Design and Architectural Implications of A Spatial Information System

Prashant D. Vaidya
Linda G. Shapiro
Robert M. Haralick
Gary J. Minden*

CS820003-R

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

December 1981

---

*Department of Electrical Engineering, University of Kansas, Lawrence, KS 66045

ABSTRACT

Image analysis, at the higher levels, works with extracted regions and line segments and their properties, not with the original raster data. Thus a spatial information system must be able to store points, lines, and areas as well as their properties and interrelationships. In a previous paper (Shapiro and Haralick [22]), we proposed for this purpose an entity-oriented relational database system. In this paper we describe our first experimental spatial information system, which employs these concepts to store and retrieve watershed data for a portion of the state of Virginia. We describe the logical and physical design of the system and discuss the architectural implications.

Key Words: spatial information system, relational database system, geographic information system,

I. INTRODUCTION

A digital image is a raster data structure. It consists of one or more bands of spectral and/or symbolic information, each band consisting of a rectangular matrix of elements called pixels. For example, a color image of an outdoor scene might be represented by three spectral bands: red, green, and blue. A land-use map of a nation might be represented by a single symbolic band where the value of each pixel defines the land use in that part of the map. The raster representation allows a choice of resolution, from fine to coarse, and retains the spatial relationships among the pixels of the image or map. Certain operations such as smoothing or sharpening and the so-called "neighborhood operations" can be performed very efficiently on the raster structure. However, these operations are generally of use only in the early, low-level phases of image or scene analysis. The mid- and higher-level intelligent processes require more concise and meaningful structures than pixels. At this level, the primitives are points, regions, and lines. These are the same primitives that are required in geographic or spatial information systems.

Once an image has been transformed from pixel level to edge-region level, the information must be organized for efficient access. A spatial information system provides the hardware and software support necessary for storage and retrieval of spatial data. In a previous paper (Shapiro and Haralick, 1980 [22]), we suggested a relational approach to designing a spatial information system. Our approach has the advantage of allowing either vector or raster data or both in a unified framework suitable for high-level query. In this paper, we describe the implementation of this approach in our first experimental database system for storage and retrieval of watershed data. In Section II we review the definition of the general spatial data structure that is the building block of the system and give the logical design of the experimental database. In Section III we describe the physical design of the system. In Section IV we describe the query language interpreter that is used to communicate interactively with the system and define the low-level and high level operations required to answer queries. In Section V we discuss the architectural implications of the system. In the remainder of this section, we give a brief review of related work on representations used by spatial information systems in both geographic and computer vision applications.

General Representations

Most spatial information systems have been organized either as a set of polygons or as a raster of grid cells. Conceptually, in the raster representation of spatial data, a regular grid is placed over a picture or map, and certain properties such as spectral reflectance and texture (for pictures) or population, vegetation type, and soil type (for maps) are recorded for each grid cell. The raster representation retains the spatial relationships of the photographic image or map, but computational and input/output efficiency is sacrificed for some operations. Another disadvantage is that the locational precision is not inherent in the size of the grid chosen.

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

Geographic Systems

Of the geographic spatial information systems, the Canadian Geographic Information System (CGIS) [23] is one of the earlier and successful ones. Two types of files are used: the image data set which contains line segments defining the polygons that represent regions and the descriptive data set, which contains the user assigned identifiers, centroid, 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 line segments that continue bounding the polygons on the left and on the right.

In the late 1960's the U.S. Bureau of the Census developed the Dual Independent Map Encoding (DIME) [25] concept 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 polygons on the right and left sides of the segment. The data structure was kept simple at the expense of extra processing time for certain operations. For instance, determining that 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.

The POLYVRT system [16] is similar to the DIME system described above. In the POLYVRT data structure, the polygons are formed by chains (sequences of points). 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. The polygon, however, is established as a separate entity linked to the chains which compose it. This allows 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.

GEOGRAF is a system proposed by Peucher and Chrisman [21] 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 advantages and drawbacks of the grid formatted data structure are almost perfectly complementary to those of the topological or polygonal data structure. With this in mind Weber [27] has proposed a combination of locational data structure and grid formatted data structure. The operations on this data structure are defined in a hierarchical manner, so that the transition from grid formatted to linear (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 traverse the structure and no explanation of how to store the linear representation and the grid formatted representation.

IBM's Geo-data Analysis and Display System (GADS) [2] is the first documented geographic database system which used the approach of relational database. It is the most ambitious of the geographic information system in terms of flexibility and interactive problem solving capability. This system has database management facilities and supports database integrity and different user views of a pictorial database. 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.

The GEO-QUEL system developed at U. C. Berkeley is another system which uses the relational database approach to manipulate geographic data [10]. 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 as a 9-ary relation. A query language QUEL which is similar to SEQUEL [3] is used to interrogate the system. The system can handle simple queries about a map and can display information from 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 [18] 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.

Hagan [12] 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. This two level structure has the advantage of allowing inter-feature relationships to be computed using direct links instead of searches.

Computer Vision Systems

S. K. Chang et al [6] designed and implemented a pictorial database system for storage and retrieval of tabular, graphical, and image data. Logical pictures are extracted from images and stored in relational form, while physical images are retained in a separate image store. A generalized zooming technique [5] has been implemented to allow for flexible hierarchic information retrieval.

N. S. Chang and Fu [4] designed a relational pictorial database system where access is through a high-level relational query language called "Query-By-Pictorial-Example." The system extracts structures and features from the original images and stores the information in the relational database, while saving the images in separate storage. The users' queries are answered by the database system if possible. Otherwise, selected images are retrieved and processed further to obtain the answer.

Hanson and Riseman [13] have designed and partially implemented an integrated computer vision system (called VISIONS) for interpreting natural scenes. Although this system includes a variety of elements such as three-dimensional models, hierarchic process control, and low level image processing that are not present in the geographic systems, it shares with them the intermediate level representation consisting of lines, regions, and points. At this level, they use a partitioned directed graph structure to represent the relationships among regions, their line segments, and their end points.

Of course there are too many computer vision systems being developed to mention them all. However, they all have in common a need for storage and retrieval of the mid- and high-level information extracted from the image. (See Marr [17] and Barrow and Tenenbaum [1] for important discussions of mid-level information.) The structure that we have proposed represents a unified approach to storing such information in a universal structure.

II. LOGICAL DESIGN

II.1 The Spatial Data Structure

In this section we define a general spatial data structure that can be used to represent any spatial information or relational data in raster, vector, or tabular format.

An _atom_ is a unit of data that will not be further broken down. Integers and character strings are common examples of atoms. An _attribute-value table_ 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.

A _spatial data structure_ D is a set D = {R1,...,RK} of relations. Each relation Rk has a dimension Nk and a sequence of domain sets S(1,k),...,S(Nk,k). That is for each k = 1,...,K, Rk $\subseteq$ S(1,k) X ... X 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, 2) that it will often be possible to describe operations on the structure by simple recursive algorithms, and 3) that it can naturally represent both relational and hierarchical dependencies.

A spatial data structure represents a spatial 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 spatial entities. Each spatial data structure has 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 spatial 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 spatial 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.

