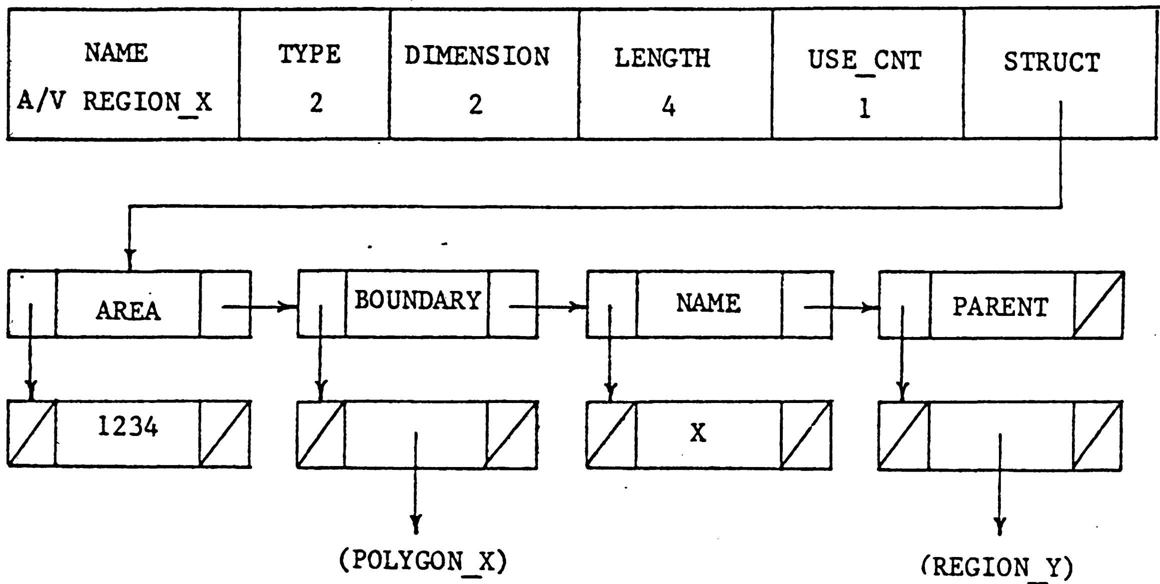


Figure 5.4 : Physical Structure of relation A/V_REGION_X.

pointer to the relation header on TOS.

The stack contents are:

```
| REL-- A/V_REGION_X |
|____________________|
```

```
* [STRUCT]          Get the first component of the first
                    N-tuple.
```

The top of stack now contains a pointer to a TREE_CELL which contains that component. The stack contents are:

```
| TREE--  |
|_________|
```

```
* REPEAT            Starting of the repeat-until loop.
```

Save the pointer to the first component of the N-tuple in variable X.

```
  X                 Name of the variable.
* STORE             Command to store the pointer in X.
```

The stack is now empty, so get the pointer to the first component of the N-tuple back on the top of stack.

```
  X                   The variable.
* FETCH               Fetch the value of the variable.
```

The top of stack now contains a pointer to the TREE_CELL.

```
* [DATA]              Get the value of the component that
                      resides in the data field of the
                      TREE_CELL which is on the TOS.
```

The top of stack now contains the value of the component which is

```
| CHAR-- AREA |
|_____________|
```

Now get the next component of the N-tuple.

```
  X                   Get the pointer to the first component.
* FETCH               The command to fetch the value of X.
* [STRUCT]            Get the pointer to the FIRST_CHILD of
                      the TREE_CELL.
```

The TOS now contains a pointer to the TREE_CELL which contains the second component of the N-tuple. The contents of the stack are:

```
| TREE--        |
| CHAR-- AREA   |
|_______________|
```

```
* [DATA]           Get the value of the second component.
```

The TOS now contains the second component of the N-tuple which is INT-- 1234

```
| INT-- 1234    |
| CHAR-- AREA   |
|_______________|
```

Now drop the components of the current N-tuple from the stack and get the next N-tuple.

```
  2                Number of elements to be dropped.
* #DROP            The command to drop the elements from the
                   stack.
```

The stack is now empty. Now get the next N-tuple.

```
  X                Get the first component of the current
                   N-tuple.
* FETCH            Fetch the value of X.
* LINK             Get the NEXT_SIBLING of X.
```

The TOS now points to the TREE_CELL containing the first component of the next N-tuple if it exists. The stack contents are:

```
| TREE--        |
|_______________|
```

* -NIL? Check the TOS for the nil pointer.

* UNTIL End of the REPEAT-UNTIL loop.
If the TOS is nil then terminate the loop. Else get the next N-tuple.

This example thus traverses the entire relation. The stack contents can be observed by using the commands '.' and '.STACK' at appropriate places.

The repeat-until loop described above can be used to construct a new command using the DEFINITION-END_DEF construct. Any binary relation having a TREE structure can be traversed by invoking that command. Similarly, any relation having a LIST structure or any spatial data structure can also be traversed.

Chapter VI

THE MEMORY MANAGEMENT

Our geographic database resides on a secondary device and a portion of it is retrieved and brought into memory as necessary. Each spatial data structure and relation is stored in a separate file on the disk. We described the physical structure of the database on the secondary device and its implementation in Chapter V. In this chapter we describe the paging algorithm or the memory mangement scheme that controls the transfer of information in the database back and forth between memory and disk.

Although, we use the term 'paging', conventional paging is not being performed here. We do not retrieve the information from the disk in the form of fixed-size pages. Instead we transfer entire relations or spatial data structures into memory from the disk and back.

6.1 THE ARRAYS REL IN CORE AND RDS IN CORE

The system keeps two arrays REL_IN_CORE and RDS_IN_CORE for relations and spatial data structures, respectively. These arrays are circular, and their elements are the indices of the relations and spatial data structures whose structures are currently in memory, in the REL_DICTIONARY and RDS_DICTIONARY, respectively. The variables

LRU_REL_IN_CORE and MRU_REL_IN_CORE represent the indices in the array REL_IN_CORE of the least recently read and the most recently read relations in memory, respectively. Similarly, the variables LRU_RDS_IN_CORE and MRU_RDS_IN_CORE represent the indices of the least recently read and the most recently read spatial data structures in memory, respectively, in the array RDS_IN_CORE. Every time any relation or spatial data structure is read or created in memory, its index in the corresponding dictionary is entered in the array REL_IN_CORE or RDS_IN_CORE, respectively. The variables NRDS_IN_CORE and NREL_IN_CORE represent the total number of spatial data structures and relations, respectively in memory.

