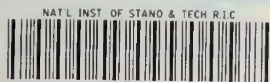

NAT'L INST. OF STAND & TECH R.I.C.
A11104 149335

NIST PUBLICATIONS

NISTIR 5258

Towards SQL Database Language Extensions for Geographic Information Systems

Edited by:
Vincent B. Robinson

Institute for Land Information Management
University of Toronto
Mississauga, Ontario L5L 1C6
Canada

Henry Tom

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

QC
100
.U56
#5258
1993

NIST

NISTIR 5258

Towards SQL Database Language Extensions for Geographic Information Systems

Edited by:
Vincent B. Robinson

Institute for Land Information Management
University of Toronto
Mississauga, Ontario L5L 1C6
Canada

Henry Tom

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

August 1993

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arati Prabhakar, Director

Table of Contents

Title and Author(s) *page*

Foreword i

Henry Tom
National Institute of Standards and Technology

On Heterogeneous Geographic Information Systems, Architectures, Spatial Data Models, Transactions and Database Languages 1

Vincent B. Robinson and D. Scott Mackay
University of Toronto

Database Language SQL: Emerging Features for GIS Applications 36

Leonard Gallagher
National Institute of Standards and Technology

Proposed Spatial Data Handling Extensions to SQL 69

Orest Halustchak
GeoVision Systems Inc.

A Geographic Information Systems Perspective on Spatial and Object Oriented Extensions to SQL 85

Mark Ashworth
Computervision GIS

Conceptual Folding and Unfolding of Spatial Data for Spatial Queries 133

Jan W. van Roessel
ESRI

Foreword

This report is a collection of papers by the GIS/SQL Working Group, NIST GIS Standards Laboratory. The working Group is chaired by Dr. Vincent B. Robinson, Institute for Land Information Management, University of Toronto. The focus of this report was established during the GIS/SQL Workshop held at the National Institute of Standards and Technology (NIST), April 3-4, 1991, Gaithersburg, MD, sponsored by the GIS Laboratory. A prime objective of the GIS Standards Laboratory is to facilitate joint efforts in adapting information technology standards for use by the GIS community.

A Geographic Information Systems (GIS) extension to the Structured Query Language (SQL) has significant implications. For the GIS world, a GIS extension to SQL benefits both GIS software vendors and end users. The integration of SQL into GIS software will require less effort and maintenance by GIS vendors. GIS users will have a common database language to directly perform many of the functions and operations they require. A GIS/SQL extension can provide the transactional capability with the particular features required in GIS processing.

Other advantages of incorporating GIS capabilities into the current SQL refinements include distributed GIS processing through Remote Data Access (RDA) and potential adaptation of object-oriented constructs by relational-based GIS technology. Accordingly, a GIS extension to SQL, within the larger context of generic information technology standards, affords GIS technology a greater open systems capability. For the general computing community, the GIS extension to SQL provides fundamental GIS functions without specialized GIS software.

The intent of this report is to stimulate dialogue and development of a GIS extension to SQL among those interested. Internationally, a new project in this regard has been approved within the ISO/IEC Joint Technical Committee 1 (JTC1), Information Technology. Nationally, the recently formed technical committee for GIS standards X3L1 will consider participating in such efforts. As these efforts progress and become formalized, the adoption of a GIS extension to SQL will greatly benefit Federal GIS users.

The views expressed by the authors of this report are their own and do not necessarily reflect those of NIST. Citations, inclusion or omission of specific vendors or commercial products does not imply either endorsement or criticism by NIST.

Henry Tom
Manager
NIST GIS Standards Laboratory

On Heterogeneous Geographic Information Systems, Architectures, Spatial Data Models, Transactions and Database Languages

Vincent B. Robinson and D. Scott Mackay
Institute for Land Information Management
University of Toronto, Erindale College
Mississauga, Ontario L5L 1C6
Canada

1 Introduction

This paper addresses several essential concepts, developments, and issues pertaining to Geographic Information Systems (GIS) and development of a SQL database language for GIS. Suggestions or proposals for database language extensions to SQL are collectively addressed by Ashworth (1993), Gallagher (1993), Halustchak (1993), and van Roessel (1993). This paper sets a general context for considering SQL database language extensions for GIS.

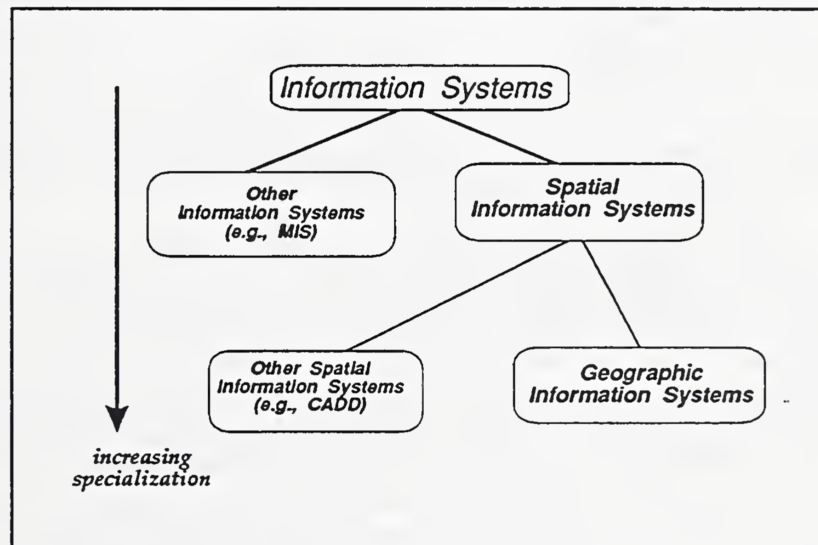


Figure 1. Geographic information systems as a specialized information system.

An important implication of the Mackay and Robinson (1992) discussion of heterogeneous information systems and geographic data interchange is the central role a database language such as SQL may play in providing a unified model for the management of geographic data. Meeting this challenge requires that many issues be addressed (Robinson 1991). Foremost is to place GIS in context with other areas of information technology (IT). GIS are a specialised information, or database, system that deals with geographically referenced, spatially structured information (Bracken and Webster, 1989). Although GIS may have some special characteristics, they have many of the needs and characteristics of general purpose information systems and of spatial information systems (Figure 1). GIS are but one, albeit specialized, area of information management technology. Therefore it is affected by many of the same trends and developments found generally in the IT field.

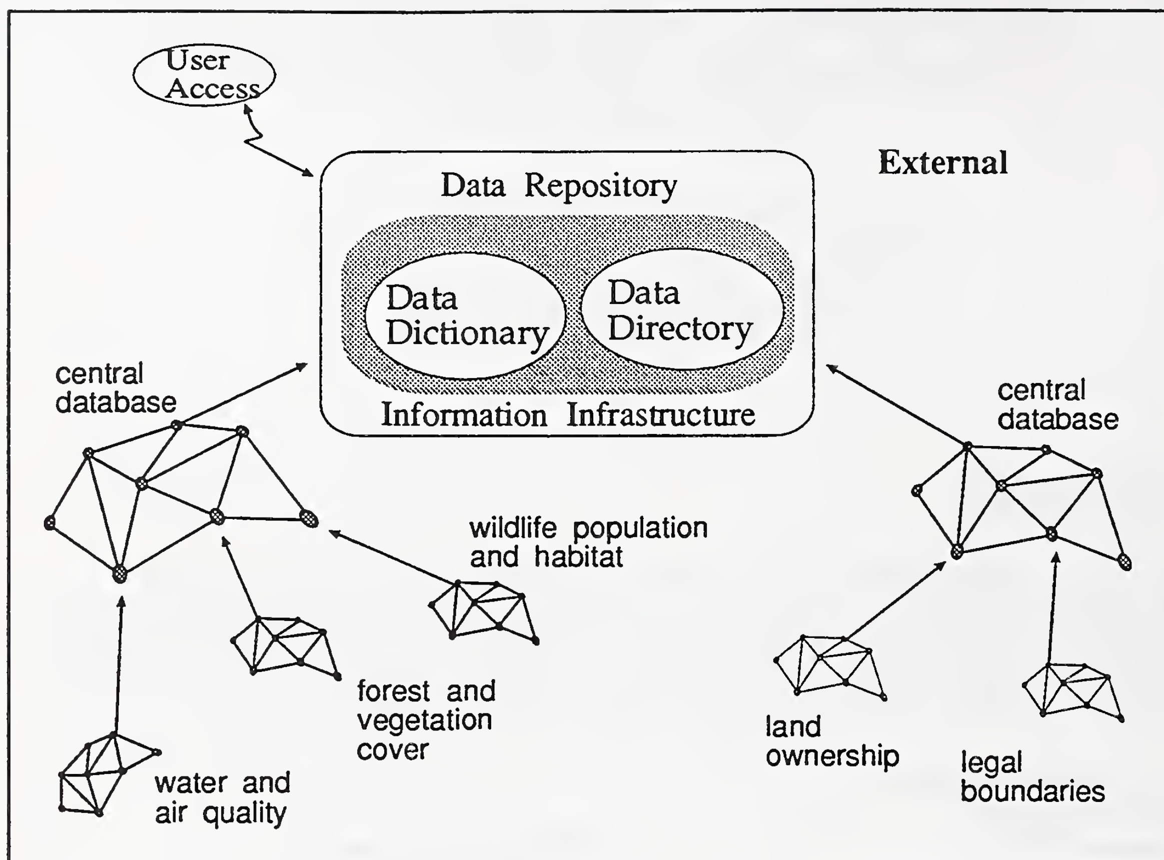


Figure 2. Conceptual structure of a geographic information infrastructure (source: Mackay and Robinson 1992).