II.2 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 (Date, 1977 [8]). 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 _schema_

(Wiederhold, 1977 [28]). A schema defines the logical structure of the database without any storage/access details.

In this section we develop the schema for the spatial database system. The spatial data structure is the primitive or the building block of the system. Each spatial data structure represents one particular type of spatial data 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.

Our first use of the system is with geographic data. The data* used in this system is of two types: stream data and road data. The stream data consists of watershed areas, water streams, and labels, while the road data consists of a road network.

A digitized map** of the stream data, along with a description of symbols used in the map to represent various entities, is shown in Figure 1. The stream data comes from the region labeled N3 in Figure 1. Region N3 is a watershed area.

The road data used is a subset of the road data for the entire

---

* This data was obtained from the Dept. of Fisheries and Wildlife Science, Virginia Tech, Blacksburg, VA, courtesy of Dr. Robert Giles.
** 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.

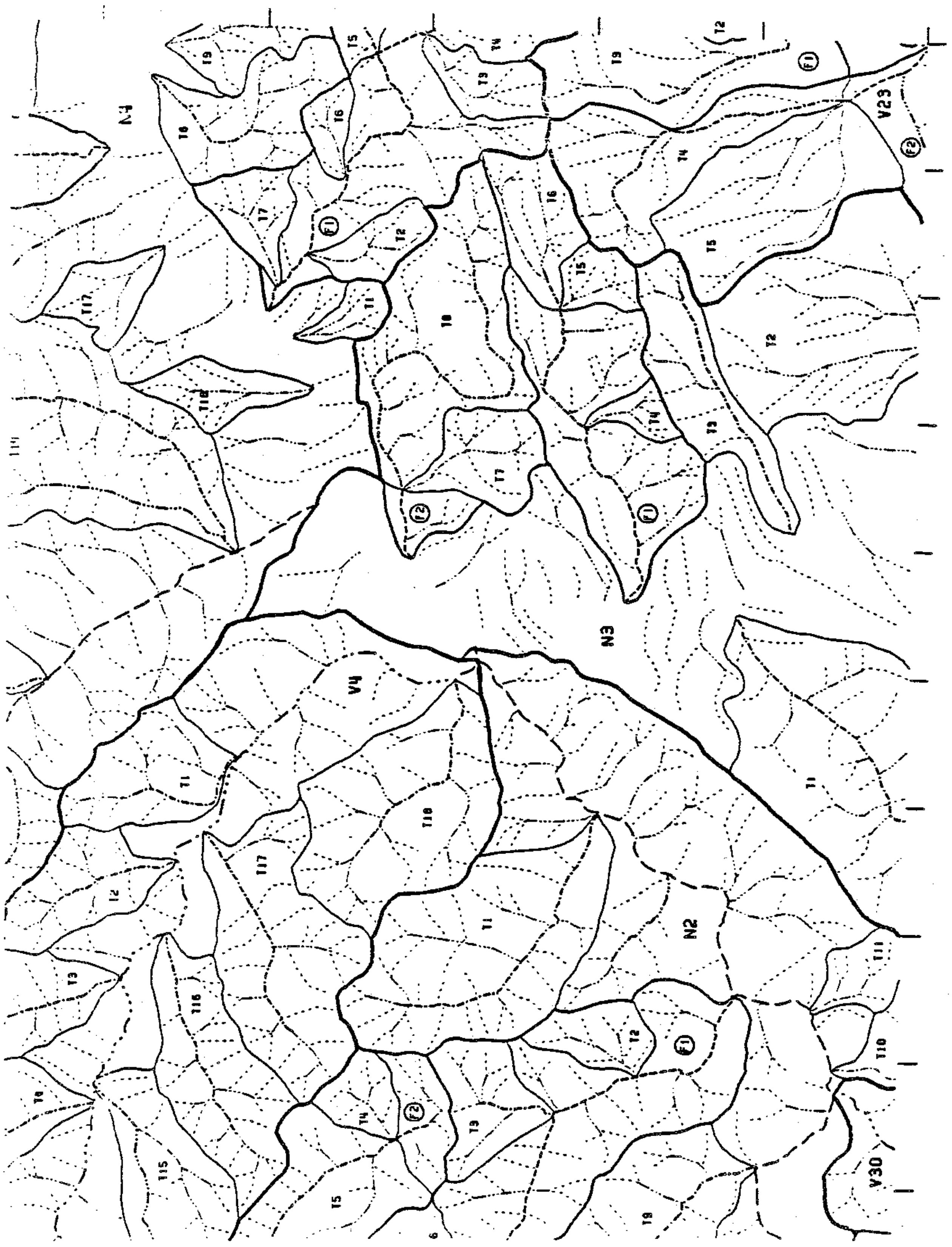


Figure 1 shows a digitized map of the stream data used in our system.

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

From the description of the data, it can be observed that the basic geographic entities used in the system are regions, water streams, roads, and labels. A region 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. 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.

Figure 2 illustrates the prototypes REGION, WATER STREAMS, STREAM, LABEL, POLYGON, and CHAIN. 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. 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 immed-