6.2 COMPUTATION OF SPACE

A global variable SPACE represents the amount of space occupied in memory. Only the space occupied by the relations and spatial data structures is taken into consideration, and that of the global arrays and dictionaries is not counted. Whenever any relation or spatial data structure is read into memory, the space taken by the corresponding structure is added to this variable. Similarly, whenever any structure is transferred back to the disk and deleted from memory, the space freed by that structure is subtracted from the variable SPACE.

The space occupied by the structures is computed in terms of full words. The guideline for computing the space is as follows. The VAX/VMS implementation of Pascal takes one full word for an integer, a real, or a pointer data type, and one byte to store a character. A character string variable in our system is fifteen characters long and hence each character string variable uses fifteen bytes of memory. A full word in a VAX system is formed of four bytes. Thus a character string occupies four words in memory. For example, the space occupied by a relation or a spatial data structure header (an RDS) is as shown below.

FIELD	DATA TYPE	SPACE
NAME	Character String	4 words.
TYPE	Integer	1 word.
DIMENSION	Integer	1 word.
LENGTH	Integer	1 word.
USE_CNT	Integer	1 word.
STRUCT	Pointer	1 word.
	TOTAL	9 words.

Now, we explain the operations performed when a spatial data structure or a relation is read into memory, transferred back to the disk, and deleted from memory.

6.3 READ RELATIONS AND SPATIAL DATA STRUCTURES INTO MEMORY

Whenever the user references any relation or a spatial data structure which is not in memory, only its header is initially created in memory. The entire structure of the relation or spatial data structure is read only when the user wants to access the information in it.

6.3.1 READING A RELATION INTO MEMORY

Whenever the user references any relation, its header is created in memory as follows. The file containing that corresponding relation is opened for reading[12]. The first five records which contain the header information such as its NAME, TYPE, DIMENSION, LENGTH, and USE_CNT are read from the file. A new relation header (an RDS) is created using the standard Pascal function 'NEW', with the corresponding name, type, and dimension. The space occupied by the header is added to the global variable SPACE. The REL_ADDRESS field corresponding to that particular relation in the REL_DICTIONARY is set to point to this new header. It should be noted that the relation is empty at this time. The file is then closed for reading.

---

12) This is a VAX/VMS implementation dependent feature and not a standard Pascal function.

Now if the user wants to access any information in the relation, the entire relation is read into memory. Once again the file containing that relation is opened for reading. The REL_DICTIONARY is searched to get its index in the dictionary and to obtain a pointer to its header. The pointer to its header is obtained from the corresponding REL_ADDRESS field in the dictionary[13]. The index of the relation is entered in the next available element of the array REL_IN_CORE. The variable MRU_REL_IN_CORE is updated to point to this new array element. The REL_TAG field of that relation in the REL_DICTIONARY is set to one to indicate that the entire relation has been read into memory. The variable NREL_IN_CORE is incremented by one. The N-tuples are then read one by one from the file and are inserted into the relation. If a component of any N-tuple is a spatial data structure, and if that particular spatial data structure is not present in memory, then a header for that spatial data structure is created (to be explained in the next section) and a pointer to it is returned as the component. The file is closed for reading after all the N-tuples are read and inserted in the relation.

----

13) It should be noted that before reading an entire relation into memory, a header for that relation is already created (refer to the preceding paragraph).

Whenever the user creates a new relation, its header is created with the specified name, type, and dimension. The relation is automatically cataloged in the REL_DICTIONARY. The index of the new relation is entered in the array REL_IN_CORE as before. Since cataloging the new relation in the REL_DICTIONARY changes the indices of some relations in the dictionary, the other entries in the array REL_IN_CORE are adjusted appropriately. The variables MRU_REL_IN_CORE and NREL_IN_CORE are modified accordingly. The N-tuples are then inserted in the new relation as they are specified by the user.

6.3.2 READING A SPATIAL DATA STRUCTURE INTO MEMORY

As in the case of relations, whenever the user references any spatial data structure which is not in memory, its header is created in memory. This is done in a way similar to the way in which relations are handled. A file containing the spatial data structure is opened for reading and a header (an RDS) is created with the corresponding name. The RDS_ADDRESS field in the RDS_DICTIONARY corresponding to that particular spatial data structure is set to point to the header. The file is then closed for reading.

If the user wants to access any information in the SDS, the entire SDS is read into memory. The file containing the

spatial data structure is opened for reading. The RDS_DICTIONARY is searched to get the index of that SDS in the dictionary. A pointer to its header is also obtained from the dictionary. The index of the spatial data structure is entered in the array RDS_IN_CORE. The variable MRU_RDS_IN_CORE is updated to point to this new array element. The RDS_TAG field in the RDS_DICTIONARY for that spatial data structure is set to one to indicate that the entire SDS has been read into memory. The name of each relation associated with that spatial data structure is then read from the file, and the relations are attached to the spatial data structure. If the header of any of those relations is not already in memory, then one is created for that relation as described in the preceding section. The file is closed for reading when all the relations associated with the spatial data structure are read and attached to the spatial data structure.

If the user creates a new spatial data structure, its header is first created with the specified name, type, and dimension. The spatial data structure is automatically entered in the RDS_DICTIONARY. The index of the spatial data structure is entered in the array RDS_IN_CORE. Since entering the new spatial data structure in the dictionary changes the indices of some spatial data structures in the diction-

ary, the other entries in the array RDS_IN_CORE are adjusted appropriately. The variables MRU_RDS_IN_CORE and NRDS_IN_CORE are modified accordingly. The relations associated with spatial data structure are then attached to it as they are referenced by the user.