Within the last decade a significant development in the GIS industry has been that of *enterprise GIS*. Traditionally GIS filled the role of a technical subsystem often isolated from other information resources of an organization. Enterprise-wide GIS is a broad multi-department driven, database technology (GeoVision 1992). It is because of this broad, organizationally driven database perspective that in many parts of the world (*e.g.*, USDI 1989, 1990; DMR 1989) there is a clear trend towards developing integrated geographic information systems consisting of geographically distributed systems (Figure 2). The cultivation of these systems demands the use of one or more models of heterogeneous or multidatabase systems. Mackay and Robinson (1992) argue that geographic data interchange standards efforts represent attempts at heterogeneous geographic database design. They observe that the maturation of such systems will promote the need for effective multidatabase languages systems that enable GIS to become actively integrated into broad, organization-wide database systems (Mackay and Robinson 1992).

To understand some of the complexities of developing a SQL database language extension to support the integration of GIS into a heterogeneous information systems environment we briefly present the two prevailing GIS database architectures, spatial data models, geographic database transactions and GIS database languages.

2 GIS Database Architecture

GIS handle data describing both spatial location, extent, *etc* and characteristics of spatial entities located in geographical space. These dual demands that have given rise to two major database architectures in the GIS industry - dual and unified database architectures. The *dual database model* stores and manages spatial and nonspatial (attribute) data in independent structures which are often separate software systems. The *unified database model* brings spatial and attribute data together into a single database framework (Bracken and Webster, 1989). (Figure 3).

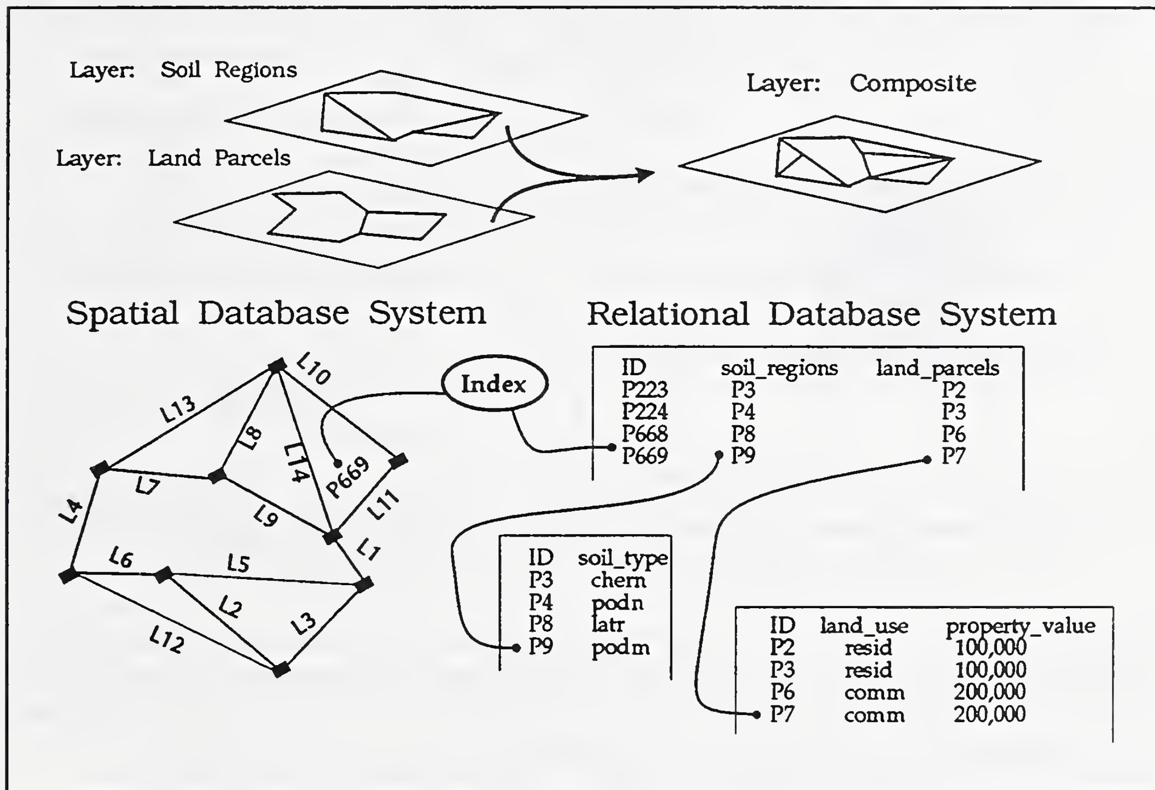


Figure 3. Dual database GIS showing a composite layer as a layer of *indices* pointing to attributes.

Since query languages (QL) are the primary means for users to access and manage existing databases, they can become an expression of how a GIS is integrated with other organizational functions. This leads to an interesting observation regarding the dual vs unified database models. In the dual database model the nonspatial, or attribute, information is kept in a relational database and accessed using a database language, such as SQL, while spatial data is accessed using a specialized spatial query language. The dual database architecture grew out of past limitations of nonspatially oriented database technology when applied to spatial problems of databases (*e.g.* Chang and Fu 1981, Webster and Bracken 1989,) and the need to link to existing nonspatial databases.

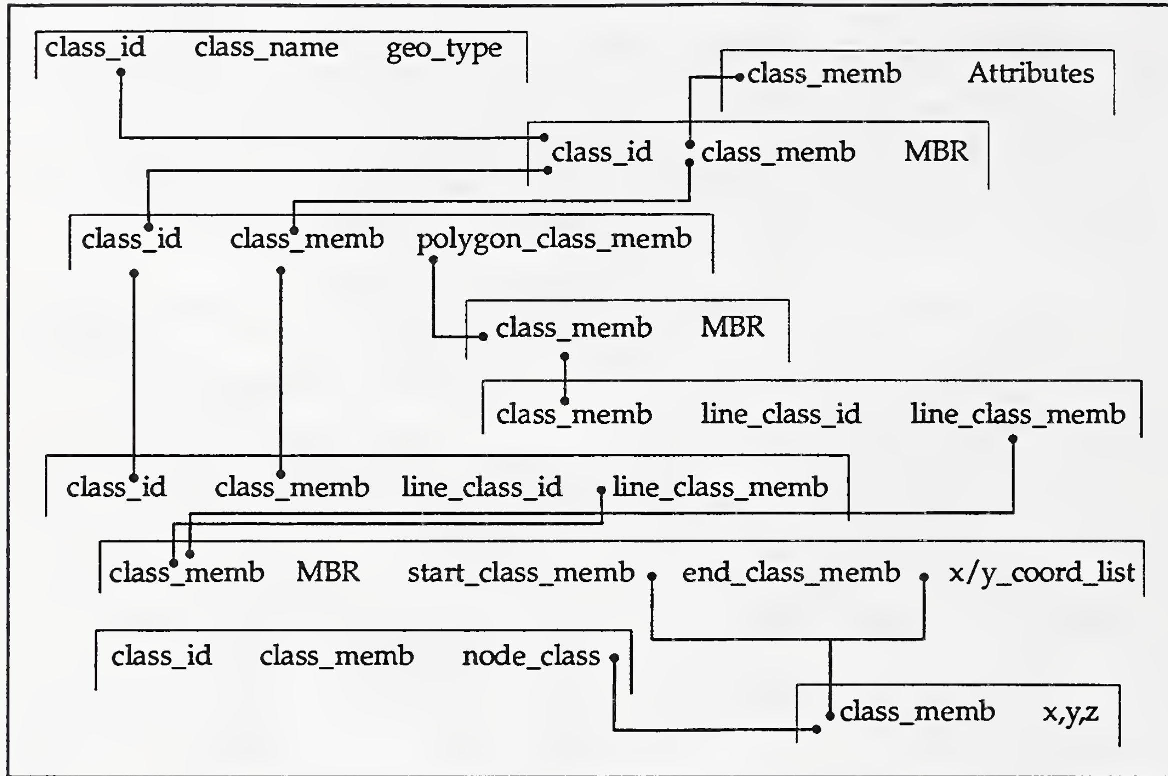


Figure 4. Example of schema for a unified geographic database model based on relational database technology with extensions for GIS (source: Ashworth 1992).

The *unified database* approach manages spatial and nonspatial data together with a single database schema and language (Figure 4). An important reason for maintaining a geographic database within a single database management system is that all the data can then be subject to facilities such as security, integrity control, communication, and transaction management. For example, in land information systems that receive updates regularly the importance of referential integrity is especially important. Without a unified database management system, referential integrity can easily be lost (Robinson 1989).

Often systems based on the unified database model have been built using relational database technology (*e.g.* Ashworth 1993; Halustchak 1993). These unified database systems usually exploit *bulk-field* types in relational databases to enable efficient handling of spatial data. To overcome access performance problems spatial access methods based on R-tree or quad-tree indexing is often exploited (*e.g.*, Ashworth 1993; Faloutsos et al 1987; Westwood and Brinkman 1988).

3 Spatial Data Models

Historically use of a raster or vector based data model had a fundamental impact on the database, or query, language. For example, QPE-based spatial query languages have tended to be built upon a raster-based model while SQL-based languages have tended to be built upon a vector-based model (Figure 5). The language of cartographic modeling (Tomlin 1990) also has its roots in the raster model of data organization, however is being adapted to the vector-based domain of SQL-based GIS products (e.g., Ashworth 1993). Although there are transforms that allow systems to map between these models the process is not without its difficulties. Because current spatial query languages are usually tightly coupled to the underlying data model, the data model determines the how the language is structured and the meaning of its vocabulary.

More relevant to the problem of heterogeneous GIS in an enterprise are the layer-based and object-based spatial data models (Worboys and Deen 1991). The general distinction between the layer vs object based models is that a layer is a function from a set of spatial references to an attribute set, thus providing information on the gobal variation of an attribute over the layer. In contrast, the object-based approach models the information structure as being populated with constituent objects which have to spatial objects. Consequently, one model can be view as the inverse of the other.