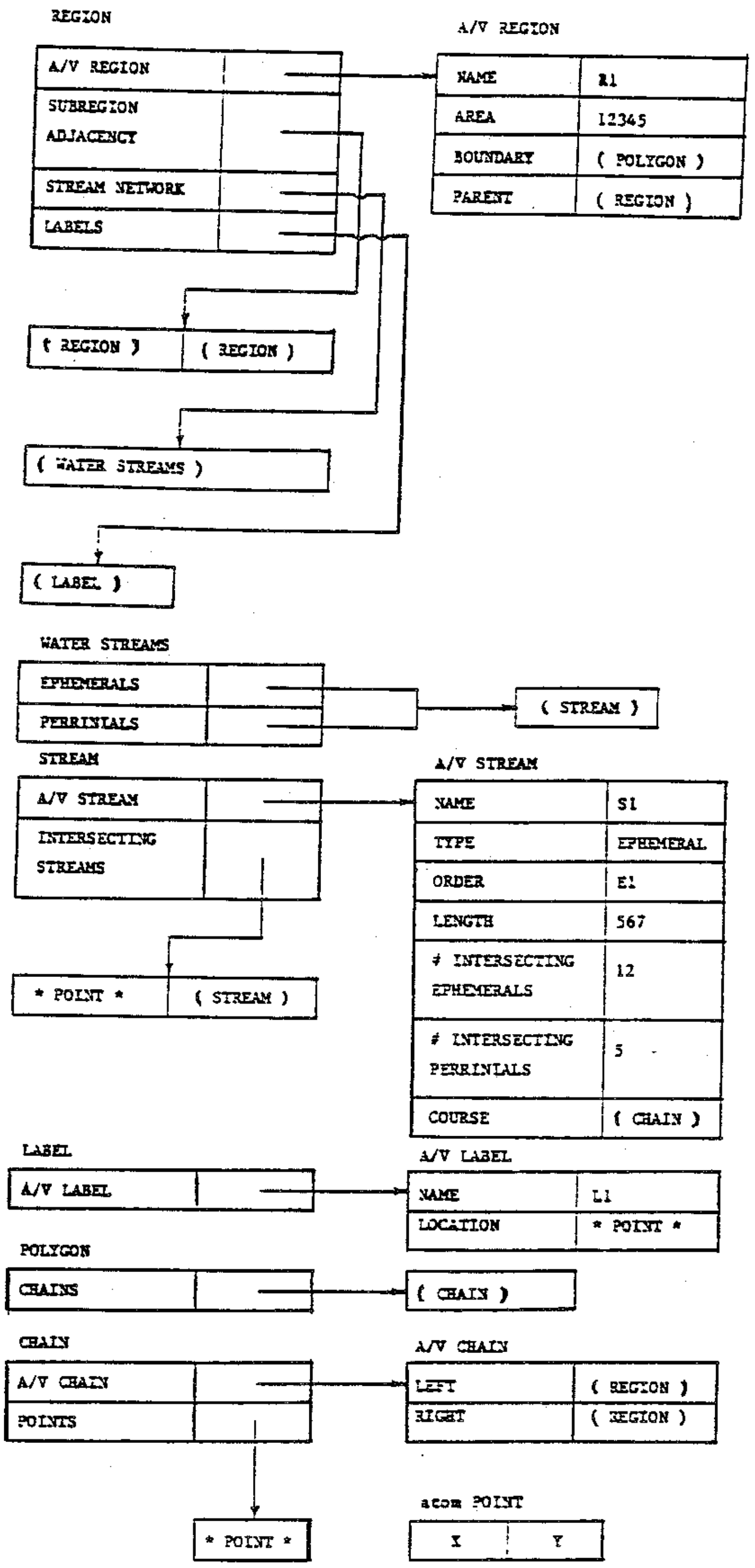


Figure 2 illustrates the prototypes for spatial data structures REGION, WATER STREAMS, STREAM, LABEL, POLYGON, and CHAIN.

iate region which encloses the region under consideration.

A region may have to be divided into subregions, in which case the subregions are stored in a SUBREGION ADJACENCY relation. This 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 structures of type REGION.

The relations of type STREAM NETWORK are unary relations whose components are spatial data structures of type WATER STREAMS. The relations of type LABELS are unary relations whose components are spatial data structures of type LABEL. There are two types of streams: ephemerals and perrinials. WATER STREAMS therefore consists of two relations: EPHEMERALS and PERRINIALS. Both EPHEMERALS and PERRINIALS are unary relations whose components are spatial data structures of type STREAM.

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. The A/V STREAM 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 representing the course of the stream.

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

Each spatial data structure of type LABEL consists of only one relation, an A/V relation called A/V LABEL. 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. The labels in our experimental system would be replaced by other point data such as cities in a real system.

The low level spatial data structure types include the POLYGON and CHAIN and also the atom POINT. 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 POLYGON has a unary relation called CHAINS, whose components are spatial data structures of type CHAIN.

We represent the course of any water stream or road by a spatial data structure which is of type CHAIN. 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 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.

The relation POINTS is an ordered list (a binary relation) of points that define the chain. A POINT is an atom, a data element at the innermost level which can not be further broken down. A POINT consists 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.

III. PHYSICAL DESIGN

The physical design consists of three parts: the data structures used in internal memory, the file structures used in external storage, and the memory management system that interfaces between the two.

III.1 Internal Memory Data Structures

In internal memory, spatial data structures and relations are linked structures implemented in PASCAL. Each spatial data structure (SDS) and each relation has a unique name. An SDS can be accessed by name through the SDS-Dictionary, and a relation through the REL-Dictionary. The dictionaries (temporarily implemented as ordered-lists in Version 1) will be implemented as height-balanced search trees [14] in our next version. Looking up an SDS or relation name in the appropriate dictionary returns a pointer to the header of the structure. Both SDS's and relations use a structure called an RDS (relational data structure header) for their headers.

The RDS is a 6-tuple consisting of NAME (the unique character string name of the structure), TYPE (Ø for SDS headers and 1 or 2 for relation headers), LENGTH (number of relations for SDS headers; number of tuples for relation headers), USE_CNT (number of structures that reference this one), and STRUCT (pointer to the rest of the structure). In an SDS header, STRUCT points to a list of relation cells, and each of these points to the header of a relation. In a relation header, STRUCT points to the first tuple of the relation.

There are two types of relations: TREE relations (type 1) and LIST relations (type 2), depending on whether the N-tuples are stored in a TREE or a LIST structure. For a TREE structure, the tree is N levels deep, corresponding to the dimension 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 structured so that all the N-tuples with the same first component share the same TREE_CELL on the first level. 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. To facilitate searching the tree, and hence the relation, the N-tuples are stored in a lexigraphical order.

In a LIST relation there are two lists: a list of N-tuples, which grows vertically, and a list of components of an N-tuple, which grows horizontally. Figure 4 shows how a relation consisting of N-tuples (a, b, c), (a, b, d), and (b, c, d) is represented as a LIST relation.

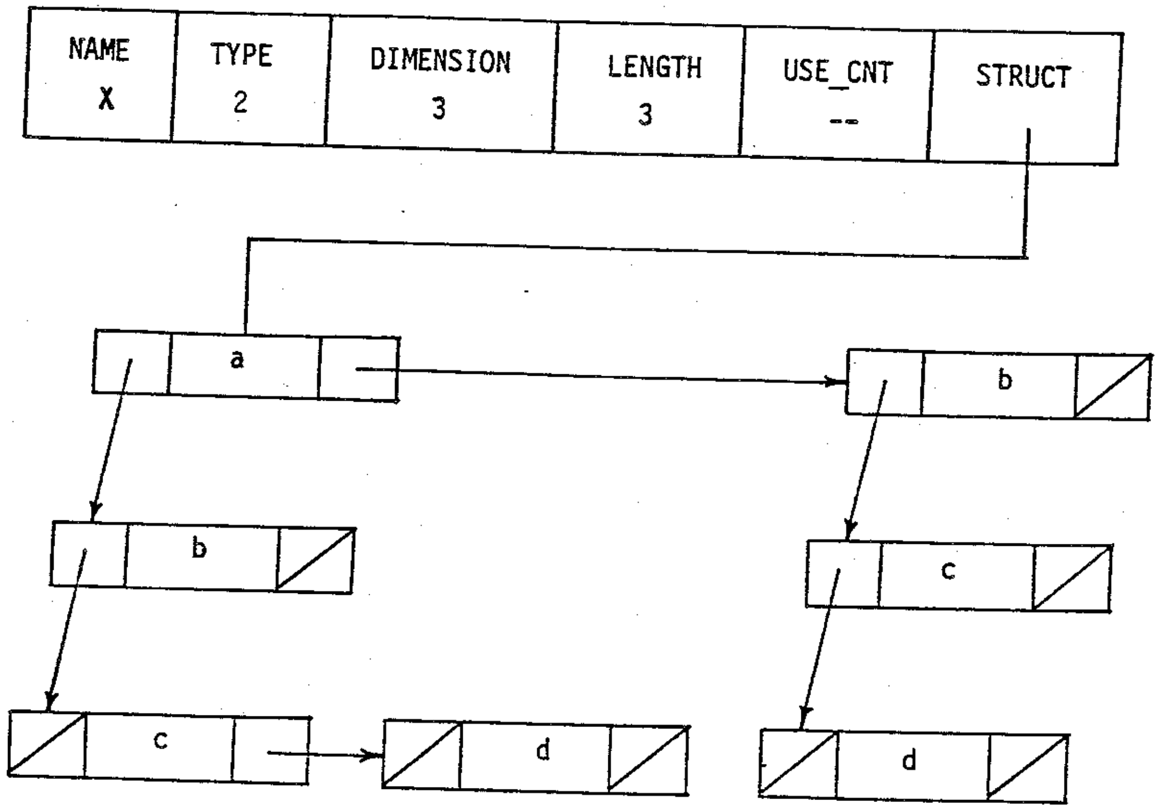


Figure 3 illustrates the structure of a TREE-structured relation with tuples (a, b, c), (a, b, d), and (b, c, d).