6.4 TRANSFER OF RELATIONS AND SPATIAL DATA STRUCTURES TO THE DISK

If a sufficient space is not available in memory for the new relations or spatial data structures, some relations and/or spatial data structures are transferred back to the disk and then deleted from memory (to be explained in the following section). A relation or a spatial data structure is written to the disk only if its structure has been changed. For a relation, the REL_CHANGE field corresponding to that relation in the REL_DICTIONARY indicates whether or not the relation has been changed (one if changed and zero if not). Similarly, for a spatial data structure, the RDS_CHANGE field in the RDS_DICTIONARY corresponding to that SDS indicates whether or not the structure of the SDS has been changed.

6.4.1 TRANSFERING A RELATION TO THE DISK

Whenever any relation is to be written back to the disk from memory, its entry in the array REL_IN_CORE is deleted. The REL_TAG field corresponding to that relation in the REL_DICTIONARY is set to zero indicating that the relation is no longer in memory. The relation is then transferred to the disk as described in Chapter V. A file is created on the disk with the name of the relation, and the header and all the N-tuples are written onto the disk. Since the old file containing the same relation already exists on the disk the new file has the same name but a higher version number[14]. The file with the older version can be deleted using a VAX/VMS command.

6.4.2 TRANSFERRING A SPATIAL DATA STRUCTURE TO THE DISK

The operations performed when any spatial data structure is transferred back to the disk are as follows. The entry for that spatial data structure in the array RDS_IN_CORE is deleted. The RDS_TAG field in the RDS_DICTIONARY for that particular spatial data structure is set to zero indicating that it has been transferred to the disk. A new file is then created on the disk with its name and the entire structure is written onto that file as described in Chapter V. As in

----

14) This is a VAX/VMS file maintenance feature.

case of relations, this new file has a higher version number since the old file containing the same SDS already exists on the disk. The old file then can be deleted using a simple VAX/VMS command.

6.5 DELETE RELATIONS AND SPATIAL DATA STRUCTURES FROM MEMORY

6.5.1 DELETING A RELATION FROM MEMORY

A relation is deleted from memory as follows. Given a relation to be deleted from memory, the structure (either a TREE or a LIST depending on the type of the relation) pointed to by the header of the relation is deleted first. Each N-tuple in the relation is deleted one by one. A standard Pascal function DISPOSE [JENSK74] is used to delete the individual structures which store the components of the N-tuples. Whenever any data structure is deleted using the funcion DISPOSE, the space occupied by that structure is made known to the database system by subtracting it from the global variable SPACE.

If any component of an N-tuple is a spatial data structure, the USE_CNT field of the spatial data structure header is subtracted by one. If the USE_CNT of this SDS is not zero (that is, if the spatial data structure is referenced by some other relation which is in memory), the spatial data

structure is not deleted from memory. If the USE_CNT of the SDS is zero, it is checked to determine whether or not the entire structure of the SDS is in memory. If not, the SDS header is deleted from memory and the space is recovered. This means that, if only the header of that spatial data structure is in memory, and it is not referenced by any other relation, then only the header is deleted from memory (a later section describes how a spatial data structure along with its header is deleted from memory).

After the structures pointed to by the relation header are deleted, the header itself is freed only if its USE_CNT is less than one, that is, only if the relation is not associated with any spatial data structure which is currently in memory. If the relation is attached to any of the spatial data structures in memory, (this is indicated by the USE_CNT greater than zero) the header of the relation is not deleted and remains in memory. If the header is deleted, the space it occupies is subtracted from the global variable SPACE.

6.6 DELETING A SPATIAL DATA STRUCTURE FROM MEMORY

If any spatial data structure is to be deleted from memory, its RELATION LIST is traversed, and the header of each relation associated with it is deleted from memory if and only if it has a USE_CNT less than one and the structure

pointed to by it is nil. That is, if the relation is associated with some other spatial data structure which is in memory, or if the entire relation is present in memory, then the header of that relation is not deleted from memory.

After deleting the RELATION_LIST the header of the spatial data structure is deleted from memory and the space is recovered only if its USE_CNT is less than one, that is, only if it is not referenced by any relation in the system which is presently in memory.

The algorithm for deleting a relation or spatial data structure avoids the problem of deadlock even in case of recursive structures. For example, consider a situation where a spatial data structure 'S' has a relation 'R' associated with it, and the relation 'R' has an N-tuple which has a component that points to the spatial data structure 'S' itself. Let us further assume that the entire structures of both of them are in memory. Now, if the SDS is to be deleted from memory, the relation 'R' will not be deleted from memory because its entire structure is in memory. Also, the SDS header will not be deleted as it will have a USE_CNT equal to one (because it is referenced by the relation 'R' which is in memory). Thus, the component of the N-tuple in relation 'R' will still be pointing to the SDS

header. Although, the SDS header is not deleted, the list of RELATION_CELLs linked to it will still be deleted to recover some space.

On the other hand, if the relation 'R' is to be deleted from memory, the spatial data structure 'S' will not be deleted because its entire structure is in memory. Also, the relation header will not be deleted because the relation 'R' is referenced by the SDS 'S' which is in memory. All the other components of the relation will be deleted to recover space.

6.7 THE PAGING ALGORITHM

We describe the paging algorithm using the operations described earlier in this Chapter.

The database can be accessed by the user only through the query language interpreter explained in Chapter VI. The query language command DB_LOAD (refer to Chapter VI) loads the database and gives the user an access to it. When this command is invoked, the names of the relations and the spatial data structures in the system are read from their respective files into the dictionaries, REL_DICTIONARY and RDS_DICTIONARY, respectively, in memory. All the other fields of both the dictionaries are initialized to either

zero or nil appropriately. The arrays RDS_IN_CORE and REL_IN_CORE are initialized to zero. The variables representing the least and the most recently read spatial data structures and relations are also initialized to zero.

The data elements are structured in the form of relations and spatial data structures. The user can access any information in the relations or spatial data structures using the query language commands. The relations and spatial data structures are referenced by their names.