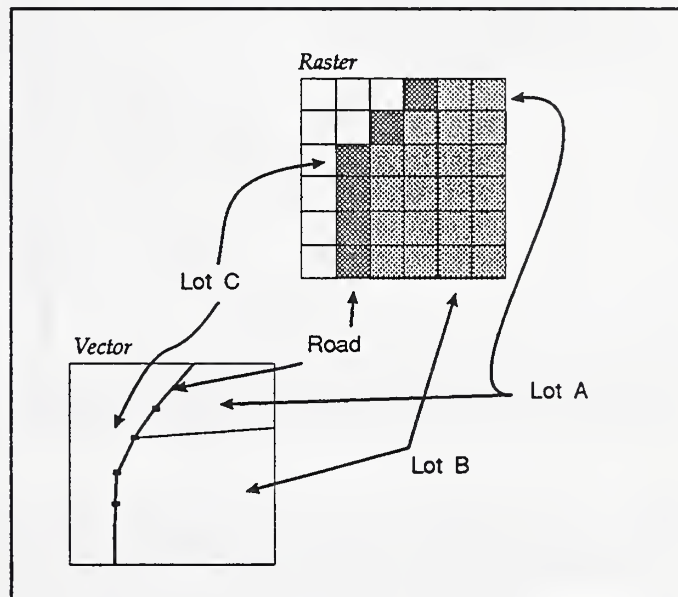


Figure 5. Simple contrast of Raster and Vector models of spatial data representation.

3.1 Layer-based Spatial Data Model (LSDM)

The layer-based spatial data model consists of a finite collection of layers $\{L_i \mid 1 \leq i \leq n\}$ and assume an underlying spatial framework as set Γ of spatial references. For example, Γ might be the points of a regular grid such as a digital elevation model. For $1 \leq i \leq n$, each layer L_i is a function from set Γ to an attribute set A_i. L_i is a function from a set of spatial references to an attribute set, thus providing information on the global variation of A_i over L_i. For example, a layer of soil polygons maps an arbitrary partitioning of a geographic space onto attributes from some soil domain (*e.g.* series). Thus, the layer-based model of geographic information represents spatially distributed-attributes as a set of data layers each of which defines a distribution of a single attribute over a well defined space (Figures 3 and 6).

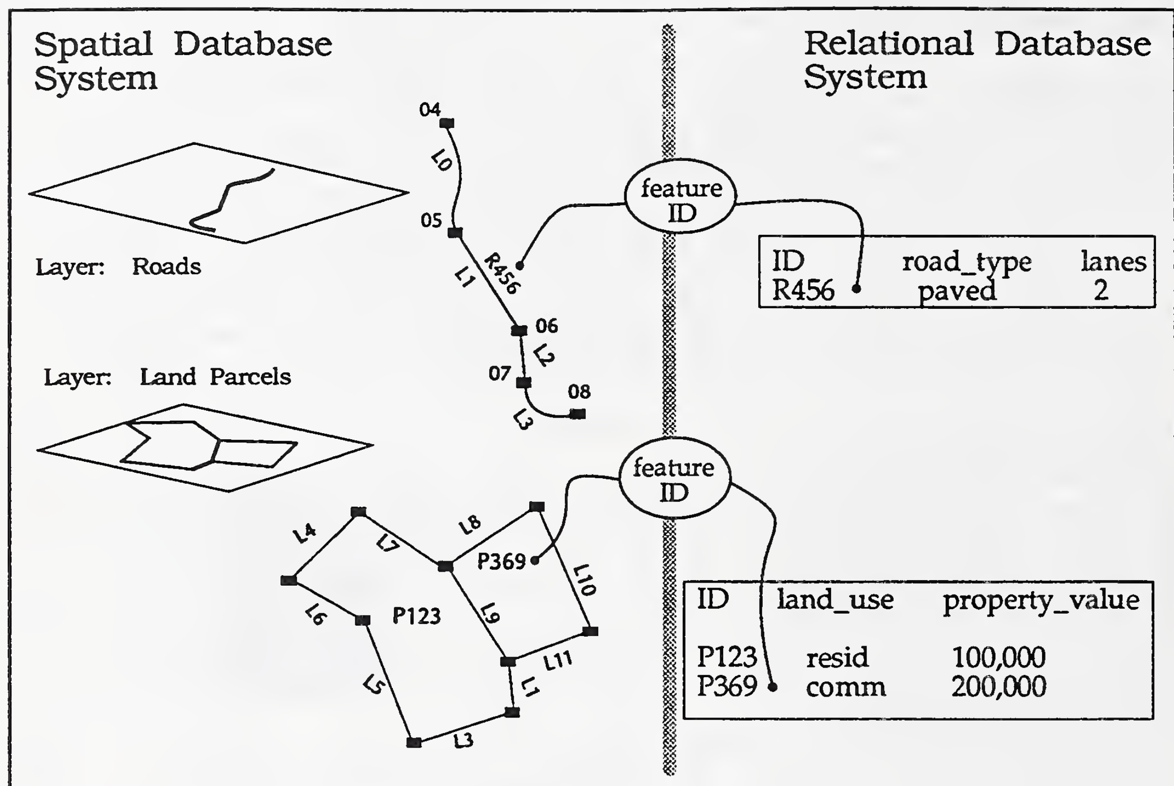


Figure 6. GIS organized on the layer spatial data model using a dual database architecture.

Layers are often organized so that points, lines, and polygons (i.e., spatial primitives) are stored in separate layers. For example, well sites represented by points might be stored in one layer, with roads, represented by lines, in a second layer with land parcels, represented by polygons, in a third layer (ESRI 1991). In addition, it is common to organize the layers by *theme* where parcels and roads are on separate layers (Figure 6). With dual-database systems it is often recommended to organize layers along joint theme/spatial primitive lines since the associated attributes differ so dramatically that it would be difficult to represent the associated attributes in both the spatial and relational systems.

The algebra of a layer-based model is specified by giving the layers and the operations on the layers. Layer-based operations take as arguments existing layers and produce a new layer. Generally speaking operations may be local, focal, or zonal where (Tomlin, 1990) :

Local operations. An attribute is created at location $\mathcal{L}$ which depends on the attributes of $\mathcal{L}$ associated with the layers which are arguments of the operation (Figure 7).

Focal operations. An attribute created at $\mathcal{L}$ depends not only on the appropriate attributes of $\mathcal{L}$ but also on the attributes in the neighborhood of $\mathcal{L}$. Given framework Γ, a neighborhood function $n: \Gamma \rightarrow \wp(\Gamma)$ may be defined which associates with each location x a set of locations within a specified distance of $\mathcal{L}$.

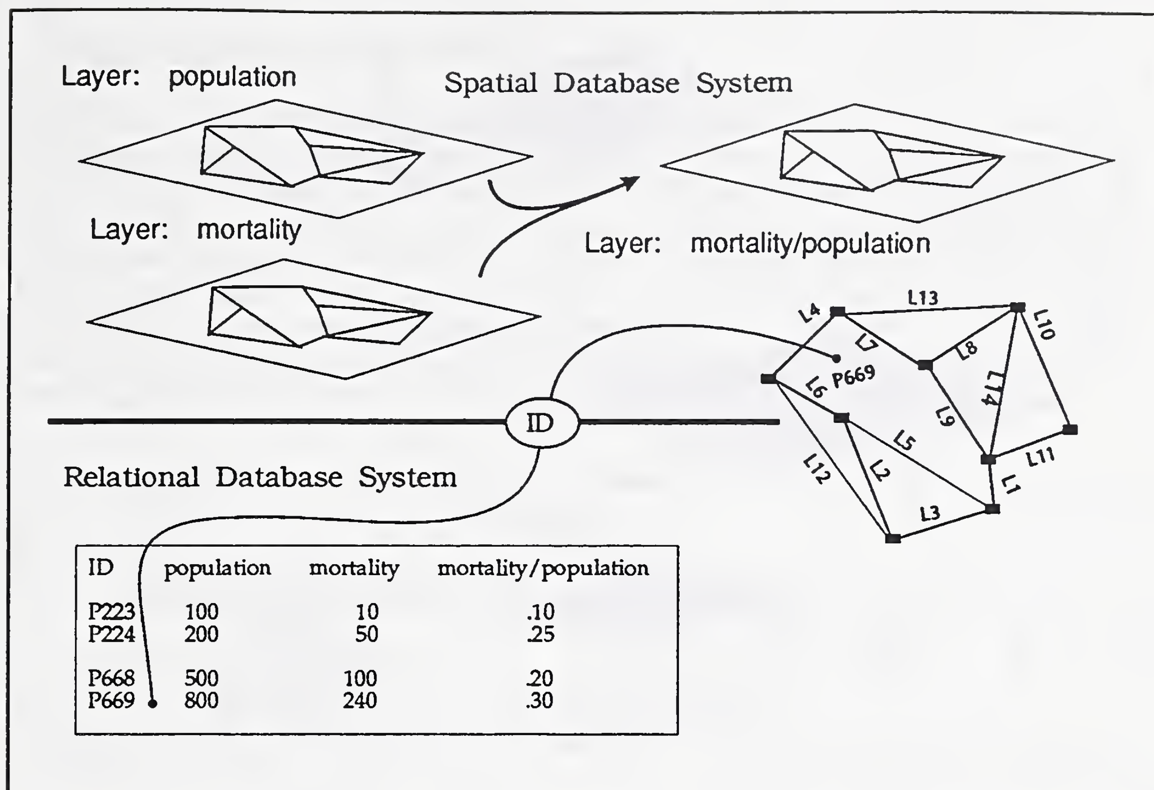


Figure 7. Local operation where *mortality per 100 persons* depends on layers of total population and total mortality for each location.

Zonal operations. An attribute created at location $\mathcal{L}$ is dependent upon the appropriate attributes within the zone containing $\mathcal{L}$. Given a layer L_i , a zone is a subset of Γ, the values of whose attributes satisfy a predefined condition such as in the case where the zone is elevation above 120m. A zoning of Γ is a partition of Γ into disjoint zones whose union is Γ.