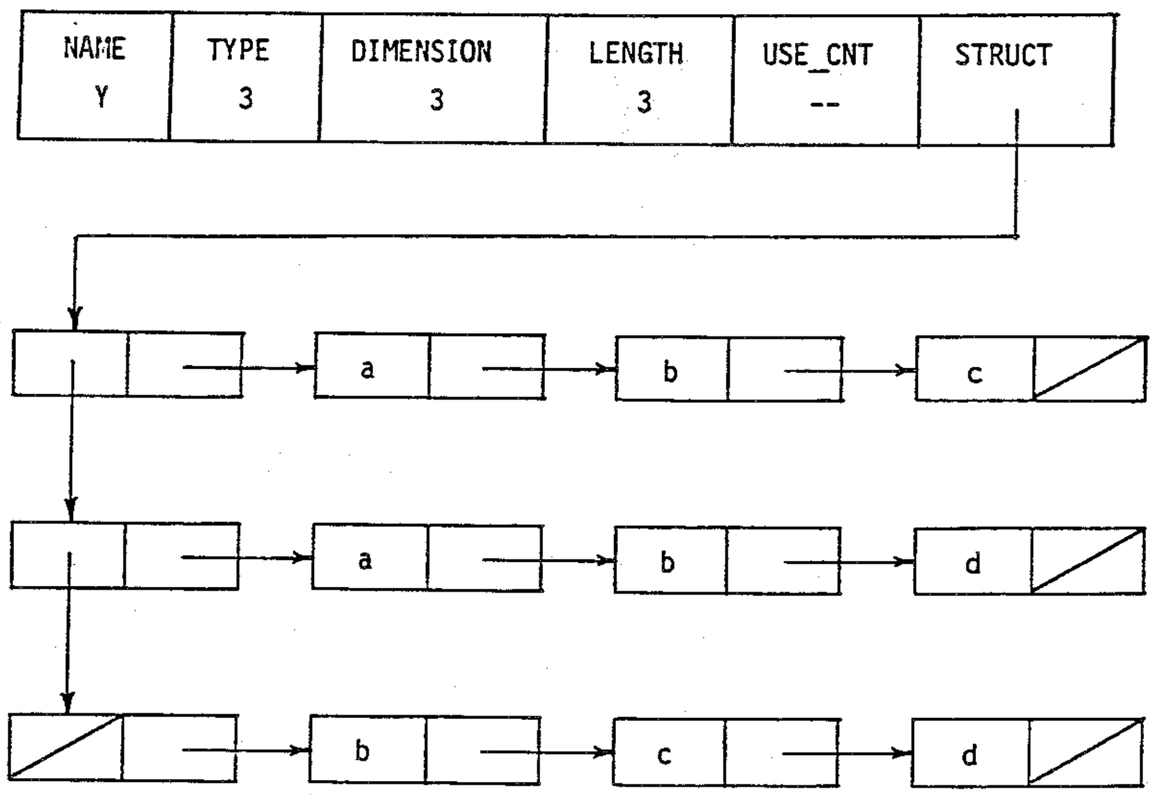


Figure 4 illustrates the structure of a LIST-structured relation with tuples (a, b, c), (a, b, d), and (b, c, d).

The order of the N-tuples stored in a LIST relation is user defined, and N-tuples may be added at the beginning of the list, at the end of the list, or at a user-specified position in the list. 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.

Figure 5 illustrates the physical structure of the spatial data structure for a region and its A/V relation in our experimental system.

III.2 External Storage Structures

The physical structures defined in the preceding section are suitable for internal manipulations. However, it is not feasible to set up these physical structures in memory every time the database is loaded. A real spatial information system would be much too large to load the entire database into memory at one time. Our experimental system, therefore, resides in secondary storage and parts of it are retrieved as necessary.

The main objective of Version 1 was to bring up a system that used the spatial data structure as a building block and stored real geographic data obtained by low-level processing of multispectral remotely sensed imagery or by digitizing maps. Therefore, the physical structure of the database in secondary storage was kept as simple as possible; the issues of optimal record and file organization have been ignored. Instead, the VAX/VMS file system was used as much as possible.

