rithm-structured architecture, which allows for efficient processing (when program and data are mapped properly into the architecture). To the extent that this is true, it may be possible to make MIMD networks and mixed SIMD–MIMD systems similarly efficient.

At least two other architectures should be examined, and more detailed, deeper comparisons devised to compare these radically different serial, parallel and parallel-serial systems. The two architectures are pipelines (e.g., the Cytocomputer [12] and systolic arrays [7]).

A pipeline tends to use processors specialized for a particular type of problem (e.g., a window operation for image processing, floating-point arithmetic for numerical matrices). Hence, the processor can be relatively simple, with a relatively high percent of active gates. Memory traditionally is even smaller than in the arrays, since intermediate results are immediately pumped into the next processor in the pipe. But each processor must have its own controller, albeit a relatively simple one.

Systolic arrays are, basically, two-dimensional pipelines that have very simple special-purpose processors, configured into a system that will execute a particular algorithm. Therefore, the "processor" can be quite small, e.g., 25, 10, or even 5 gates, and the memory tailored to the absolute minimum needed for that algorithm. The program instruction and controller functions are taken over by hard-wiring.

Both pipelines and systolic arrays are more specialized, or even special-purpose. They handle a much smaller set of programs, but they may be more efficient for those they can handle. A set of special-purpose systolic arrays reminds one of a CPU's armory of special-purpose processors on a common bus. But now data might be pumped through sequences of these processors, appropriately configured so that many, rather than one, will be busy at each cycle.

Here we see specific examples of the general phenomenon of the price that must be paid for generality.

Systolic arrays are built exactly to achieve isomorphic mappings of hardware to algorithm.

Serial computers pay a heavy price in random memory access for a single processor, which allows isomorphic mapping via software. Serial computers have a very low percent of active resources because of the serial "Von Neumann bottleneck" [1].

Arrays and networks can be given a more or less general topology, and sometimes good algorithms map well, or even perfectly, but sometimes not.

Networks of traditional processors have all the inefficiencies of serial computers, plus many more of their own, because of the often excessively high overheads from message-passing and poor synchronization.

Networks might be made more efficient by specializing both processors and topology to (sets of) problems, in the spirit of systolic arrays.

Limited reconfiguring might help effect this (at the nonnegligible cost, which must be taken into account, of reconfiguring switches).

Summary and Conclusions

Several radically different types of multicomputer array and network systems can now be built. Each has its advantages and its disadvantages.

The very large SIMD arrays of simple 1-bit processors, each with a small local memory, appear to use the highest percent of active resources and therefore, potentially, to be able to give the greatest, and fastest, throughput.

But arrays tend to be specialized, at least today. Judicious combinations of arrays (as in pyramids or other stacks of arrays) and of arrays with networks, and also limited reconfiguring, might serve well to increase generality while maintaining high utilization.

References

[1] J. Backus, "Can programming be liberated from the Von Neumann style: A functional style and its algebra of programs," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 613–641, 1978.
[2] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, 1968.
[3] K. E. Batcher, "Architecture of a massively parallel processor," in *ACM Proc. 7th Annu. Symp. Comput. Arch.*, 1980, pp. 168–174.
[4] F. Briggs, K. S. Fu, K. Hwang, and J. Patel, "PM4—A reconfigurable multimicroprocessor system for pattern recognition and image processing," in *Proc. AFIPS NCC*, 1979, pp. 255–265.
[5] M. J. B. Duff, "Review of the CLIP image processing system," in *Proc. Nat. Comput. Conf.*, 1978, pp. 1055–1060.
[6] D. J. Kuck, *The Structure of Computers and Computation*, vol. 1. New York: Wiley, 1978.
[7] H. T. Kung, "The structure of parallel algorithms," in *Advances in Computers*, vol. 19, M. C. Yovits, Ed., 1980, pp. 293–326.
[8] J. Lipovski, "On a varistructured array of microprocessors," *IEEE Trans. Comput.*, vol. C-26, pp. 125–138, 1977.
[9] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[10] S. F. Reddaway, "DAP—A flexible number cruncher," in *Proc. 1978 LASL Workshop on Vector and Parallel Processors*, Los Alamos, NM, 1978, pp. 233–234.
[11] H. J. Siegel *et al.*, "PASM: A partitionable multimicrocomputer SIMD/MIMD system for image processing and pattern recognition," School Elec. Eng., Purdue Univ., West Lafayette, IN, TR-EE 79-40, 1979.
[12] S. R. Sternberg, "Cytocomputer real-time pattern recognition," presented at the 8th Pattern Recognition Symp., Nat. Bureau of Standards, Apr. 1978.
[13] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*—A modular, multi-microprocessor," in *Proc. AFIPS NCC*, 1977, pp. 637–663.
[14] L. Uhr, "Network and array architectures for real-time perception," Dep. Comput. Sci. Tech. Rep. 424, Univ. Wisconsin, 1981.
[15] ——, *Computer Arrays and Networks: Algorithm-Structured Parallel Architectures*. New York: Academic, 1982.
[16] L. D. Wittie, "MICRONET: A reconfigurable microcomputer network for distributed systems research," *Simulation*, vol. 31, pp. 145–153, 1978.