The layer-based approach is efficient for performing locally-based spatial operations, including point-based operations (*e.g.* leaf area index calculated ratio of band 4 and band 3), neighborhood operations (*e.g.* deriving gradient at point $\mathcal{L}$ using finite difference calculations of elevations within a small neighborhood of $\mathcal{L}$), and zoning operations (*e.g.* given soil polygons in one layer and gradient in another layer, compute mean gradient for each soil polygon). Note that these operations do not require explicit coding of topological relationships. Explicit topology is not provided within layers of a raster-based GIS, so topology must be stored external to layers (dual-database model) for a vector-based GIS.

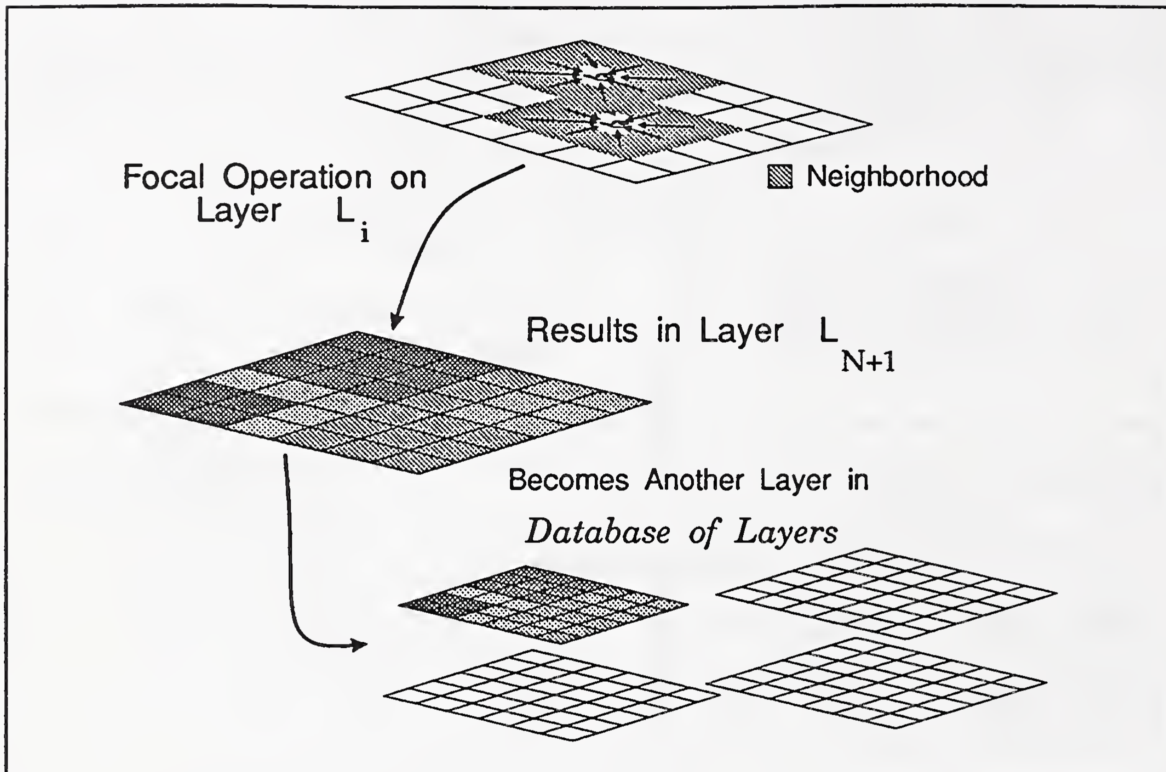


Figure 8. Concept of a focal operation in the layer model.

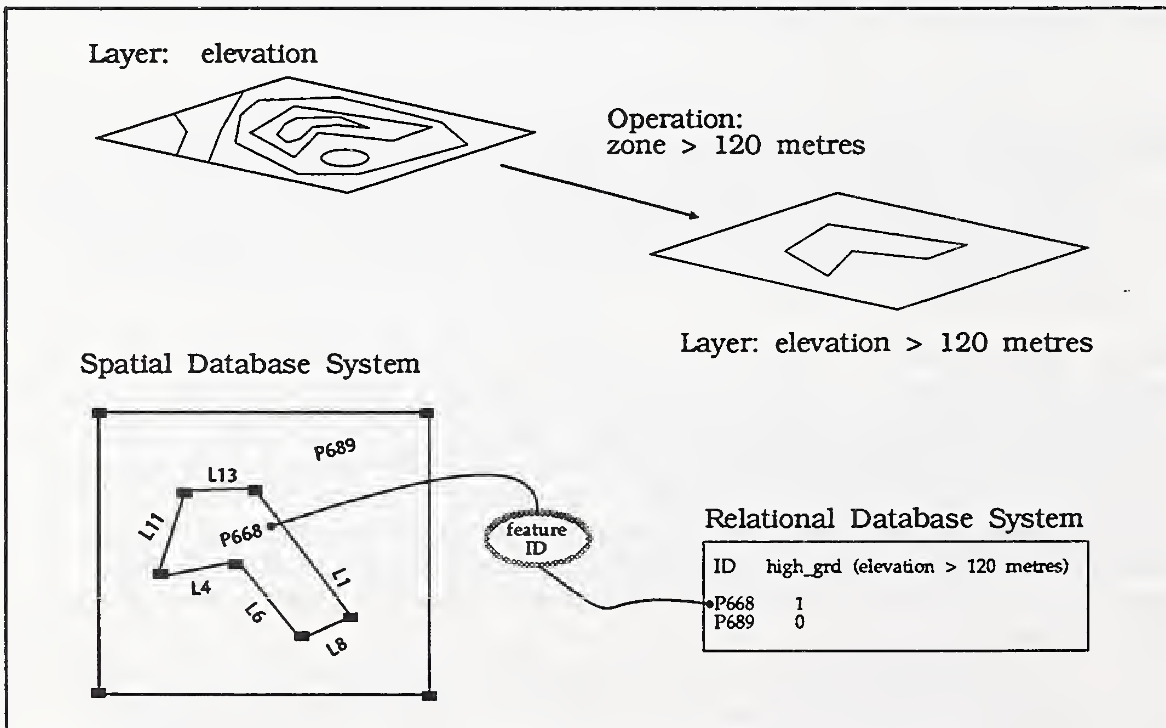


Figure 9. Concept of a zonal operation in the layer model.

3.2 Object-based Spatial Data Model

While the layer-based model provides information on the global variation of an attribute over a defined space, the object-based spatial data model (OSDM) views the information structure as populated by component objects which have as attributes references to spatial objects. The OSDM assumes the usual object data model (e.g., Mohan and Kasyap, 1988; Drabowski et al 1990) which has at its core an entity, the object, that combines the properties of procedures and data. The fundamental aggregate unit in such object representations is the *class*. A class is an intentional description of data and the various instances of a class form the extensional data. In addition, objects in the OSDM have attributes which inherit properties from generic spatial objects such as point, line, and polygon (Figure 10).

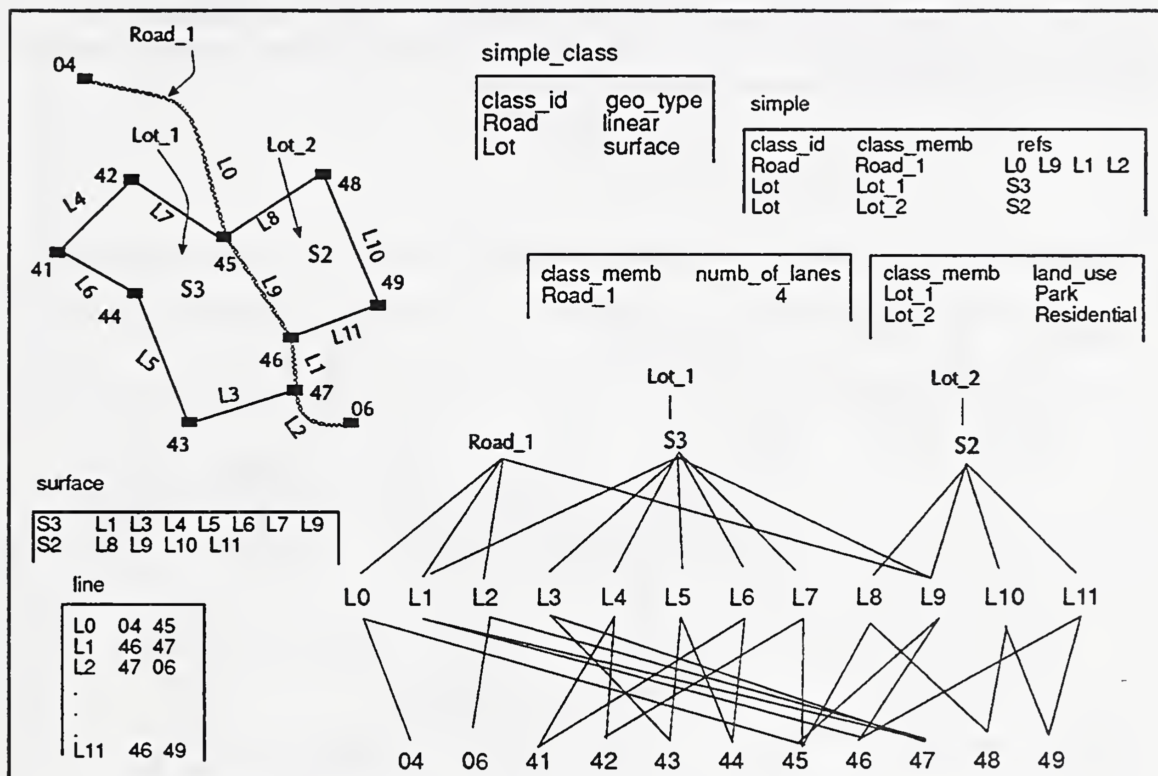


Figure 10. Object spatial data model treats geographic objects as component parts of information strucutre.

Three general classes of primitive spatial operations describe OSDM:

Euclidean operations (Figure 11) include measurements not only of length but also of angle. Some have broken operations that involve measurement of length out as a separate category such as metric operations (Worboys and Deen 1991).