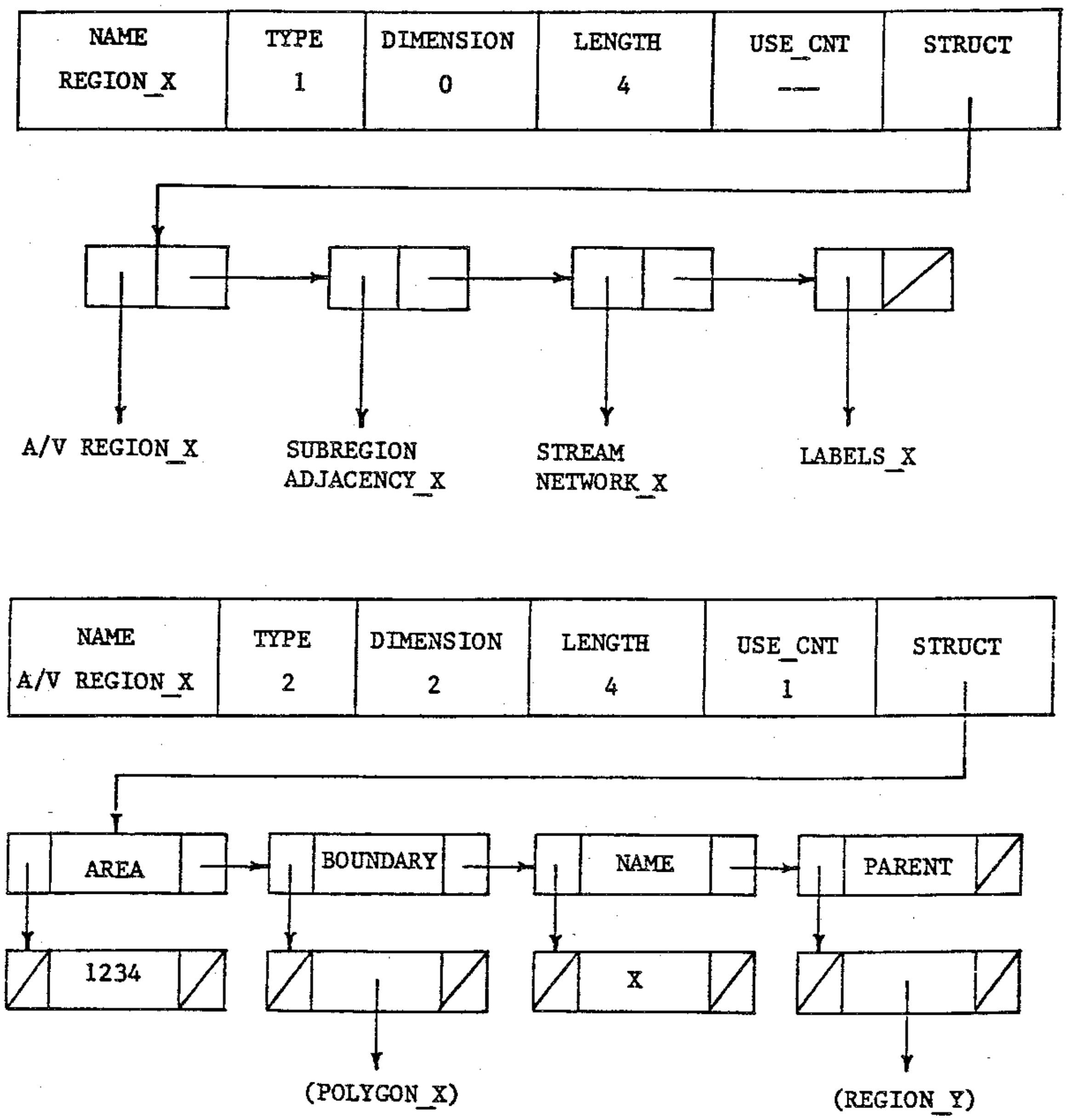


Figure 5 illustrates a spatial data structure for a REGION and its attribute-value relation.

Each spatial data structure and each relation in the database is stored in a separate file on the disk. The unique character string names of the structures are used as file names. This makes it very easy to fetch an entire SDS or relation into internal memory. The data is stored sequentially within a file, and no indexing is used. Thus there is no efficient way to access a part of an SDS or relation in external storage. This is reasonable since Version 1 only accesses external storage in order to read an entire structure into internal memory or write one back to secondary storage. The advantages of using the sequential file organization are 1) simplicity and 2) the ability to store the data in that order which best facilitates the formation of the internal physical structures.

III.3 Memory Management

The memory management method used by our experimental system is similar to the scheme known as "segmentation" when used by operating systems. The MULTICS operating system uses this type of memory management.

The experimental system keeps two tables SDS_IN_CORE and REL_IN_CORE which are circular lists of the spatial data structures and relations, respectively, that currently reside in internal storage. The least and most recently used spatial data structures in these lists are known. In addition, the SDS-Dictionary and the REL-Dictionary store a tag in each element, indicating if the structure is currently in internal memory or not. The variable SPACE keeps track of the amount of space currently in use in internal memory.

When the user references a relation whose tag indicates it is not in memory, the first five records of the file for that relation are read from the disk, and the header is created in internal memory. The variable SPACE is incremented to account for the header, and the REL_DICTIONARY entry is updated to indicate "in-core" and to point to the new header. Now if the user tries to access any tuple in the relation, the relation is entered in the REL_IN_CORE table and made the most recently used relation in core. The tuples are read one by one from the file and inserted into the linked struture being constructed. If a component of any tuple is a spatial data structure and is not present in internal memory, a header for that spatial data structure is created, and a pointer to the header is inserted in the tuple. In any case the USE_CNT of the spatial data structure is incremented to reflect the new pointer to it.

When the user references any spatial data structure that is not in memory, its header is created in the same way as relation headers are created. If the user wants to access any information in the SDS, the remainder of the structure (the relation list) is transferred to internal memory. At this time, it is added to the table SDS_IN_CORE and made the most recently used spatial data structure. As the relation list is read in, the headers of those relations not yet in internal memory are constructed. The USE_CNT fields of all the relations are incremented.

If sufficient space is not available in internal memory for newly created structures, then other structures are transferred back to the disk (only if they have changed since they were read) and deleted from internal storage. When a relation is deleted from internal storage, its tuples are deleted one by one. The SPACE variable is decremented, and the USE_CNT fields of any spatial data structures referenced by such a tuple are decremented. If its USE_CNT becomes zero and its entire structure is not in internal memory, the SDS is also deleted. After the tuples are all deleted, the relation header is deleted if its USE_CNT is less than one. When a spatial data structure is deleted from internal memory, its relation list is traversed, and the USE_CNT's of the relation headers are decremented. If they become zero and the relations are not fully in internal memory, the relations are also deleted. Then the SDS header can be deleted and its space recovered if its own USE_CNT is now less than one. When the system needs more space, it will take turns trying to remove the least recently used relation and least recently used SDS from internal storage. It is interesting to observe that in the experiments we have done so far, our memory management scheme always performs better than the VAX/VMS virtual memory management scheme. One such test ran five times faster with our memory management scheme. The extra time had been due to page faults.

IV. THE QUERY LANGUAGE INTERPRETER

Most relational database systems include a relational query language through which a user can interact with the database. For example, in the SEQUEL language (Chamberlin and Boyce, 1974 [3]), a user might specify the command

```
SELECT    SUPPLIER_NUMBER, STATUS
FROM      SUPPLIER_RELATION
WHERE     CITY = `BLACKSBURG`
AND       STATUS > 2Ø.
```

This command instructs the system to go to the relation whose name is SUPPLIER_RELATION, find those tuples having 'BLACKSBURG' in the CITY component and a number > 2Ø in the STATUS component, and construct a new relation consisting of the SUPPLIER_NUMBER and STATUS components of the selected tuples.

Because our system is entity-oriented rather than relation-oriented, languages like SEQUEL are only indirectly applicable. We envision, in the future, an intelligent system with natural language query facilities so that a user might make the request:

```
FIND  ALL RIVERS
      WITHIN 2ØØ MILES OF ROANOKE,
      LONGER THAN 5Ø MILES, AND
      CROSSED BY INTERSTATE 81.
```

The intelligent system, using knowledge of the prototypes of the spatial data structures and the semantics of the various tuple components, would invoke a deduction system that would determine the best sequence of relational operations required to extract this information from the database. Part of our work involves the design of this intelligent component.

IV.1 Low-Level Operations

The current experimental database system uses a stack-oriented query language similar to the FORTH language (Moore, 1974 [19]). Users can define constants, variables, and arrays and perform arithmetic operations or database operations using control structures. New commands can be defined via a simple macro-definition facility. The system itself is viewed as a calculator with a large stack, which operates with numbers, relations, or spatial data structures. The arguments of all operations are performed on the top entry or top n entries of the stack, where n depends on the particular operation. Results are returned to the stack. To give the reader a feel for the power of the query language, we will briefly describe the primitive operations and give an example of their use.