Design and Architectural Implications of a Spatial Information System

PRASHANT D. VAIDYA, LINDA G. SHAPIRO, ROBERT M. HARALICK, AND GARY J. MINDEN

***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 [17]), 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.**

***Index Terms*—Geographic information system, relational database system, spatial 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

Manuscript received January 4, 1982.

P. D. Vaidya, L. G. Shapiro, and R. M. Haralick are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.
G. J. Minden is with the Department of Electrical Engineering, University of Kansas, Lawrence, KS 66045.

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 [17]), 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 we 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 we 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.

A. Geographic Systems

Of the geographic spatial information systems, the Canadian Geographic Information System (CGIS) [18], [19] 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) [20] 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 POLYVRT system [12] is similar to the DIME system described above except that the polygon is established as a separate entity linked to the chains which compose it. This allows easy maintenance and manipulation of the chains.

GEOGRAF is a system proposed by Peucher and Chrisman [16] to handle both planar data and surface data. To handle the 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 [22] 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 refinements 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.

IBM's Geo-Data Analysis and Display System (GADS) [2] is the first documented geographic database system which used the approach of the relational database (Codd [6]). 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.

The GEO-QUEL system developed at the University of California, Berkeley, is another system which uses the relational database approach to manipulate geographic data [7]. 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 is used to interrogate the system. Modeleski [15] proposed adding relational attributes to GEO-QUEL to permit topological manipulation of geographic files. Hagan [8] 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.

B. Computer Vision Systems

Chang *et al.* [5] 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 [4] was implemented to allow for flexible hierarchic information retrieval. Chang and Fu [3] designed a relational pictorial database system where access is through a high-level relational query language called "Query-by-Pictorial-Example."

Hanson and Riseman [9] 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 endpoints. Levine and Shaheen [13] use relational databases to store raw data, current interpretations, and detailed scene models in a modular computer vision system.

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 [14] and Barrow and Tenenbaum [1] for important discussions of midlevel information.) The structure that we have proposed represents a unified approach to storing such information in a universal structure.

II. Logical Design

A. 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 $D = \{R1, \cdots, RK\}$ of relations. Each relation Rk has a dimension Nk and a sequence of domain sets $S(1, k), \cdots, S(Nk, k)$. That is, for each $k = 1, \cdots, K$, $Rk \subseteq S(1, k) X \cdots X S(Nk, k)$. 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 that: 1) the spatial data structure is defined with a recursive definition, 2) it will often be possible to describe operations on the structure by simple recursive algorithms, and 3) 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.

The above approach to database design is called an *entity-oriented approach.* Most commercially available relational database systems are relation-oriented rather than entity-oriented. In order to investigate the entity-oriented approach, we implemented our own database system.

B. The Logical Database Structure

Our first use of the system is with geographic data. The data[1] used in this system are of two types: stream data and road data. The stream data consist of watershed areas, water streams, and labels, while the road data consist of a road network.

A digitized map[2] of the stream data, along with a description of symbols used in the map to represent various entities, is shown in Fig. 1. The stream data come from the region labeled *N*3 in Fig. 1. Region *N*3 is a watershed area. The road data used are a subset of the road data for the entire Appalachia quadrangle which includes region *N*3. 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.