Topological operations (Figure 12) depend upon the topological structure of the space. Examples include the Boolean operation to determine whether two regions are adjacent, and the Boolean operation to determine whether a region is connected.

Set-based operations (Figure 13) treat spatial objects as sets. The intersection, or union, of two regions would be an example of a set-based operation.

It is clear that the basic idea of objects and their importance in GIS database languages will continue to exert substantial influence on their development. A common assumption is that users tend to organize their model of geographic information and processing around the idea of geographic objects. But there are other advantages as well.

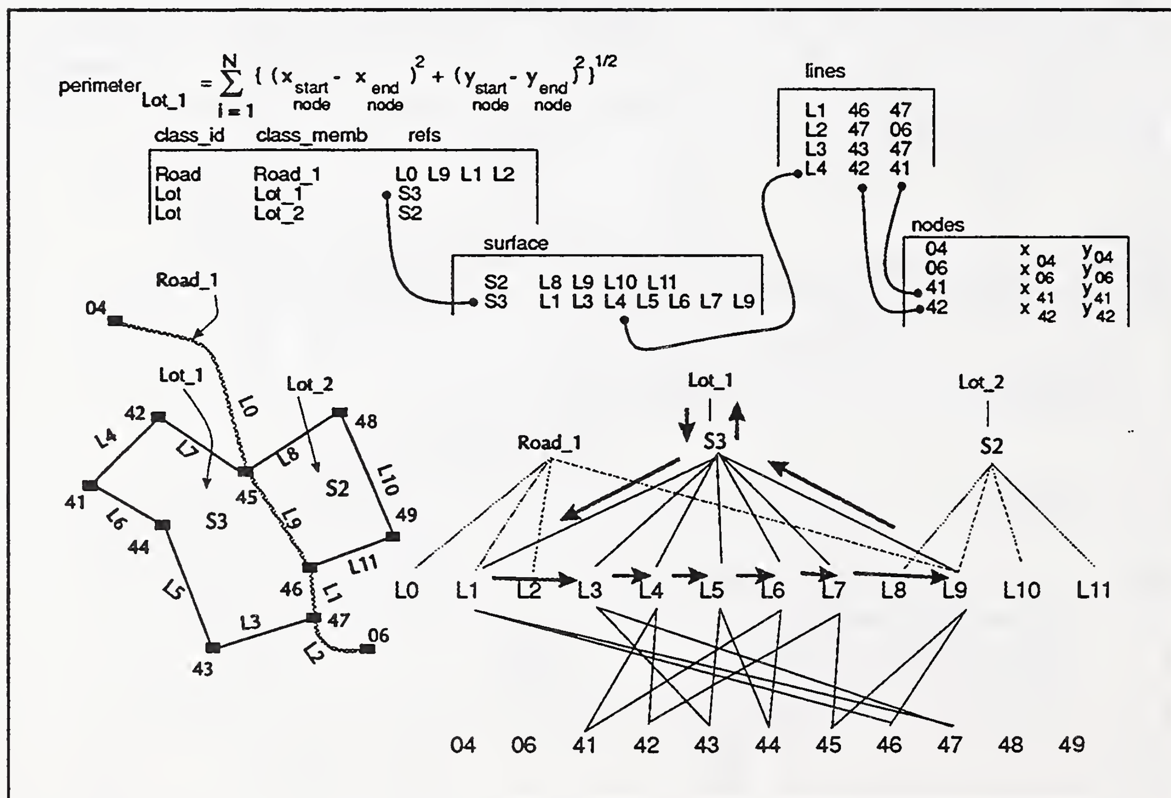


Figure 11. Euclidean operation in object spatial data model and relation to information structure.

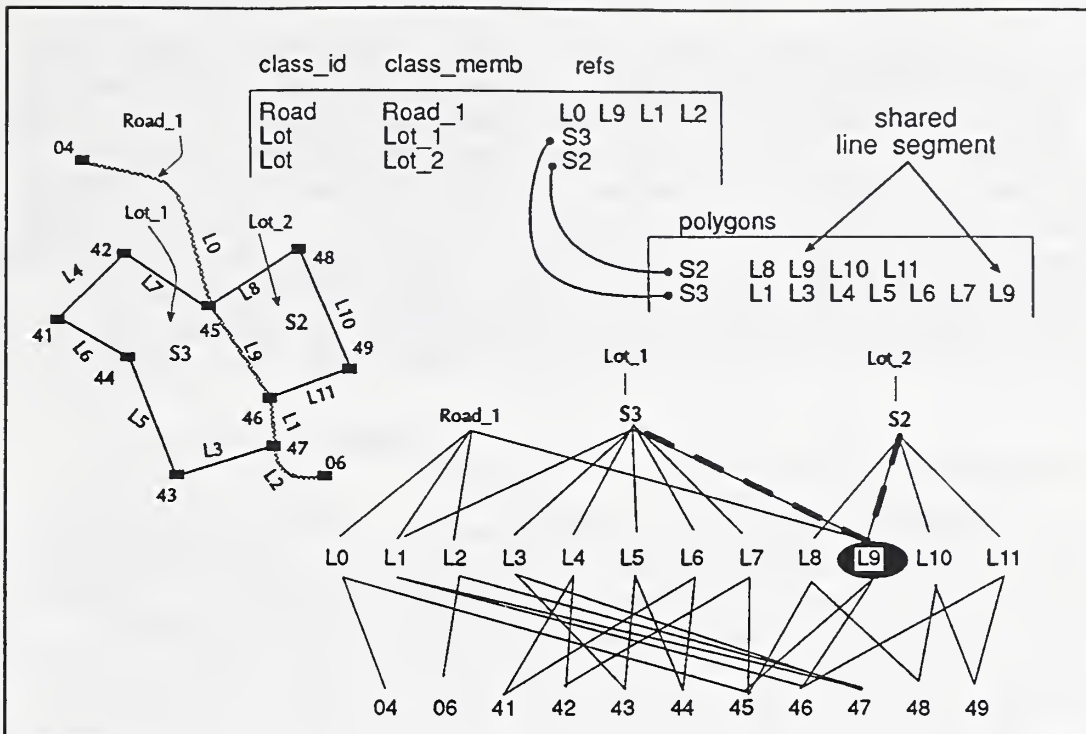


Figure 12. Topological operation to determine if two spatial objects share a boundary.

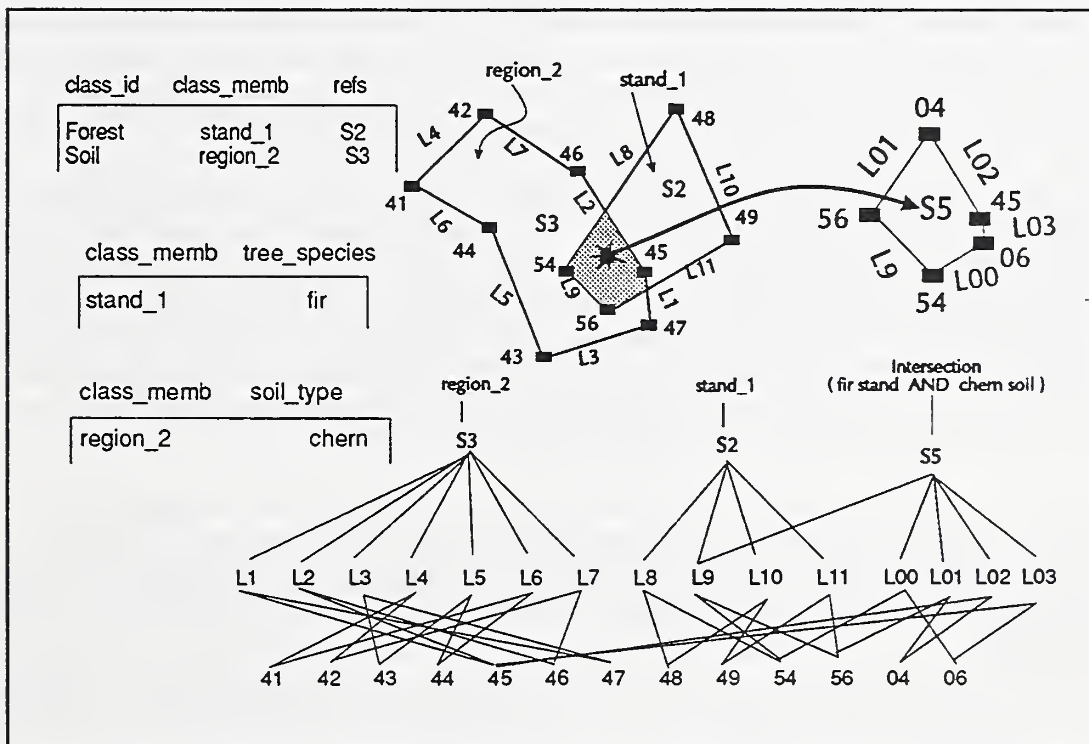


Figure 13. Set-based operation to determine the intersection of two geographic objects.

The object-based model combines generic spatial objects (*e.g.* points, lines, polygons), explicit spatial relations between objects (*e.g.* lines have left and right polygons), and attributes (*e.g.* size) into a unified structure that allows for easier maintenance of updates to a spatial database. The object-based model thus avoids problems associated with maintaining a dual database model (Bracken and Webster, 1989). Important to GIS database managers, this approach can more easily capture integrity contraints by exploiting the structure of an object class. Special functions may be tagged to a class of objects to keep a data integrity check on object instances of that class (Mohan and Kashyap 1988). Furthermore, a unified object-based model is particularly important in developing spatial decision support systems (Armstrong and Densham, 1990), and knowledgebased GIS applications (Robinson et al 1989; Mackay et al 1993).