The query language interpreter supports the high-level datatypes _integer_, _boolean_, _character-string_, _point_, _spatial data structure_, and _relation_. Standard arithmetic and relational operations are provided. The type of the top stack element (TOS) can be determined by test commands. The stack elements can be manipulated, regardless of content, by such operations as SWAP (exchange TOS with next element in stack) DROP (remove TOS), DUP (duplicate TOS), ROT (rotate top three stack elements), -ROT (reverse rotate top three stack elements), and other similar operations.

The stack-oriented query language program control primitives include the IF-ELSE-THEN construct for conditional execution and the DO-+LOOP construct and REPEAT-UNTIL construct for iteration. The vocabulary manipulation primitives include the DEFINE-END_DEF construct for defining new commands, the CONSTANT and VARIABLE commands for defining new constants and variables, the FETCH and STORE commands for adding to and removing from the stack selected vocabulary entries, and the ALLOCATE and FORGET commands for requesting and freeing vocabulary space.

The database manipulation primitives ALLOC_RDS and ALLOC_REL allow the user to allocate and catalog new spatial data structures and relations. The command FIND allows the user to locate structures by alphanumeric name. The commands LIST_RDS, LIST_REL, XLIST_RDS, and XLIST_REL allow the headers and contents of spatial data structures and relations to be listed.

The primitives NT_ATTACH and REL_ATTACH allow the user to add N-tuples of data from the stack to a relation and add a relation to a spatial data structure, respectively. The command INRELA? allows the user to check whether a specified N-tuple is in a given relation. The commands [NAME], [TYPE], [DIMEN], [LENGTH], and [USE_CNT] allow the user to request that certain attributes of a spatial data structure or relation be returned to the stack. The command [STRUCT] returns to the stack a pointer to the beginning of a structure, and the command [LINK] allows the user to advance the pointer through the structure, while the command [DATA] returns the data from the record pointed to.

The user can also request the loading and unloading (DB_LOAD and DB_UNLOAD) of the database, input and output from files (INPUT> and

>OUTPUT), or print a portion of the vocabulary (DUMP). The session is terminated by the command DONE.

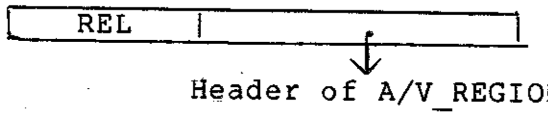
A short example will illustrate the use of the query language. The example session creates a new binary relation A/V_Region_X, inserts N-tuples into it, and attaches that relation to an existing spatial data structure Region_X using the query language commands which are marked by asterisks here.

Input by the User	Explanation and Action Taken
2	The dimension of the relation to be created is going to be the integer 2.
2	The type of the relation is going to be the integer 2.
" A/V_REGION_X"	The name of the relation is going to be A/V_REGION_X.

CHAR	A/V_REGION_X
INT	2
INT	2

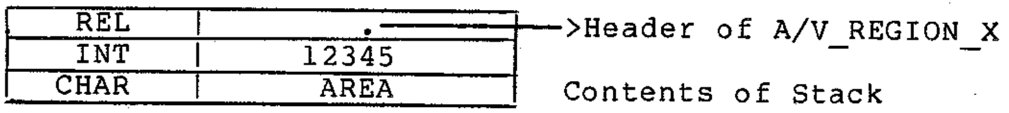
Contents of Stack

*ALLOC_REL	Allocates a relation header with the name A/V_REGION_X, dimension two, and type two, and puts a pointer to it on the top of the stack.



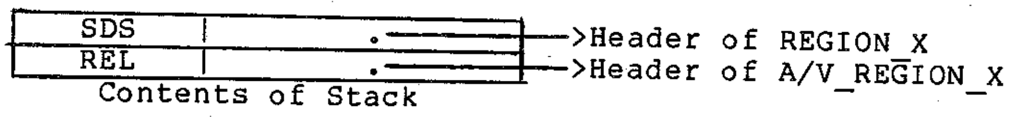
Contents of Stack

" AREA"	An attribute to be part of the A/V relation.
12345	The value of the attribute.
*ROT	Bring the pointer to the relation to the top of the stack.



Contents of Stack

*NT_ATTACH	Attach the N-tuple consisting of attribute and value to the relation. The stack becomes empty.
" A/V_REGION_X" *FIND *DROP	Restore pointer to the relation A/V_REGION_X to the top of the stack, and drop status indicator.
" REGION_X"	Name of the spatial data structure to which the relation is to be attached.
*FIND	Search the dictionary for the given spatial data structure name and put a pointer to it on the stack.
*DROP	Drop the top of the stack which indicated the status of the search.



*REL_ATTACH	Attach the relation to the spatial data structure.

IV.2 Higher Level Operations

The Query Language Interpreter currently recognizes commands that perform simple, low-level operations on spatial data structures, relations, system dictionaries, and the stack. In order to understand the architectural needs of the system, we need to look at these and the higher-level operations that must be performed to answer a query. Consider the following list of sample queries, which can all be answered using the information in the current experimental system.

Q1) How many rivers (perrineal streams) are there in region X?

Q2) How many streams of any kind are in region X?

Q3) What are the names of the three longest rivers (perrineal streams) in region X?

Q4) What is the length of river Y?

Q5) What cities (labels) are in region X?

Q6) What cities (labels) are within distance D of stream Y in region X?

Q7) What cities (labels) are within distance D of every stream in region X?

Q8) What regions does stream X flow through?

Q9) What regions does road X pas through?

Q1Ø) What regions are adjacent to region X?

Q11) What points do stream X and stream Y have in common?

Some of these queries involve quick look-up operations, and others involve more complex searching. We can group the operations required to process these sample queries into three kinds: 1) low-level access and manipulation functions, 2) high-level relational operators, and 3) geometric or distance operations.

The low-level functions include

1) accessing a SDS from its name,
2) accessing a named relation of a given SDS,
3) creating new relations or SDS's,
4) creating tuples,
5) adding tuples to relations or relations to SDS's,
6) accessing the name, type, dimension, or length of a SDS or relation, and
7) accessing the value of a given attribute for a given SDS.

These are all already provided by the Query Language Interpreter.

The higher-level relational operations involved in answering these sample queries include

1) extracting information from SDS's referenced by each tuple of a given relation,
2) selecting tuples of a relation that satisfy a dynamically changing constraint,
3) joining pairs of tuples from two relations if the pair satisfies a contraint,
4) projecting a relation onto certain columns, and
5) selecting tuples of a relation that satisfy a constraint with respect to every tuple of a second relation.

These operations suggest that generalized forms of the now standard relational database operators will be useful in a spatial information system. Let R be an N-ary relation and S be an M-ary relation. Let P be an N-ary predicate and Q be an (N+M)-ary predicate. Let I be the set of positive integers and f be a binary relation over I. We propose the following general forms.