Whenever any relation or any spatial data structure that is not in memory is referenced, that particular relation or SDS is read into memory from the disk. Each user of the database is assigned a space quota; that is, there is a limit on the amount of memory space which he/she can utilize. Every time any relation or a spatial data structure is read into memory, or every time a new relation or a spatial data structure is created, the amount of space occupied by all the structures is compared with the space quota of the user. If the amount of space occupied is greater than the space quota, the least recently read relations and/or spatial data structures are transferred back to the disk (if their corresponding structures have been changed), and in any case their structures are deleted from memory until the

space occupied is less than the space quota. The headers of those relations and/or spatial data are deleted from memory only if their reference counts, represented by the USE_CNT field, are zero. If the number of relations (represented by NREL_IN_CORE) is greater than the number of spatial data structures (represented by NRDS_IN_CORE) in memory, then a relation is selected for transferring and deleting from memory. On the other hand, if the number of spatial data structures is greater than the number of relations in memory, then a spatial data structure is selected. The index of the least recently used relation or spatial data structure is obtained from the array REL_IN_CORE or RDS_IN_CORE, respectively. Figure 6.1 illustrates the paging algorithm in the form of a Pascal-like procedure.

The database is unloaded from memory by the query language command DB_UNLOAD. When this command in issued, all the relations and spatial data structures which are in memory are written back to the disk if their structures have been changed, and then are deleted from memory. The two dictionaries are then written to the disk in their respective files and are deleted from memory. Thus, after this command the user looses the access to the database.

Let R1 be the name of the relation referenced by the user (the same algorithm holds good if a spatial data structure is referenced). For any relation or spatial data structure, the operations, creating a header, reading into memory, transferring back to the disk, and deleting from memory are as explained earlier in the chapter.

```
Search the REL_DICTIONARY for R1.
Let I1 := index of R1 in the dictionary.
if REL_ADDRESS[ I1 ] := nil
   then Create a new header for R1.
```

Now let us assume that the user references some information in relation R1.

```
Let WS = space quota of the user.
Let MAX_REL = Maximum number of elements in the array
              REL_IN_CORE.
Let MAX_RDS = Maximum number of elements in the array
              RDS_IN_CORE.
(* Check if the relation R1 is already in memory.     *)

Search the REL_DICTIONARY for R1.
Let I1 := index of R1 in the dictionary.

if REL_TAG := 0
```

```
   then begin    (* R1 is not in memory. *)

(* If the relation is not in the memory then read      *)
(* it in the memory.                                   *)

(* First check if there is sufficient space available  *)
(* for this relation. If not, transfer some relations  *)
(* and/or spatial data structures to the disk and then *)
(* delete from the memory.                             *)

while ( SPACE >= WS ) do begin
   if ( NREL_IN_CORE > NRDS_IN_CORE )
      then begin    (* Transfer and Delete a relation. *)
         K := REL_IN_CORE [ LRU_REL_IN_CORE] ;
         R2 := REL_NAME [ K ];
         if ( REL_CHANGE [ K ] = 1 )
            then Transfer R2 to the disk;
         Delete R2 from memory;
         LRU_REL_IN_CORE := ( LRU_REL_IN_CORE + 1 ) mod
                              MAX_REL;
         NREL_IN_CORE := NREL_IN_CORE - 1;
      else begin          (* Transfer and Delete an SDS *)
         K := RDS_IN_CORE [ LRU_RDS_IN_CORE ];
         S2 := RDS_NAME [ K ];
         if ( RDS_CHANGE [ K ] = 1 )
```

```
            then Transfer S2 to the disk;
          Delete S2 from memory;
          LRU_RDS_IN_CORE := ( LRU_RDS_IN_CORE + 1 ) mod
                                MAX_RDS;
          NRDS_IN_CORE := NRDS_IN_CORE - 1;
      end;
   end;

(* Now read the relation into memory.                  *)
Read R1 into memory;
MRU_REL_IN_CORE := ( MRU_RDS_IN_CORE + 1 ) mod MAX_REL;
REL_IN_CORE [ MRU_REL_IN_CORE ] := I1;

end;
```

Figure 6.1 The Paging Algorithm
Using Pascal-like Conventions.

Chapter VII

PERFORMANCE MEASUREMENT

In this chapter we describe several experiments which were performed to measure the performance of our geographic database system. Most of the experiments were conducted using the query language interpreter described in Chapter V. For some experiments there are no structures in memory to start with, while, some experiments require setting up the entire database in memory before invoking the interpreter.

7.1 INITIAL SET UP OF THE DATABASE

7.1.1 SET UP THE DATABASE IN MEMORY

The entire database was set up in the memory from the original data files (refer to Chapter IV); that is all the relations and spatial data structures in the system were created in memory.

Mean CPU time = 1 min. 30.14 secs.

Number of page faults = 3101

Total space occupied in memory = 51737 words.

The following observations indicate how many times the primitive operations were performed while setting up the database.

1) Create an SDS or relation header (RDS_NEW) = 1512.

2) Attach a relation to an SDS (REL_ATTACH) = 1004.

3) Add an N-tuple to a TREE relation (NT_ATTACH) = 1754.

4) Add an N-tuple to a LIST relation (LNT_ATTACH) = 2790.

From these observations we obtain the following results:

Total number of relations in the system = 1004.

Total number of SDS's in the system = 508.

Total number of N-tuples in all the relations = 4544.

7.1.2 TRANSFER THE DATABASE TO THE DISK

The databse was set up in memory and then all the SDS's, relations, and dictionaries were transferred to the disk from memory (refer to Chapter IV). The CPU time and page faults for the combined operations of setting up and transferring the databse was noted.

Mean CPU time = 2 mins. 54.25 secs.

Number of page faults = 6685.

Subtracting the time for setting up the database, we get the CPU time to transfer the databse from memory to the disk.

Mean CPU time to transfer the databse = 1 min. 24.11 secs.

7.2 INDIVIDUAL PRIMITIVE OPERATIONS

In this section we evaluate the performance of the individual primitive operations such as, creating a new relation or a spatial data structure, searching the dictionaries, etc. For each experiment, a data file consisting of interpreter commands was generated. The input for the interpeter was taken directly from these files. This involved certain overheads such as, invoking and closing the interpreter, and executing the command 'INPUT>'. Also, some experiments required setting up the entire database in memory before invoking the interpreter. First, we compute this overhead.

7.2.1 THE OVERHEAD

7.2.1.1 BIBLIOGRAPHY

In this experiment the interpreter was invoked and then closed immediately without executing any command.