3.3 Layer-Object Transformation

LDSM and ODSM are not mutually exclusive. It is possible to translate from layer-to-object and back. Worboys and Deen (1991) suggest that a suitable model of transformation between the layer and object models is point-set topology. As an example, Mackay et al (1993) focus on a layer-to-object transformation in a knowledgebased approach to forest ecosystem modeling. However, the object-to-layer transformation could be used to generate maps showing daily, monthly or annual snapshots of simulated variables. Since Mackay et al (1993) are working mainly with raster data (*e.g.* DEM and satellite imagery) their point-set topology is represented by pixels and their neighborhoods.

In Mackay *et al* (1993) transformation from image data to symbolic elements requires a three-step process (Figure 14): (1) topographic partitioning of drainage basins into hillslopes and stream links, (2) analysis of the hillslope and stream links, and (3) generation of an object-based database. Terrain partitioning uses techniques described by Band (1989). In this approach a drainage area transform is recursively computed, then pruned to form a tree of stream links. Hillslopes are attached to each stream link as left and right polygon areas. The strength of this approach is its flexibility in scaling terrain partitioning from a few, large hillslopes to many, small hillslopes for any given drainage basin, by adjusting the constraints on how the drainage area transform is pruned. It is this scale adaptability that allows layer-to-object transformations without loss of essential spatial heterogeneity. The spatial heterogeneity of land surface properties is required for realistic forest ecosystem simulations. The scaling tool retains enough physical basis so that processes vary spatially while also allowing for direct measurement of land surface and environmental parameters.

As Mackay *et al* (1993) note a major advantage of the object model is the ease with which topological operations are performed on objects. Since many query languages are based on First Order Logic, and FOL treats propositions (individuals or attributes) as objects, the object-based model of GIS provides a data model that is compatible with many query languages, including SQL.

As interest in object-based and object-oriented approaches to GIS database languages continues to grow it is apparent that there are two architectures being used to develop such systems. One approach includes an object-based database language as an integral part of an object-oriented

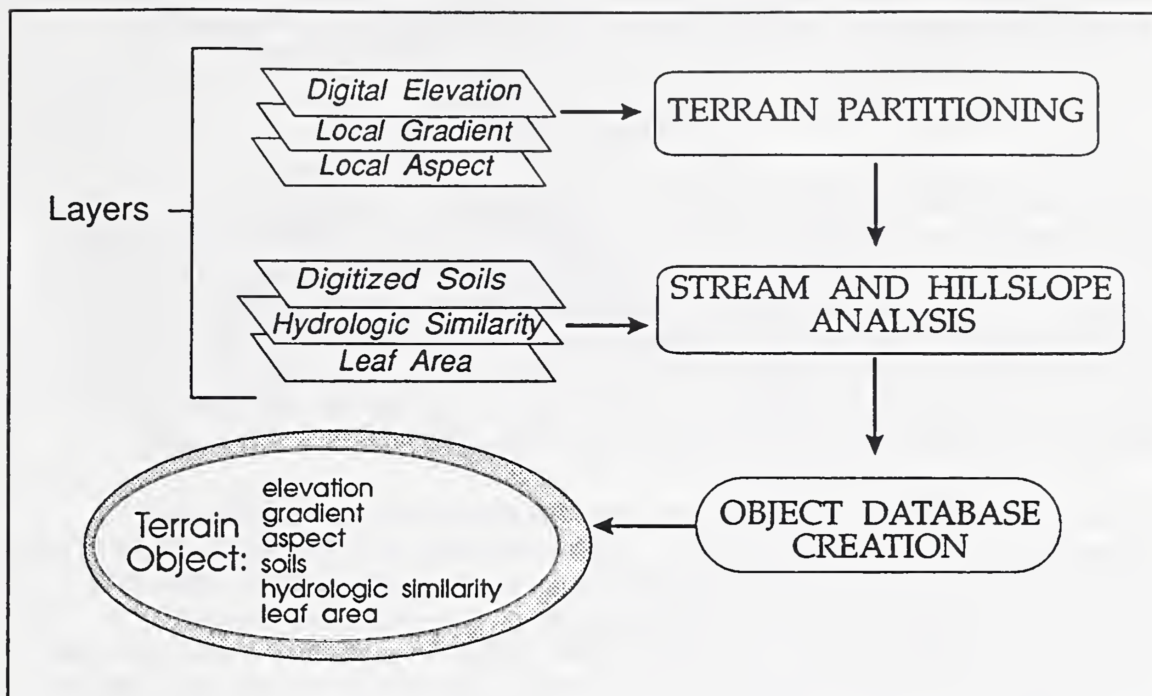


Figure 14. Concept of Layer-to-Object transformation process (source: Mackay *et al* 1993).

GIS. Another approach attempts to build an object-based database language as an interface layer between the user and the underlying nonobject-oriented (geographic) database system. This appears to be the more practical approach.

4 Transactions and GIS Environment

Although its importance is becoming increasingly apparent there is a relatively small body of work addressing issues related to managing GIS transactions (e.g. Halustchak 1993; Meier and Ilg 1986; Robinson and Zhang 1988; Robinson 1989). Two goals of a transaction module are to provide concurrency and maintain database consistency (Wang, 1991). These goals apply equally to nonspatial and spatial database management systems. In other words, management of transactions should be of ,at least, equal importance in GIS as in other database systems. However, like CAD, GIS transactions can be considerably more complex than the usual short duration transactions typical of many business applications.

4.1 Transactions

Basically, a transaction maps a consistent database state to a new consistent state atomically. It also serves as a unit of recovery (Wang, 1991). A database system must ensure that a database remains consistent despite many users doing retrievals and updates concurrently and the possibility of system failures.

A geographic database, like other databases, consists of a countable set of entities E = $\{x_1,x_2,...\}$ and a GIS transaction is a finite sequence of operations of the form

$$T_i=p_i^1[x_1]...p_i^{n_i}[x_{n_i}]c_i \qquad (1)$$

where x_i's are entities from E, where p^j_i is the jth operation of T_i which is either a read or write on entity x_j, and where c_i indicates that T_i commits. Should T_i fail to commit it will conduct a rollback. A consequence of the durability property of transactions is that committed transactions should never be rolled back. GIS transactions are rolledback when consistency contraints prevent an operation on the database that would result in a loss of integrity. The specification of those consistency constraints can be quite involved since the integrity of the geographic database depends on the logical consistency of a potentially very large number of geographic relations (Meier and Ilg 1986; Robinson 1989; Robinson and Zhang 1988).

A transaction system T = $\{T_1,...,T_N\}$ is a set of N transactions that are being executed concurrently. A complete schedule is the result of an execution of T. It is a sequence of all the operations from the transactions in T that respects the ordering within each T_i. These database transactions are typically of short duration, generally in terms of minutes. However, it is common for GIS transactions to be of long duration, lasting hours, days, or weeks. Meier and Ilg (1986) note an extreme case where, in some jurisdictions, modifications to a land register can last as long as a year due to legal procedures, during which time a user should not be able to gain exclusive use of the database for their updating work.

4.2 Long Duration Transactions

Like other spatial information systems, such as CAD, the GIS data management environment is characterized by a need to manage long-duration transactions (Korth et al 1990; Robinson 1989; Robinson and Zhang 1988) and versions (Katz 1990).

Users of GIS have a longer duration *transaction* than is typical in many business applications. Sani (1991) modeled the facility alteration process at an airport with regard to its implications on the management of airport technical data. The update of a technical database due to alterations to facilities can take weeks or months. A user who may be designing changes to a part of a geographic database may take hours or days to make all the necessary changes (for other examples see Halustchak 1993).

A **long duration transaction** (LDT) is a transaction which spans multiple short transactions, multiple sessions, and multiple application processes. Its lifetime does not have a system-imposed upper bound. A *session* is the duration of an application's interaction with the GIS (Figure 15). When a session is started, a short transaction is begun. A session takes place within a LDT. Therefore, when a session begins it can also begin a long transaction or connect to an existing one. And when a session ends it can end the LDT (Rotzell 1991).

Enterprise GIS make use of two basic models of managing access to the geographic databases on which a LDT may operate. The two models are the *check-in/check-out* and *branching versions*, both of which are not peculiar to GIS but have been described in other contexts.

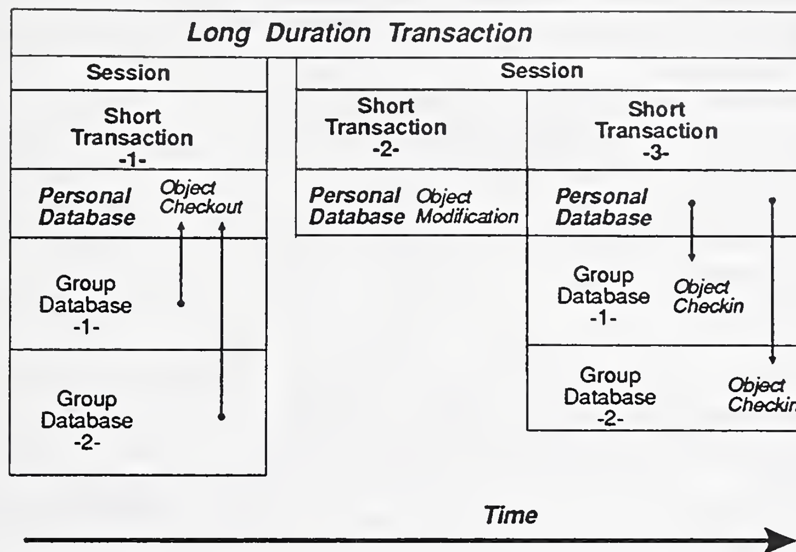


Figure 15. Concept of a Long Duration Transaction composed of sessions and short duration transactions (source: Rotzell 1991).

4.2.1 Check-in/Check-out Model. Conceptually, a LDT can be implemented by checking out data from a public database into a private database and merging the results back into the public database when the work is done (Figure 15). A LDT holds the locks in a public database on the objects, or layers, that are checked out to prevent their access by other transactions (Wang 1991). This is a model used by some GISs (*e.g.* Ashworth 1993).