<u>Projection</u>

$$\mathrm{PROJ}(R;f) = \{(c_1,\ldots,c_K) \mid \text{for some } (a_1,\ldots,a_N) \in R,\ (i,j) \in f \text{ implies } a_i = c_j\}.$$

<u>Selection</u>

$$\mathrm{SEL}(R;P) = \{(a_1,\ldots,a_N) \in R \mid P(a_1,\ldots,a_N) = \text{true}\}.$$

Join

$$\mathrm{JOIN}(R,S;Q) = \{(a_1,\ldots,a_N,b_1,\ldots,b_M) \mid (a_1,\ldots,a_N) \in R, (b_1,\ldots,b_M) \in S, \text{ and } Q(a_1,\ldots,a_N,b_1,\ldots,b_M) = \text{true}\}.$$

Division

$$\mathrm{DIV}(R,S;Q) = \{(a_1,\ldots,a_N) \in R \mid \text{for every } (b_1,\ldots,b_M) \in S,\ Q(a_1,\ldots,a_N,b_1,\ldots,b_M) = \text{true}\}.$$

Finally the geometric or distance functions required for the sample queries are

1) intersection of chain with chain,
2) intersection of chain with polygon, and
3) distance from a point to a chain.

Of course a larger special purpose system will have its own set of geometric functions as required by the users of the system.

V. ARCHITECTURAL IMPLICATIONS

V.1 General Implications

The architectural implications of the general spatial data structure and the operations we want the architecture to perform on such structures are influenced by two facts:

1) the use of an arbitrary predicate in the select, join, and division operations, and

2) the assumption that many tasks are going to require sequencing through all the tuples of a relation to obtain an answer.

To some extent, fact (1) implies fact (2) since the use of an arbitrary predicate means that the usual database mass storage organizations with dictionaries, hash tables, or inverted files may not be adequate. Because the general operation takes the form: find all tuples satisfying a given condition, where the condition may be a constant one or where the condition may depend on the tuples currently being examined in a related relation, there can be a natural parallelism. Divide the tuples of the relation into mutually exclusive subsets and have one CPU responsible for processing the tuples in the subset assigned to it. After this kind of processing collect the results together and output them when appropriate.

Another consequence of the use of an arbitrary predicate is that on the average, the time taken to process a tuple can easily be greater than the time taken by a smart controller with memory to retrieve it from the mass storage device. Thus, the general flow of control is to access the mass storage device in an anticipatory manner, and store these read ahead tuples in the memory of the input CPU. This input CPU with its memory acts like a queue, always making sure that it has the tuples which are going to be requested. Its smart algorithm for doing the input in the anticipatory manner based on current disk head and disk pack positions just reduces the average time to retrieve a block of data.

Ordinarily, we think of the tuple as the logical entity to be read and manipulated. Thus, blocks on the mass storage device are sets of tuples. However, this is not an efficent way to retrieve data when not all components of the tuple are required for processing. We suggest a data organization in which each block of data contains one component from a set of tuples. In this manner, with a random file organization, only the components of tuples required for an operation need be accessed.

Connected to the input CPU is an input selection CPU to which are connected the number crunching CPU's. These in turn are connected to an output selection CPU which is connected to an output CPU. The output CPU acts like an output queue to the mass storage devise. Figure 6 illustrates a block diagram of the architecture in which the mass storage device can be an entire disk. For faster processing, this architecture can be replicated, one replication per disk head. In the remainder of this section, we briefly sketch the multiprocessor algorithms for executing the projection, selection, join, and division operations on the architecture of Figure 6.

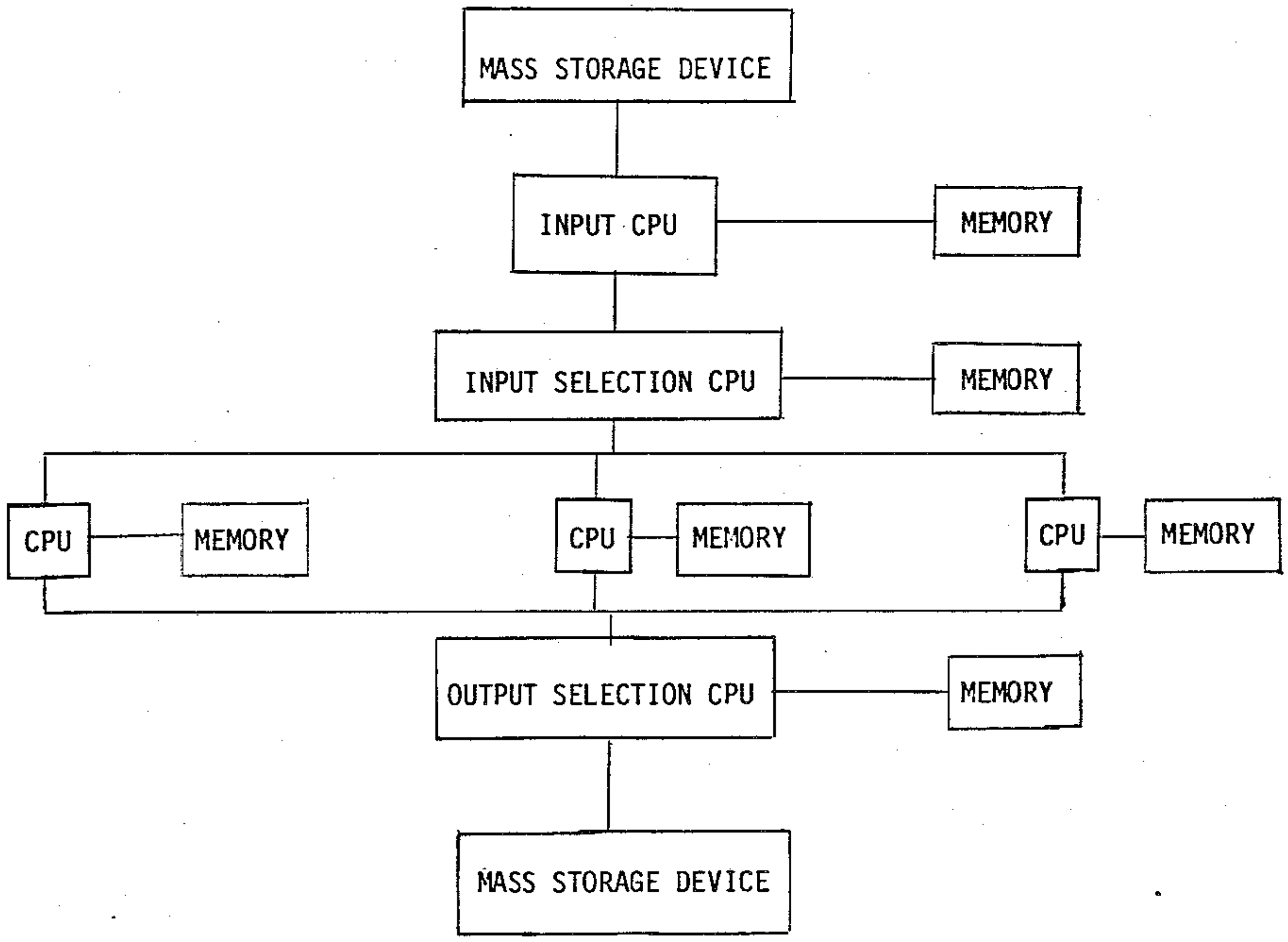


Figure 6 illustrates a block diagram of the multiprocessor.

V.2 Projection

Projection is conceptually simple. Sequentially go through all tuples accessing only those components desired. Then throw away all duplicate tuples which might have been generated.