Mean CPU time = 00.84 secs.
Number of page faults = 379.

We will refer to this overhead of invoking and closing the interpreter as overhead (A).

7.2.1.2 BIBLIOGRAPHY

The interpreter was invoked and the command 'INPUT>' was executed with a blank input file, and the combined overhead of invoking the interpreter, executing the command, and closing the interpreter was determined.

Mean CPU time = 00.95 secs.

Number of page faults = 409.

We will refer to this overhead as overhead (B).

7.2.1.3 BIBLIOGRAPHY

The entire database was set up in memory first, the interpeter was invoked next, and the command 'INPUT>' was executed with a blank input file. The combined overhead was as follows:

Mean CPU time = 1 min. 30.80 secs.

Number of page faults = 3158.

We will refer to this overhead as overhead (C).

7.2.2 LOAD THE DATABSE IN MEMORY

In this experiment the database was loaded from the disk into memory by using the command 'DB_LOAD' (refer to Chapter V). Recall that this command initializes both the dictionaries and reads the names of all the relations and spatial data structures in the system into memory. The mean CPU time and the number of page faults were observed to be as follows:

Mean CPU time = 05.49 seconds.
Number of page faults = 629.

However this includes overhead (A). Therefore subtracting CPU time for overhead (A) we get,

Mean CPU time for loading the database into memory = 04.65 seconds.

7.2.3 CREATE A RELATION OR SDS HEADER

Headers (an RDS) for 1000 new relations and spatial data structures were created using the interpreter commands 'ALLOC_REL' and 'ALLOC_RDS'. The following observations were noted.

Mean CPU time = 56.38 seconds.
Number of page faults = 763.

This includes the overhead (B). Subtracting we get,

Mean CPU time for creating 1000 relation or SDS headers = 55.43 seconds. Hence,

Mean CPU time for creating a new relation or SDS header = 00.05543 seconds.

7.2.4 SEARCH THE DICTIONARIES FOR THE GIVEN NAME

For this experiment all the relations and spatial data structures were structured in memory, and then the dictionaries were searched for 1000 different relation and SDS names to get pointers to their headers on the top of the interpreter stack using the command 'FIND'. Given a name, the command 'FIND' first carries out a binary search in the RDS_DICTIONARY (refer to Chapter V). If the name is found in the RDS_DICTIONARY the pointer to the corresponding SDS header is returned on the top of the interpreter stack. If the name is not found in the RDS_DICTIONARY, a binary search is performed in the REL_DICTIONARY and a pointer to the relation header is returned to the top of stack if the name is found. The status of the search is also returned. The following observations were noted.

Mean CPU time = 2 mins. 01.18 seconds.

Number of page faults = 3139.

In order to compute the mean CPU time for one search, we need to subtract the CPU time for overhead (C). Therefore,

Mean CPU time for 1000 FIND operations = 30.38 seconds.

Hence, mean CPU time for one FIND operation = 00.03038 seconds.

7.2.5 GET THE INDEX OF A RELATION OR SDS

The databse was first loaded into memory, using the command DB_LOAD, and then indices of 1000 names in their respective dictionaries were obtained using the commands 'REL_INDEX' and 'RDS_INDEX'.

Mean CPU time = 36.65 seconds.
Number of page faults = 869.

However, this includes the overhead (A) and the overhead of invoking the command DB_LOAD. Therefore, subtracting we get,

Mean CPU time for 1000 RDS_INDEX and/or REL_INDEX operations = 31.16 seconds.

In other words,

Mean CPU time for one RDS_INDEX or REL_INDEX operation = 00.03116 seconds.

7.2.6 ATTACH A RELATION TO AN SDS

In this experiment, initially, all the relations and SDS's were structured in memory, and then same relation was attached to 100 different spatial data structures using the command 'REL_ATTACH'. The results were as follows:

Mean CPU time = 1 min. 37.52 seconds.
Number of page faults = 3203.

This includes overhead (C) and an overhead for 200 FIND operations (100 operations to get a pointer to the relation header and 100 operations to get pointers to the SDS headers to which the relation is to be attached). Therefore, subtracting these overheads we get,

Mean CPU time for 100 REL_ATTACH operations = 00.65 seconds.
Or, we can say, the mean CPU time for one REL_ATTACH operation = 00.0065 seconds.

7.2.7 ATTACH AN N-TUPLE TO A RELATION

Again, the structures for all the relations and SDS's were set up in memory, and the same N-tuple was inserted into 100 different relations.

Mean CPU time = 1 min. 39.29 seconds.
Number of page faults = 3370.

This includes overhead (C) and an overhead of 100 FIND operations for getting a pointer to the relation headers. Subtracting we get,

Mean CPU time for 100 NT_ATTACH operations = 05.45 seconds.

It should be noted that the mean CPU time for an NT_ATTACH operation depends upon the size of the relation in which the N-tuple is inserted. This is so because, in TREE relations the N-tuple is inserted at an appropriate order, and hence involves searching the tree for the appropriate place. In LIST relations this operation inserts the N-tuple at the end of the list.

7.3 EVALUATION OF THE PERFORMANCE OF THE SYSTEM

In order to evaluate the performance of the entire data-base system using the memory management scheme (refer to

Chapter VI) a test program consisting of query language commands was created. These commands read each structure in the system (each relation and spatial data structure) into memory and printed them on a specified output file. Eighteen different runs were performed for three values of VAX working sets[15] and five different values of space quota (refer to Chapter VI). The value 'space quota' represents the maximum number of words occupied by all the structures in memory at any one time. The remaining three runs were performed without our memory management scheme for the same three VAX working sets.

The results are tabulated in the following pages. The following abbreviations are used in the results.

RDS_NEW : Number of times the primitive RDS_NEW is called. The primitive RDS_NEW creates a new header for any relation or spatial data structure.

REL_ATTACH : Number of times the primitive operation REL_ATTACH, which attaches a relation to a spatial data structure, is called.

---

15) VAX working set is defined as the maximum number of pages that can be resident in memory at any one time for a given process.

NT_ATTACH : Represents the number of n-tuples inserted in relations which store the n-tuples in a tree structure.