4.2.2 Branching Versions Model. A branching versions model for LDT does not require holding both public and private databases. A LDT is modeled by a sequence of *regular short* transactions that operate on a private branch of versions of the database. Transactions accessing data of different branches of versions do not interfere with each other (Wang 1991).

4.3 GIS Transaction Environment

Many observations regarding the conduct of large GIS projects are similar to those made by Korth et al (1990) with regard to the conduct of large design projects in a CAD environment. The first observation is that a partitioning of a geographic database is induced by the partitioning of a large GIS application effort into a number of projects. Each partition consists of spatial and attribute data relevant to each project. GIS transactions from distinct projects require shared access to certain classes of data, such as the database directory. This means that long duration waits may be acceptable for cases where project transactions attempt to access

another project's database partition.

Secondly, a transaction hierarchy is induced naturally from the interaction of a user and the *window system* on the GIS workstation. The user may create and manipulate multiple windows, executing multiple tasks concurrently. The sequence of transactions initiated from the same window implies a user-defined ordering of transactions. This is only a partial ordering of transactions. For this reason, Korth et al (1990) model a user's transaction as a set rather than as a sequence of short-duration transactions.

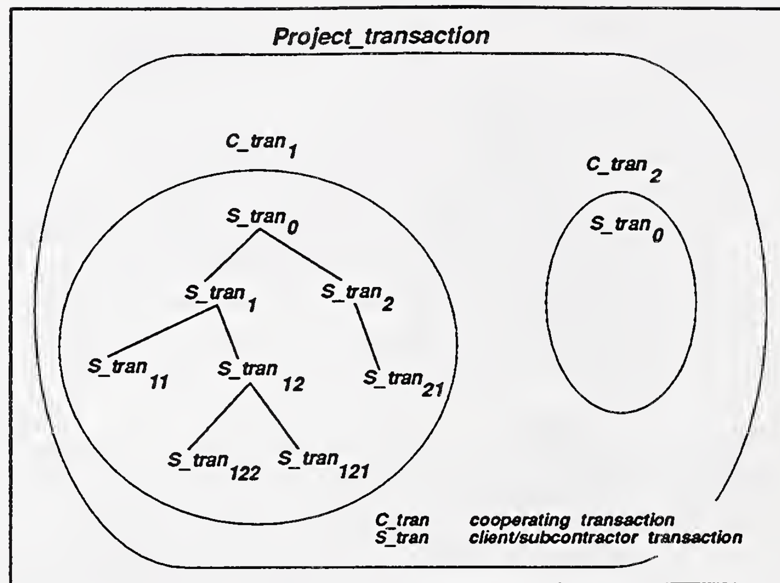


Figure 16. Project transaction made up of cooperating transactions which are directed acyclic graphs of short duration transactions (source: Korth et al 1990).

Each project has a number of users who further subdivide the project into subtasks. As users work on well-defined, fairly small subtasks, there is greater need for shared access to the project's database than among projects. This leads to the notion of *cooperating transactions* (Korth *et al* 1990) which is a set of user's transactions. Each user's short-duration transaction needs wait only until any currently-executing, conflicting short-duration transactions of other user's transactions are complete. From the database consistency point of view, it is immaterial how many users participate in the same cooperating transaction. All the short-duration transactions of all the users' transactions are treated as if issued by a single user.

In a complex GIS application project it is frequently the case that some tasks are subcontracted to other users. The client specifies the task to be completed and grants the subcontractor limited access to the client's database. Korth et al (1990) represent the notion of subcontracted work with the notion of a *client/subcontractor transaction* pair. A subcontractor transaction is a cooperating transaction which exists solely to work on behalf of a cooperating client transaction. Subcontractor transaction may itself subcontract work, thus becoming a client of one transaction while being a subcontractor of another transaction (Figure 16). Thus, similar to Korth et al 's (1990) conclusions regarding transactions in the CAD environment, the GIS transaction environment could be said to consist of :

- a set of concurrent project transactions where a project transaction is
    - a set of cooperating transactions, where a cooperating transaction is

- a hierarchy of clients/subcontractors, where a client/subcontractor transaction is
  - a directed acyclic graph of short-duration transactions, where a short-duration transaction is
    - a sequence of database operations, where a database operation is
      - a sequence of system operations,

which typically occur in a database hierarchy that supports the GIS application(s).

4.4 GIS Database Hierarchy

The system configuration often envisioned for enterprise GIS systems consists of at least one public system (central server) and a collection of private systems connected to the public system (*e.g.* Figure 2). The public system manages the public database of stable spatial and attribute data. A private system manages the private database of a user on a GIS workstation. A GIS transaction initiated on a private system involves

- checking out geographic data from the public system and their insertion into the private database
- checking in updated geographic data to the public database, and
- reading and writing of both the private database and the public database.

The model of transactions leads to a logical partitioning of a global database of a GIS system into a set of public, private, and project databases. The *public database* holds released geographic objects and data about their status. A released set of geographic objects, or layers, is accessible to all authorized users in the GIS environment but cannot be updated or deleted.

A *private database* exists on a workstation, or subnetwork of workstations, and contains non-released designs, on which a user is currently working. The user who creates the private database is the owner and administrator of the database. A private database of a user is generally not accessible to other users.

The *project database* serves as the repository of geographic objects, or layers, that are being passed back and forth among users within a project. It contains those geographic objects and data about those objects (or layers) that are accessible only to cooperating users within a project and specified subcontractor transactions.

Given this general model of GIS transactions and the database hiearchy to support those transactions, LDT management in the GIS environment is regularly taking place within a multidatabase environment. Therefore, multidatabase transactions are another characteristic of GIS application environment.

4.5 Multidatabase Transactions

A factor determining the level to which global transactions can be managed effectively is the degree of autonomy which exists. Distributed systems with the least autonomy offer the greatest global control (Ozsu and Valduriez, 1991). Since a distributed system has tight global control, distributed models of transaction serializability and atomicity are reasonable, and transactions are relatively short-lived. On the other hand, *open systems*, where transactions are based on bulk data exchange, atomicity is well-defined, but transactions may be long such as in the GIS environment. Consistency between systems exchanging data can be maintained by each respective system. However, it is the multidatabase systems which lie in the grey area between no autonomy and full autonomy that present the most difficult problems in transaction management.

The notion of transaction in a centralized DBMS is determined according to a model of serializability. Each transaction is considered atomic which means it either fully succeeds in retrieving information or updating the database, or it completely fails. Models of transactions are different in multidatabase systems, where notions of global and local control need to be defined. In a multidatabase system with a global schema, or a multidatabase language system with global query language, the user submits a global query which is decomposed into a set of subqueries each of which is passed to a local DBMS where it is processed. Each subquery is then considered an atomic transaction in itself. Queries may be translated many times as they travel through various system layers of the multidatabase. Thus, query processors must manage the global resources which may be distributed throughout the system. However, site autonomy requires that global control not include control of the data resources residing in the local DBMSs (Bright *et al*, 1992). A combination of global and local states of a global transaction, and cascading queries, makes transactions in multidatabase systems relatively long-lived and non-atomic.

Each autonomous DBMS in a multidatabase system has its own transaction processing services for managing, scheduling, and recovery of transactions. The multidatabase layer has its own transaction processing services which accept and coordinate global transactions. The global transaction manager is not aware of the local transactions being processed by the local DBMSs, so it cannot control local conflicts or conflicts between global transactions caused by interference of local transactions. It is therefore difficult to define and enforce distributed transaction atomicity (Ozsu and Valduriez, 1991).

5 Heterogeneous GIS

Given the nature of long duration transactions in a GIS environment it is clear that the management of multiple databases is an important issue in geographic information management. The GIS LDT which presumes public and private databases being held across multiple database systems has provided the impetus for groups to move towards development of spatial data interchange standards so that the interchange of the geographic data between client/subcontractor transactions and public/private databases can be accomplished in a more effective manner. On the surface this might seem to link spatial data interchange standards to LDT's in the GIS environment rather than database languages. However, data interchange standards address a relatively small portion of the problem. Mackay and Robinson (1992)

present the argument that geographic data interchange format efforts, represent attempts at heterogeneous geographic database design.

They further suggest that spatial data interchange standards efforts need to evolve towards multidatabases that provide facilities for querying, browsing, and updating geographically distributed spatial databases. Mackay and Robinson (1992) discuss the relationship of heterogeneous database design to geographic data interchange, geographic interchange standards in regard to the semantic modelling needs of GIS and the issue of whether or not dedicated geographic data interchange standards are necessary, or if their function can be provided using general database query language standards, with GIS extensions.

5.1 Data Interchange and Heterogeneous Database Design

Figure 17 illustrates database functionality as a continuum with interchange standards and remote DBMS interfaces laying at the extreme of having no database functionality. At the other extreme, federated DBMS provide full database functionality. A data interchange format allows periodic, nontransaction-based interchange of data among multiple database management systems (DBMS) (Sheth and Larson 1990). Thus, the notion of database functionality plays an important role in comparing data interchange standards to heterogeneous DBMSs, particularly with respect to transactions.

Data interchange is an important part of managing multiple DBMSs as these databases need conventions for data interchange. Litwin *et al* (1990) suggest the use of Standard Format Data Units (SFDU), self-documenting and self-describing data formats that carry universally interpretable type objects, as a possible solution to information interchange in a multidatabase. Standards such as SDTS and CGIS are to some extent interchange formats aimed at meeting this need in the GIS environment. An example of a self-describing data interchange carrier is the International Organization for Standardization *Information processing - Specification for a data descriptive file for information interchange* - ISO 8211 (ISO 1985) which is a medium-independent and system-independent file and data record format for interchanging of data structures of arbitrary complexity and size, between computer systems.

Database Functionality

exchange standards
remote DBMS interfaces

federated DBMS

Figure 17. Continuum of database functionality (source: Mackay and Robinson 1992).