The difficulty in the multiprocessor version of the algorithm is with the duplicates. To keep the number crunching CPU's from communicating with each other, the algorithm has to guarantee that two projected tuples which are identical get processed by the same number crunching CPU. One way of handling this is for the input CPU to hand off a partially projected tuple to a selection CPU whose job is to do a simple lexicographic or hash calculation to decide which number crunching CPU will handle the task. Each number crunching CPU checks to see if the tuple it gets has already been seen. If so it ignores the tuple. If not, it adds the tuple to its table of tuples seen and hands the tuple to the output selection CPU which hands it to the output CPU to be stored on the mass storage device.

Problems arise if not all projected tuples can be stored in the memories of the number crunching CPU's. In this case, the selection CPU must not select all tuples to be given to the number crunching CPU's. Those not selected are selected in subsequent rereads of the relation.

V.3 Selection

Selection works in a similar way to projection. Tuples are read by the input CPU. Each tuple is handed to the selection CPU who decides which number crunching CPU will process it. Upon receiving a tuple, the number crunching CPU evaluates the predicate. If true, the tuple is sent to the output selection CPU which simply hands it to the output CPU for storage to an output file, and the next tuple is requested.

V.4 Join

To accomplish the join, every tuple of one relation has to be paired or concatenated with every tuple of the second relation. The concatenated tuple then has to be evaluated by the joining predicate. If the predicate is true the concatenated tuple is written out.

To increase the efficiency of this task, the join predicate can be analyzed ahead of time to determine what simple predicate on the tuples from the first relation must be true whenever the join predicate is true on the concatenated tuple and what simple predicate on the tuples from the second relation must be true whenever the join predicate is true on the concatenated tuple. These simple predicates can be used by the input selction CPU to ignore tuples having no chance of being joined.

To minimize the number of times the relation files have to be read, each number crunching CPU has to store in memory as many tuples passing the selection test from the smaller relation as it can. Of course, different number crunching CPU's store mutually exclusive groups of tuples. Then the large relation is read. Tuples which pass their selection test are handed off to any available number crunching CPU.

If not all the tuples passing the selection test from the smaller relation can be collectively held in the memory of the number crunching CPU's, repeated passes over the relations must be made.

V.5 Division

Division works like join except, of course, that for a tuple to be output its concatenation with every tuple from the second relation must evaluate to true. To execute division, the number crunching CPU's load in as many tuples from the second relation as possible. Then the first relation must be read tuple by tuple. Each tuple which passes the selection test is handed off to all number crunching CPU's simultaneously. These CPU's evaluate the division predicate of the tuple concatenated with all tuples it has from the second relation. If the predicate evaluates true, the tuple is handed off to the output selection CPU whose job is to determine if all number crunching CPU's indicate that the tuple has passed all predicate evaluations. If not the output selection CPU ignores the tuple. If all number crunching CPU's indicate the tuple has passed, then it sends the tuple to the output CPU to be put as the output file on the mass storage device.

VI. CONCLUSIONS AND FUTURE WORK

We have described an experimental spatial information system whose building block is the general spatial data structure. This system demonstrates the feasibility of using such structures to store spatial information. The query language interpreter provides a simple, but powerful interface for researchers working with the experimental system. The low-level operations supported by the interpreter plus the high-level relational operations defined in this paper and a set of suitable geometric operations should provide all the support functions necessary for answering high-level user queries. An intelligent front end which can translate a user query into a sequence or even a program of such operations is needed to complete the design.

The system designed here allows queries involving arbitrary, possibly dynamically changing predicates. The architectural implications of this use of arbitrary predicates suggest that traditional ordering schemes or secondary indices will not be useful in the kind of system we envision. We have suggested storing relations by column instead of by tuple and use of parallel processors to help speed up the processing of these generalized queries. Future work includes the implementation of the high-level operations on our present system, implementation of a second version using only secondary storage for SDS's and relations, designing a specific architecture for spatial information systems, and designing the intelligent front end processor.

REFERENCES

1. Barrow, H. G. and J. M Tenenbaum, "Recovering Intricsic Scene Characteristics from Images", in Computer Vision Systems, A. Hanson and E. Riseman, Eds., Academic Press, New York, 1978.

2. 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 IFIP Congress 1974, North Holand Publishing Company, Amsterdam, 1974.

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

4. Chang, N. S. and K. S. Fu, "Query-By-Pictorial-Example", IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 198Ø.

5. Chang, S. K., B. S. Lin, and R. Walser, "A Generalized Zooming Technique for Pictorial Database Systems", National Computer Conference, 1979, p. 147-156.

6. Chang, S. K., J. Reuss, and B. H. McCormick, "Design Considerations of A Pictorial Database System", Policy Analysis and Information Systems.

7. Codd, E. F., "A Relational Model of Data for Large Shared Databases", Communications of ACM, Vol. 13, No. 6, June 197Ø, pp. 377-389.

8. Date, C. J., An Introducation to Database Systems, 2nd Edition, Addison Wesley, New York, New York, 1977.

9. 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 Structures for Geographic Information Systems, Harvard University, Cambridge, Massachusetts, October 1977.

1Ø. 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.

11. Gold, C., "Triangular Element Data Structures", Users Applications Symposium Proceedings Services, Edmonton, Alberta, Canada, 1976.

12. Hagan, P. J., A Network Data Model for Cartographic Features, Doctor of Science Dissertation, Sever Institute of Technology, Washington University, Saint Louis, Missouri, May 198Ø.

13. Hanson, A. R. and E. M. Riseman, "VISIONS: A Computer System for Interpreting Scenes", in Computer Vision Systems, A. Hanson and E. Riseman, eds., Academic Press, New York, 1978.

14. Horowitz E. and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Inc., Potomac Maryland 20852, 1977.

15. Jensen, K. and Wirth N., PASCAL User Manuaul and Report, Second Edition, Springer-Verlag, New York, 1974.

16. Laboratory for Computer Graphics, POLYURT: A Program to Convert Geographic Base Files, Harvard Universty, Cambridge, Mass. 1974.

17. Marr, D., "Representing Visual Information - A Computational Approach", in Computer Vision Systems, A. Hanson and E. Riseman, eds., Academic Press, New York, 1978.

18. Modeleski, M., Topology for INGRES An Approach to Enhance Graph Property Recognition by GEOQUEL, Geograpphic Database Coordinator, Association of Bay Area Governments, Berkeley, California 94705, 1977.

19. Moore, C. H., "FORTH, A New Way to Program A Computer", Astronony and Astrphysics Supplement, 1974, No. 15, pp. 497-511.

20. 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.

21. Peucher, T. K. and N. Chrisman, "Cartographic Data Structures", The American Cartographer, Vol. 2, No. 1, April 1975, pp. 55-69.

22. Shapiro, L. G. and R. M. Haralick, "A Spatial Data Structure", Geo-Processing, 1, 1980, pp. 313-337.

23. 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.

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

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

26. 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.

27. 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, Massachusetts, 1978.

28. Wiederhold, G., Database Design, McGraw-Hill, New York, 1977.