Fig. 2 illustrates the prototypes REGION, WATER STREAMS, STREAM, LABEL, POLYGON, and CHAIN. Each spatial data structure of type REGION consists of four relations: 1) the *A/V* relation, *A/V* REGION; 2) SUBREGION ADJACENCY; 3) STREAM NETWORK; and 4) 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 immediate 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 of 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 component 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 a region to its right. The *A/V* relation therefore has two attributes: LEFT and RIGHT. The values of both of 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 cannot be further broken down. A POINT consists of an ordered pair (X, Y) where X represents the latitude or the X coordinate and Y represents the longitude or the Y coordinate.

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. We will briefly describe the internal and external structures. The memory management system uses the concept of segmentation and is described in [21].

A. 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 [10] in our next version. Looking up an SDS or relation name in the appropriate dictionary returns a pointer to the header of the structure.

There are two types of relations: TREE relations and LIST relations.

[1] These data were obtained from the Department of Fisheries and Wildlife Science, Virginia Tech, Blacksburg, VA, courtesy of Dr. Robert Giles.

[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.

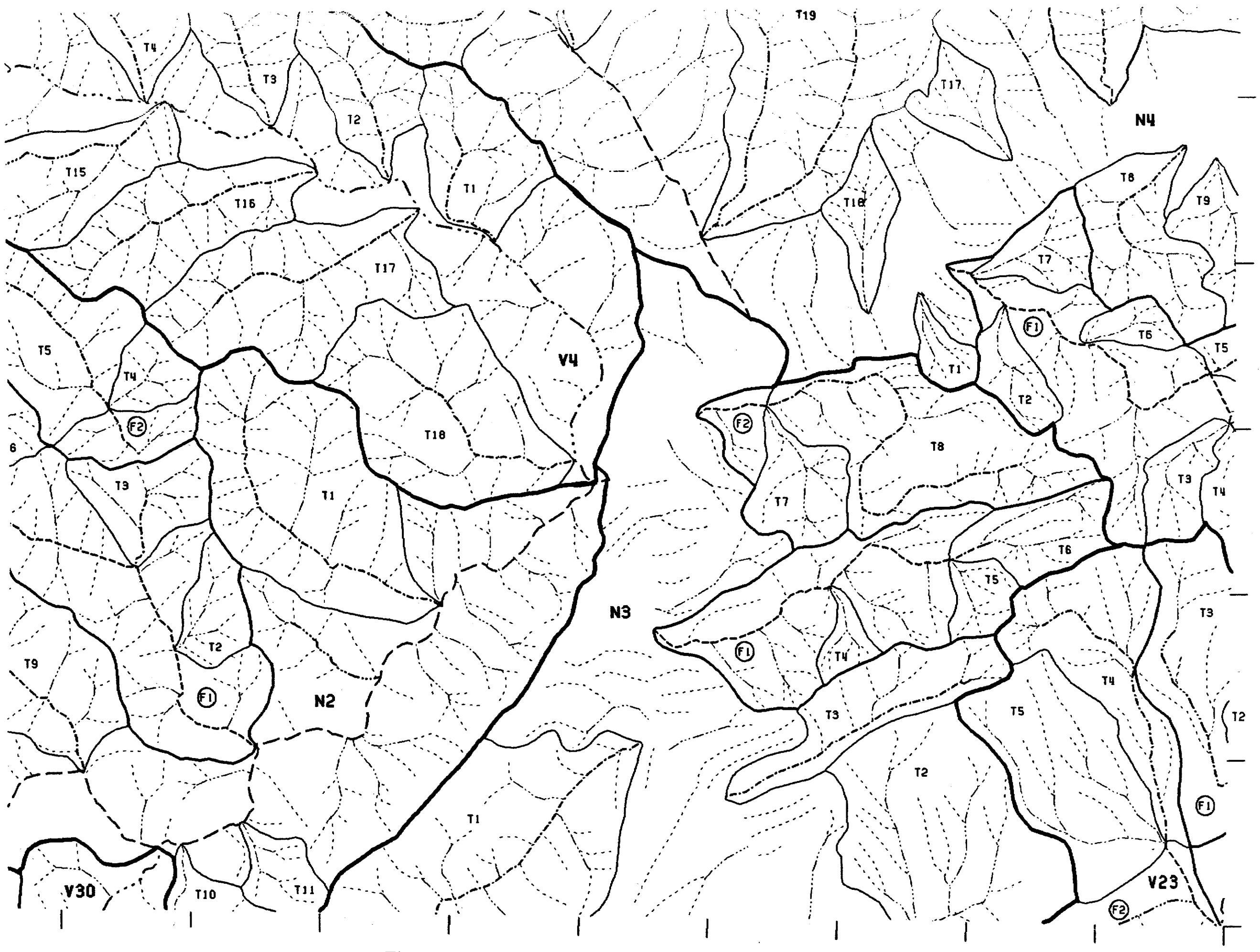

Fig. 1. Digitized map of the stream data used in our system.

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_CELL's on the first two levels, and so forth. Thus, the tree is space saving for relations whose tuples have many identical components. To facilitate searching the tree, and hence the relation, the N-tuples are stored in lexigraphical order.

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. Entities such as polygons whose points must remain in a fixed predefined order are stored in this form.

Fig. 3 illustrates the physical structure of the spatial data structure for a region and its A/V relation in our experimental system. The A/V relation is tree-structured.

B. 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.

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 are 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.

IV. High-Level Operations

In order to understand the architectural needs of the system, we need to look at the high-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.

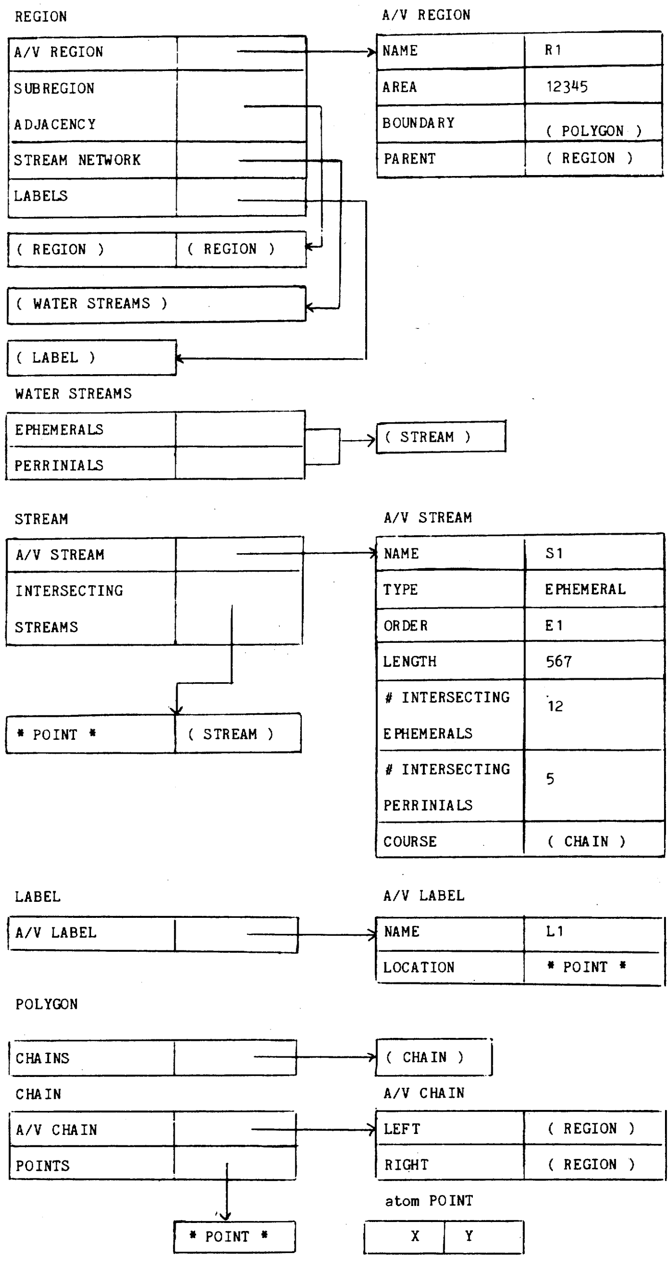

Fig. 2. Prototypes for spatial data structures REGION, WATER STREAMS, LABEL, POLYGON, and CHAIN.

Q1) How many rivers (perrinial 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 (perrinial 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 pass through?