LNT_ATTACH : Represents the number of n-tuples inserted in the relations which store the n-tuples in a list structure.

REL_IN : Represents the number of relations which are read into memory.

REL_OUT : Indicates the number of relations which are deleted from memory.

SDS_IN : Represents the number of spatial data structures read into memory.

SDS_OUT : Represents the number of spatial data structures deleted from memory.

HEADER_OUT : Represents the number of relations or SDS headers deleted from memory.

VAX WORKING SET : 700 PAGES.

SPACE QUOTA	INFINITE	20,000 WORDS
CPU TIME	7 min. 49.41 sec.	8 min. 12.99 sec.
PAGE FAULTS	8058	11068
DIRECT I/O	3142	3703
BUFFERED I/O	12138	14382
RDS_NEW	1512	2068
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	0	834
SDS_IN	508	508
SDS_OUT	0	335
HEADER_OUT	0	1297

VAX WORKING SET : 700 PAGES.

SPACE QUOTA	15,000 WORDS	10,000 WORDS
CPU TIME	8 min. 20.56 sec.	8 min. 03.86 sec.
PAGE FAULTS	10150	8669
DIRECT I/O	4037	4188
BUFFERED I/O	15718	16322
RDS_NEW	2402	2553
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	896	949
SDS_IN	508	508
SDS_OUT	397	451
HEADER_OUT	1869	2280

VAX WORKING SET : 700 PAGES.

SPACE QUOTA	5,000 WORDS	2,500 WORDS
CPU TIME	7 min. 40.00 sec.	7 min. 28.15 sec.
PAGE FAULTS	6931	6153
DIRECT I/O	4373	4561
BUFFERED I/O	17062	17816
RDS_NEW	2738	2926
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	957	969
SDS_IN	508	508
SDS_OUT	472	476
HEADER_OUT	2528	2737

VAX WORKING SET : 500 PAGES.

SPACE QUOTA	INFINITE	20,000 WORDS
CPU TIME	8 min. 14.85 sec.	8 min. 28.38 sec.
PAGE FAULTS	62084	66928
DIRECT I/O	3142	3703
BUFFERED I/O	12138	14382
RDS_NEW	1512	2068
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	0	834
SDS_IN	508	508
SDS_OUT	0	335
HEADER_OUT	0	1297

VAX WORKING SET : 500 PAGES.

SPACE QUOTA	15,000 WORDS	10,000 WORDS
CPU TIME	8 min. 30.14 sec.	8 min. 15.17 sec.
PAGE FAULTS	37669	40367
DIRECT I/O	4037	4188
BUFFERED I/O	15718	16322
RDS_NEW	2402	2553
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	896	949
SDS_IN	508	508
SDS_OUT	397	451
HEADER_OUT	1869	2280

VAX WORKING SET : 500 PAGES.

SPACE QUOTA	5,000 WORDS	2,500 WORDS
CPU TIME	7 min. 48.67 sec.	7 min. 37.95 sec.
PAGE FAULTS	12607	9833
DIRECT I/O	4373	4561
BUFFERED I/O	17062	17816
RDS_NEW	2738	2926
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	957	969
SDS_IN	508	508
SDS_OUT	472	476
HEADER_OUT	2528	2737

VAX WORKING SET : 300 PAGES.

SPACE QUOTA	INFINITE	20,000 WORDS
CPU TIME	44 min. 25.58 sec.	29 min. 16.84 sec.
PAGE FAULTS	6701923	3795424
DIRECT I/O	3142	3703
BUFFERED I/O	12138	14382
RDS_NEW	1512	2068
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	0	834
SDS_IN	508	508
SDS_OUT	0	335
HEADER_OUT	0	1297

VAX WORKING SET : 300 PAGES.

SPACE QUOTA	15,000 WORDS	10,000 WORDS
CPU TIME	15 min. 41.32 sec.	10 min. 15.37 sec.
PAGE FAULTS	1250233	332681
DIRECT I/O	4037	4188
BUFFERED I/O	15718	16322
RDS_NEW	2402	2553
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	896	949
SDS_IN	508	508
SDS_OUT	397	451
HEADER_OUT	1869	2280

VAX WORKING SET : 300 PAGES.

SPACE QUOTA	5,000 WORDS	2,500 WORDS
CPU TIME	9 min. 07.27 sec.	8 min. 35.41 sec.
PAGE FAULTS	202703	147618
DIRECT I/O	4373	4561
BUFFERED I/O	17062	17816
RDS_NEW	2738	2926
REL_ATTACH	1007	1007
NT_ATTACH	1758	1758
LNT_ATTACH	2790	2790
REL_IN	1004	1004
REL_OUT	957	969
SDS_IN	508	508
SDS_OUT	472	476
HEADER_OUT	2528	2737

We observe the following from these results.

With only VAX memory management (SPACE QUOTA value equal to infinite) as we lower the VAX working set, the performance of the system greatly deteriorates. The CPU time required for the same task increases considerably for the lower working set sizes (especially for the working set of 300 pages) due to a large increase in the number of system page faults. Only, the direct I/O and the buffered I/O remain constant because the number of disk accesses remain same for any value of the VAX working set.

However, it can be observed that when we employ our memory management scheme, the performance of the system improves greatly. With the VAX working set remaining constant, as we reduce the value of SPACE QUOTA, the number of relations and spatial data structures deleted from memory increase, as only a smaller space is available to store the structures in memory. Also, the direct I/O increases due to the increase in the number of disk accesses. This is implicitely indicated by the increase in the number of headers created (RDS_NEW). Thus, lowering the value of SPACE QUOTA causes certain overheads of disk accesses, file I/O, and deleting the structures from memory. However, with the smaller values of the SPACE QUOTA, the memory management is done

more often by the database system which reduces the overhead of the VAX memory management. This causes a reduction in the number of system page faults and eventually decreases the CPU time for the same task. Especially, for smaller values of VAX working set, the improvement in the performance of the system is the most significant. This is evident from the fact that with a VAX working set of 300 pages, if we do not emply our memory management, the test program takes about 44 minutes of CPU time for execution, and causes more than six million page faults. On the other hand, with our memory management, using a SPACE QUOTA value of 2500 words, the same test program is executed in just 8 minutes and 35 seconds. The number of system page faults are also reduced to only about 147618.