Q10) 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 lookup 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. Low-level functions are discussed in [21]. 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 constraint,

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.

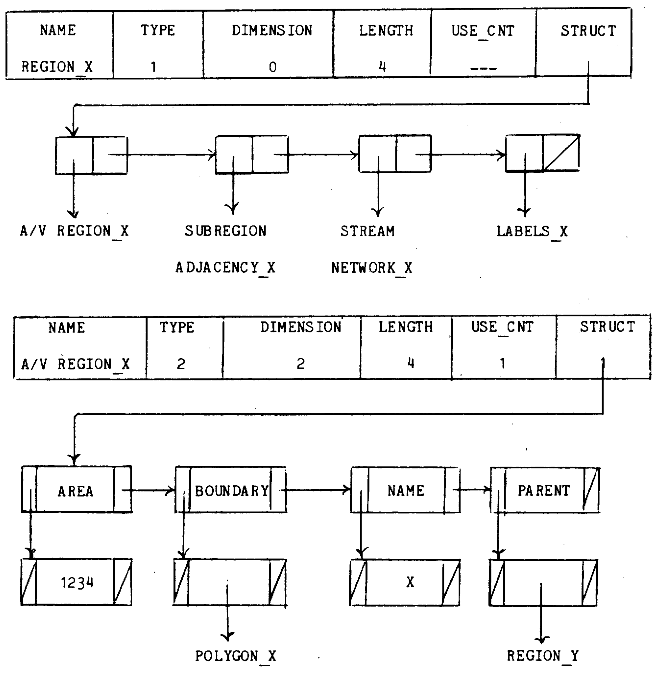

Fig. 3. Spatial data structure for a REGION and its attribute-value relation.

These operations suggest that a generalized form of the now standard relational database operators will be useful in a spatial information system. In the remainder of this section, we define what these generalized forms are. Let R be an N-ary relation and let S be an M-ary relation. Let P be an N-ary predicate and let Q be an $(N + M)$-ary predicate. Let I be the set of positive integers and let f be a binary relation over I.

A. Projection

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

B. Selection

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

C. Join

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

D. Division

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

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.

V. Architectural Implications

A. 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 an unchanging 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. Process all subsets in parallel. Then 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 efficient 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. This method has been used in [11].

Connected to the input CPU is an input selection CPU whose job it is to divide the relation into mutually exclusive subsets, each of which is processed in parallel by the number-crunching CPU's which are connected to the input CPU. The number-crunching CPU's in turn are connected to an output selection CPU which is connected to an output CPU. The output selection CPU is only active in division, and it determines whether or not to pass a tuple to the output CPU based on whether all the number-crunching CPU's select the tuple. The output CPU acts like an output queue to the mass storage devise. Fig. 4 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 Fig. 4.

B. 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.

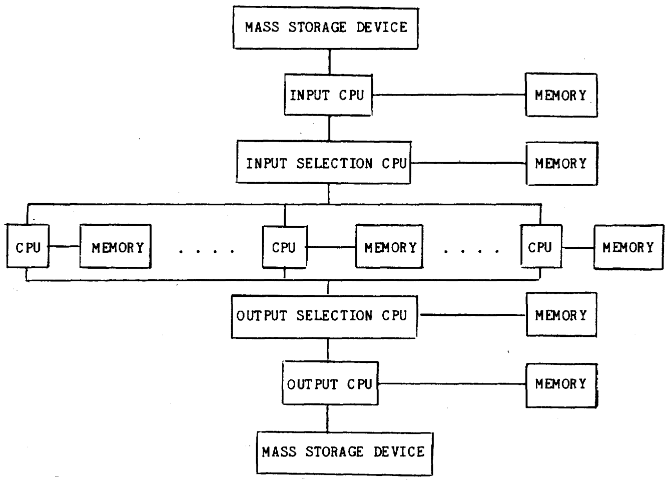

Fig. 4. Block diagram of the multiprocessor spatial information system.

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.

C. Selection

Selection works in a similar way to projection. Tuples are read by the input CPU. Each tuple is handed to the selection CPU which 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.

D. 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 handle the fact that the memory is not large enough to hold both relations, only small segments from each relation are stored in memory at once, and the relation files have to be read a multiple number of times.