7.4 FIND THE CPU TIME PER PAGE FAULT

In order to compute the CPU time per page fault a similar test program was created as in the case of the preceding experiment. The same test program was executed using different VAX working sets without the database memory management. That is, the SPACE QUOTA value was assumed to be infinite and only VAX paging was allowed. Thus, no structure was deleted or transferred from memory once it was read into memory. The following results were observed.

WORKING SET	MEAN CPU TIME	PAGE FAULTS	DIRECT I/O	BUFFERED I/O
700	7 min. 03.99 sec.	12153	3142	12138
600	7 min. 06.79 sec.	18412	3142	12138
500	7 min. 23.78 sec.	63770	3142	12138
400	10 min. 09.17 sec.	468726	3142	12138
300	46 min. 10.53 sec.	6369424	3142	12138

It can be observed that both the direct I/O and the buffered I/O remain same for all the runs. The difference in CPU time is therefore due to the difference in the number of page faults. Thus, we can compute the CPU time per page fault.

Mean CPU time per page fault = 00.3898 msec.

Variance = .001

Chapter VIII

CONCLUSIONS AND FUTURE WORK

In this thesis, we have shown how the concept of spatial data structures can be used to develop and implement a spatial information system for cartographic applications. We have demonstrated that geographic data can be represented in the entity-oriented relational structures. We have implemented an all-in-core version of the system and also developed a memory management system which controls the transfer of structures between disk and internal storage. It was observed that our memory management algorithm is quite efficient and greatly reduces the overhead of the VAX/VMS memory management, thus significantly improving the performance of the system.

As we mentioned before, this system was implemented on an experimental basis. We developed the inner level structures and implemented certain primitive operations. The next step in building a real geographic system based on these concepts would be to implement high-level geographic algorithms using the primitive structural operations now provided and to evaluate their performance. These high-level algorithms would include the set operations such as: union, intersection, and difference; and the relational operations such as: projection, selection, join, and division on the relations

and spatial data structures. The possibility of the design of a parallel architecture for performing the relational operations can not be ruled out.

The stack oriented query language is only a first step towards communication between human and database system. The interpreter could be modified to include a help facillity for the commands and to provide error checking. Also, making the interpreter more interactive would make it easier to use. The physical structure of the database on the secondary device was kept simple. Future work should also involve determining the optimal record structure and file organization on the disk to provide both fast query response and efficient storage utilization.

Another important area of future work would be to incorporate an intelligence in the system (work is already being conducted on this issue). The basic idea is that, any query would be parsed and analyzed by the intelligent system using the predefined production rules. Several paths to satisfy the query would be found, and the most efficient path would be chosen. This would automatically generate code which would eventually call the primitive operations already provided, to satisfy the query.

In short, we can conclude that this was only the begining and there is much more to do. However, this 'much more' is not an impossible task to perform. We finish our discussions by expressing an optimism that in the near future an efficient and intelligent real information system developed on these concepts will come into existance.

BIBLIOGRAPHY

[CARLE74] Carlson, E.D., J.L. Benett, G.M. Gidding, and P.E. Mantey, The Design and Evaluation of an Interactive Geo-Data Analysis and Display System, Proceedings of IFID Congress 74, North Holand Publishing Company, Amsterdam, 1974.

[CHAMD74] Chamberlin, D.D., R.F. Boyce, SEQUEL: A Structured English Query Language, Proceedings of 1974 ACM SIGMOD Workshop on Data Description, Accesses and Control.

[CHANS] Chang, Shi-Kuo and T.L.Kunii, Pictorial Database Systems, A Project Report, Knowledge Systems Laboratory, University of Illinois at Chicago Circle.

[CODDE70] Codd, E.F., A Relational Model of Data for Large Shared Databases, Communications of ACM, Vol. 13, No. 6, June 1970, pp. 377-389.

[DATEC77] Date, C.J., An Introduction to Database Systems, 2nd Edition, Addison Wesley, New York, New York, 1977.

[EDWAR77] Edwards, R.L., R. Durfee, and P. Coleman, Definition of a Hierarchical Polygonal Data Structure and the Associated Conversion of a Geographic Base File from Boundary Segment Format, An Advanced Study Symposium on Topological Data Structure for Geographic Information Systems, Harvard University, Cambridge, Massachusetts, October 1977.

[GOA75] Go, A., M. Stonebraker, and C. Williams, An Approach to Implementing a Geo-Data System, Memo No. ERL-M529, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, 1975.

[GOLDC76] Gold, C., Triangular Element Data STructures, Users Applications Symposium Proceedings Services, Edmonton, Alberta Canada, 1976.

[HAGAP80] Hagan, P.J., A Network Data Model for Cartographic Features, Doctor of Science Dissertation, Sever Institute of Technology, Washington University, Saint Louis, Missouri, May 1980

[HOROE77] Horowitz E. and S. Sahani, Fundamentals of Data Structures, Computer Science Press, Inc., Potomac Maryland 20852, 1977.

[JENSK74] Jensen, K. and Wirth N., PASCAL User Manual and Report, Second Edition, Springer-Verlag, New York, 1974.

[LABCG74] Laboratory for Computer Graphics, POLYVRT: A Program to convert Geographic Base Files, Harvard University, Cambridge, Massachusetts, 1974.

[MINDG81]Minden, G., A Systems Manual for a Query Language Interpreter, Unpublished Project Report, University of Kansas, Lawrence, Kansas.

[MODEM77] Modeleski, M., Topology for INGRES; An Approach to Enhance Graph Property Recognition by GEOQUEL, Geographic Database Coordinator, Asociation of Bay Area Governments, Berkeley, California 94705, 1977.

[MOORC74] Moore, C.H., FORTH, A New Way to Program a Computer, Astronomy and Astrophysics Supplement, 1974, No. 15, pp. 497-511.

[PEQUD79] Pequet, D.J., A Raster Mode Algorithm for Interactive Modification of Line Drawing Data, Computer Graphics and Image Processing, Vol. 10, 1979, pp. 142-158.

[PEUCT75] Peucher, T.K. and N. Chrisman, Cartograohic Data Structures, The American Cartographer, Vol. 2, No. 1, April 1975, pp. 55-69.

[SHAPL79] Shapiro, L.G. and R.M. Haralick, A Spatial Data Structure, Technical Report #CS79005-R, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061, August 1979.

[SWITW75] Switzer, W.A., The Canada Geographic Information System, Automation in Cartography, eds. J.M. Wilford-Brickwood, R. Bertland, and L. Van Zuylen, International Cartographic Association, The Netherlands, 1975.

[TOMLR76] Tomlinson, R.F., H.W. Calkins, and D.F. Marble, Computer Handling of Geographical Data, Paris: UNESCO Press, 1976.

[USBUR70] U.S. Bureau of the Census, 'Census Use Study: The DIME Geocoding System, Report No. 4, Washington D.C., 1970.

[WAGLS78] Wagle, S.G., _Issues in the Design of a Geographical Data Processing System: A Case Study_, PH.D. Dissertation, The University of Nebraska, Lincoln, 1978.

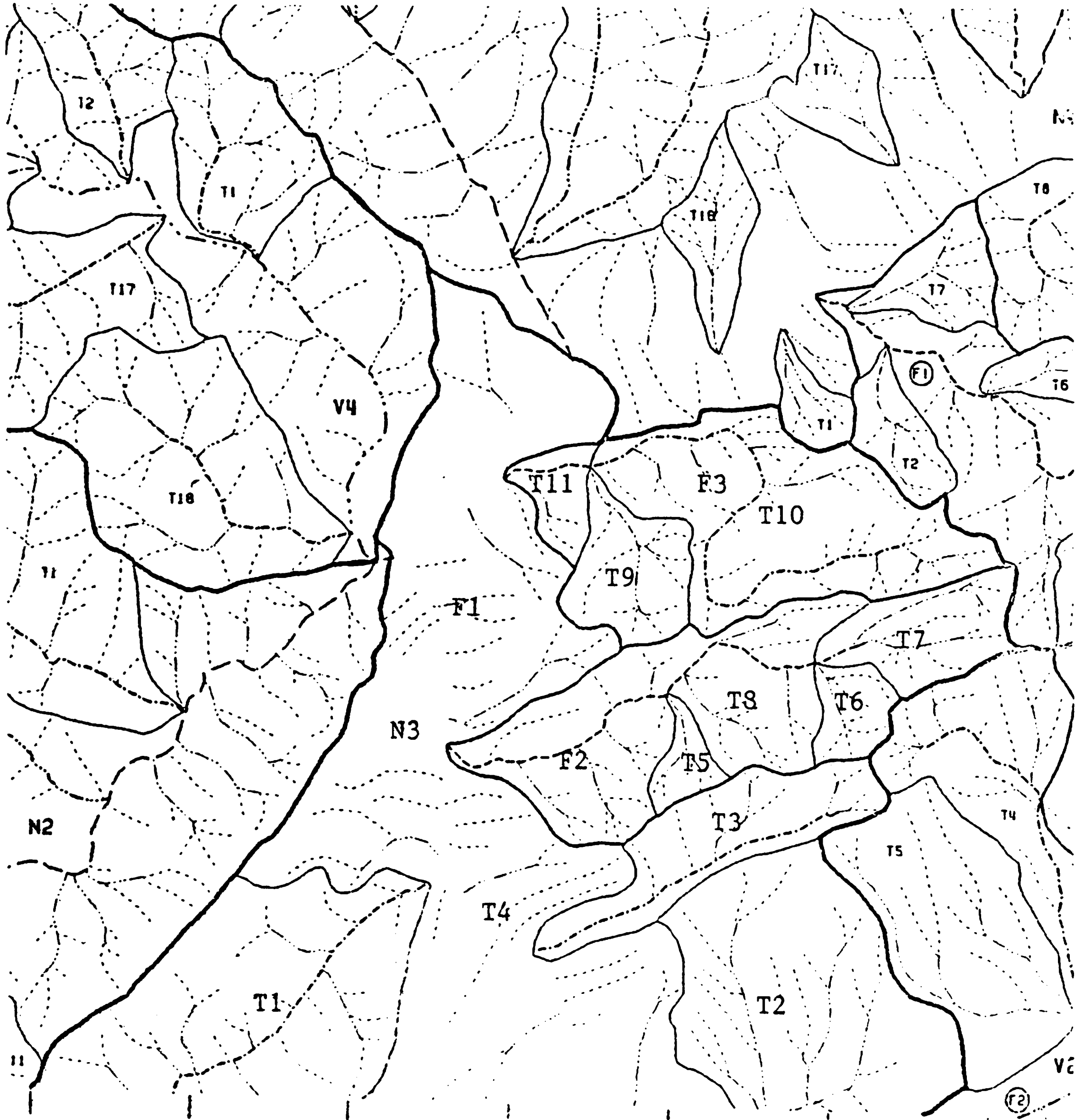
[WEBEW78] Weber, W., Three Types of Map Data Structures, Their ANDs and NOTs, and a Possible OR, _Proceedings of the First International Advanced Study Symposium on Topological Data Structures for Geographic Information Systems_, Harvard University, Cambridge, Masachusetts, 1978.

[WIEDG77] Wiederhold, G., _Database Design_, McGraw-Hill, New York, 1977.

Appendix A

A

A Digitized Map of the Watershed Area N3



The vita has been removed from the scanned document

AN EXPERIMENTAL SPATIAL INFORMATION SYSTEM

by

Prashant D. Vaidya.

(ABSTRACT)

Computer representation of the continuous two-dimensional features on a map is complicated by the spatial properties not found in typical alphanumeric data. We have designed an entity-oriented relational system for representing the cartographic data, using the concept of spatial data structures. Each geographic entity such as a region, road, or city is represented by a set of relations describing its properties, its related entities, and all the relationships among them. The thesis presents the description of the first experimental cartographic information system based on these concepts to store and retrieve watershed data for a portion of the Wise county in the state of Virginia. The thesis describes the logical structure of the database, the physical structures in memory and on the disk, a query language interpreter which is used to access the information in the database, and a memory management scheme to transfer the structures back and forth between memory and secondary device.