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 selection 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 sized relation as it can. Of course, different number-crunching CPU's store mutually exclusive groups of tuples. Then the larger sized 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.

E. 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 that the tuple has passed, then it sends the tuple to the output CPU to be added to 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 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 indexes will not be useful in the kind of system we envision. We have suggested storing relations by column instead of by tuple and the 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 an intelligent front-end processor.

References

[1] H. G. Barrow and J. M. Tenenbaum, "Recovering intrinsic scene characteristics from images," in *Computer Vision Systems*, A. Hanson and E. Riseman, Eds. New York: Academic, 1978.

[2] E. D. Carlson, J. L. Benett, G. M. Gidding, and P. E. Mantey, "The design and evaluation of an interactive Geo-Data Analysis and Display System," in *Proc. IFIP Congr. 1974*. Amsterdam: The Netherlands, North-Holland, 1974.

[3] N. S. Chang and K. S. Fu, "Query-by-pictorial-example," *IEEE Trans. Software Eng.*, vol. SE-6, Nov. 1980.

[4] S. K. Chang, B. S. Lin, and R. Walser, "A generalized zooming technique for pictorial database systems," in *Proc. Nat. Comput. Conf.*, 1979, pp. 147–156.

[5] S. K. Chang, J. Reuss, and B. H. McCormick, "Design considerations of a pictorial database system," *Policy Anal. Inform. Syst.*

[6] E. F. Codd, "A relational model of data for large shared databases," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 377–389, June 1970.

[7] A. Go, M. Stonebraker, and C. Williams, "An approach to implementing a geo-data system," Electron. Res. Lab., Coll. Eng., Univ. California, Berkeley, Memo. ERL-M529, 1975.

[8] P. J. Hagan, "A network data model for cartographic features," D.Sc. dissertation, Sever Inst. Technol., Washington Univ., St. Louis, MO, May 1980.

[9] A. R. Hanson and E. M. Riseman, "VISIONS: A computer system for interpretating scenes," in *Computer Vision Systems*, A. Hanson and E. Riseman, Eds. New York: Academic, 1978.

[10] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Potomac, MD: Computer Science Press, 1977.

[11] A. T. F. Hutt, *A Relational Data Base Management System*. New York: Wiley, 1979.

[12] *POLYVRT: A Program to Convert Geographic Base Files*, Lab. for Comput. Graphics, Harvard Univ., Cambridge, MA, 1974.

[13] M. D. Levine and S. I. Shaheen, "A modular computer vision system for picture segmentation and interpretation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-3, pp. 540–556, Sept. 1981.

[14] D. Marr, "Representing visual information—A computational approach," in *Computer Vision Systems*, A. Hanson and E. Riseman, Eds. New York: Academic, 1978.

[15] M. Modeleski, *Topology for INGRES: An Approach to Enhance Graph Property Recognition by GEOQUEL*, Geographic Database Coordinator, Ass. Bay Area Governments, Berkeley, CA 94705, 1977.

[16] T. K. Peucher and N. Chrisman, "Cartographic data structures," *Amer. Cartographer*, vol. 2, pp. 55–69, Apr. 1975.

[17] L. G. Shapiro and R. M. Haralick, "A spatial data structure," *Geo-Processing*, vol. 1, pp. 313–337, 1980.

[18] W. A. Switzer, "The Canada Geographic Information System," in *Automation in Cartography*, J. M. Wilford-Brickwood, R. Bertland, and L. Van Zuylen, Eds. The Netherlands: Int. Cartographic Ass., 1975.

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

[20] "Census use study: The DIME Geocoding System," U.S. Bureau of Census, Washington, DC, Rep. 4, 1970.

[21] P. D. Vaidya, L. G. Shapiro, R. M. Haralick, and G. J. Minden, "Design and architectural implications of a spatial information system," Dep. Comput. Sci., Virginia Polytechnic Inst., Blacksburg, Tech. Rep. CS82003-R, Dec. 1981.

[22] W. Weber, "Three types of map data structures, their ANDs and NOTs, and a possible, OR," in *Proc. 1st Int. Advanced Study Symp. Topological Data Structures for Geographic Inform. Syst.*, Harvard Univ., Cambridge, MA, 1978.