5.2 Continuum of Heterogeneous Database Design Models

A distributed DBMS is a system to manage multiple geographically distributed databases as a single, integrated database. Distributed databases are typically designed top-down, such that local and global DBMS functions are designed simultaneously, and local DBMSs are homogeneous with respect to data model and functional interface (Bright *et al* 1992). Distributed DBMSs have the advantage of providing transparent access to the physically distributed database such that information can be accessed as if the DBMS was centralized (Ozsu and Valduriez 1991). Distributed databases have the disadvantage that each database in the distributed DBMS must be designed and built at the same time, or else existing databases have to be brought off-line while their data model is made compatible with the data model of the distributed DBMS installation. Existing applications are changed or disrupted because they do not distinguish between local and global operations, and the organization of information management has to be restructured in replacing a centralized DBMS with a distributed DBMS (Sheth and Larson 1990).

A bottom-up approach to DBMS design is achieved with multidatabase or federated systems. These systems have the advantage of being able to integrate pre-existing local DBMSs without having to modify them, and allow applications to be systematically designed such that different data can reside on dedicated databases (Litwin *et al* 1990). Since the pre-existing DBMSs were not designed with the multidatabase system in mind, they are most likely based on different data models, or different implementations of the same data model. As a result of this lack of homogeneity between DBMSs, the bottom-up designed system is referred to as a heterogeneous DBMS (Bright *et al* 1992). Heterogeneous systems can be classified in terms of the degree of autonomy permitted in each local DBMS. This is illustrated in Figure 18 which shows a continuum of DBMS designs ranging from tightly-coupled systems that have the least autonomy, to loosely-coupled systems that provide maximum autonomy. The intensity of the box in Figure 18 is intended to emphasize the fact that level of independence or autonomy between DBMSs is a relative property, measured on a continuum. At the extreme of least autonomy lies the global schema multidatabase systems which provide a single database view to the system user, and provide maximum DBMS support to the user. At the other extreme lies interoperable systems which provide no integrated DBMS support, but have full autonomy. The white box in Figure 18 emphasizes a division between multidatabase systems and interoperable systems. An interoperable systems' global functions are limited to simple data interchange; they support no database functionality (Bright *et al* 1992). Multidatabase language systems provide the highest level of autonomy in current multidatabase systems, but do so by limiting DBMS support to a set of support tools with which the user is expected to integrate information from multiple, autonomous DBMSs. Since it allows for a high degree of autonomy within existing databases, and provides facilities for retrieving and updating, multidatabase language systems represent the most reasonable model upon which to base geographic data interchange.

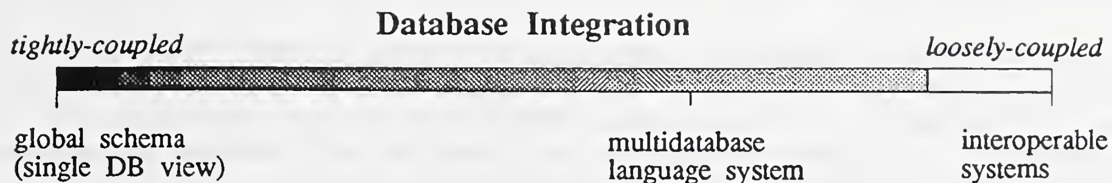


Figure 18. Level of autonomy in heterogeneous databases depends on coupling (source: Mackay and Robinson 1992).

5.3 Data Integration Problems

Heterogeneity in multidatabases results from the fact that autonomous databases are developed with a local homogeneous design. Sets of local assumptions clash and local applications do not have access to the semantics represented in foreign applications (Ventrone and Heiler 1991). Heterogeneity occurs in two forms: (1) syntactic, and (2) semantic. Syntactic heterogeneity results from differences in the data models and implementations of data models between two or more systems. Semantic heterogeneity results from differences in how the data models are used to represent the same real world in two or more systems, or as a result of evolution of a database schema. Differing data models, *i.e.* layer and object, are a common source of semantic heterogeneity in GIS (Worboys and Deen 1991).

Semantic heterogeneity may also occur as a result of having different occurrences of the same domain values having different interpretations. This condition violates the assumption in centralized relational databases that each value has a unique interpretation. Other sources of semantic heterogeneity include cardinality differences between domains represented in two or more databases, or in a given database over time, granularity differences, encoding differences, and time and unit differences (Ventrone and Heiler 1991). Data integration requires that syntactic and semantic differences be resolved using a common representation of the information, such as a global data model or an interchange format.

Schema integration and data interchange both require the use of a canonical, or globally accepted, data model. The goal is the integration of multiple, local databases each of which represents a small part of the overall world being modelled by the multidatabase system. This process consists of three schema manipulation procedures: (1) schema translation, (2) schema definition, and (3) schema integration. Schema translation is the process whereby the data definitions of the local schemes are translated into a component schema using a canonical data model. The local schema represented in one data model is mapped onto the component schema in a different data model. Translation is required when the data model of the local schema differs from the canonical model of the federation, or when the data model of the federated schema is different from the canonical data model. The translation process is facilitated by using a semantic data model as a canonical model. A semantic data model can represent additional semantics that may be difficult or impossible to specify in a traditional

model (Sheth and Larson 1990).

The purpose of schema definition is to create one or more export schemes from the component schema. An export schema describes the objects, using the canonical data model, that the local DBMS administrator decides to make available to the multidatabase community. Understanding the semantics of these objects is the responsibility of a federation user who may use these objects in generating his/her own federated database schema.

Schema integration is the process of designing a global conceptual schema, or federated schema, from which the logical schema for a new DBMS is created. During schema integration a number of conflicts may occur. Synonym naming conflicts occur when classes or attributes with different names represent the same concept or attribute, respectively. Homonym naming conflicts occur when names are the same but different concepts are represented. Structural conflicts result from different choices of modelling constructs between two or more schemes, and manifest themselves as type conflicts, dependency conflicts, where a group of concepts are related through different dependencies in different schemes, and key conflicts, where different keys are assigned to the same concept in different schemes (Gotthard *et al* 1992).

Schema transformations seek to resolve conflicts between schemes by making available to the system designer the semantic relationships of the concepts involved in the conflict. Schemes are merged by superimposing common concepts. Schemes are restructured to ensure completeness, eliminate redundancy and understandability. Completeness is achieved by introducing class hierarchies, additional relationships, or discriminating attributes (Gotthard *et al* 1992). By forming class generalization hierarchies, similar objects from different schemes can be classified together. Once objects are organized by their similarities it is easier to analyze the conflicts between the objects. Knowledge about the relationships between objects, other objects, and their attributes helps identify compatibility between objects, of different schemes, that are not identical. For example, isomorphic relationships between two types of different schemes occurs if their instances correspond (Yang *et al* 1991). Gotthard *et al* (1992) present a formal method of analyzing object types to identify semantic similarity.

Coordination of the activities of developing a multidatabase system requires the use of a data dictionary/directory. All schemes representing information about data managed in the federation are stored in the data dictionary/dictionary. The dictionary/directory also stores information about mappings among schemes, information about schemes and databases, and information about the various systems involved in the multidatabase (Sheth and Larson 1990). The data dictionary stores metadata to describe the semantics of the domain values. This metadata may include time of measurement, accuracy, source, and derivation formula, stored as text, program code, rules, constraint languages, tags, or footnotes. Other data in the database also provide context for a given item of data (Ventrone and Heiler 1991). The integrated data dictionary is the basis for information interchange in a multidatabase. The semantics of derived (or intensional) type objects should be kept in a derivation record in the system's data dictionary (Litwin *et al* 1990). Sani (1990) reports the only effort todate to apply

a data dictionary standard, namely IRDS, in a GIS context.

The data dictionary is used to manage update dependencies such that each system in the multidatabase checks derived objects and takes appropriate corrective actions on affected objects when a schema is updated. This may simply involve informing the system users of invalid intensional objects and make these objects unavailable until the database is corrected. Each export schema is thus verified before it is used in building a federated schema. Intensional definition of objects allows for the development and refinement of views without duplication; the data dictionary/directory can notify the system or user if a new view is being created that is equivalent to an existing view. This self-documentation and self-maintenance allows for a smoother and more natural evolution of autonomous databases (Litwin *et al* 1990).

6 Towards a GIS Database Language

There is a wide variety of languages developed for the query and management of GIS databases (Robinson 1991). In the case of dual database GIS there are typically two database languages with one for the spatial database which differs significantly from that used to manage the nonspatial database. Unified database GIS have a single database language but is may differ from those used in the remainder of the multidatabase GIS environment. It is important to realize that development of GIS database and query languages have generally ignored issues related to multidatabase management.

GIS database and query languages have progressed since the early days of automated cartography. In the early days, command-driven packages such as SYMAP demanded users to adhere to a rigid, fixed-format which was fairly close to the development language, e.g. Fortran. One of the early attempts at combining interactive microcomputer technology with a menu-driven geoprocessing system is described by Robinson (1980; Robinson and Coiner 1986). In this case, a programming language (CBASIC) was necessary for updating the geographic database. Command line languages are still common in GIS, particularly in layer-based systems.

Several efforts in spatial query language and geographic database management have a link with relational database systems. Among such efforts are pictorial query languages (PQL), query-by-pictorial-example (QPE), extensions to SQL, and natural language (NL) query processing systems. In addition, there has been some consideration of logic-based approached to geographic database management. The relational approach to pictorial, or geographical, database systems was proposed in the 1970's (e.g. Mantey and Carlson 1979). Chang and Kunii (1981) considered a GIS as an example of a relational database approach to pictorial database management. It is a common conclusion that the traditional relational algebra is insufficient to manipulate tabular, graphical, image, and geographical data (*e.g.* Chang and Fu 1980). For example, traditional relational algebra is inadequate for analyzing spatial relationships among geographic objects. Geographical data includes a typical case in which the number of different relations is almost equal to the number of data instances (Chang and

Kunii 1981).

6.1 Query by Pictorial Example (QPE)

In pictorial database systems a set of picture operations called picture algebra was designed for storage, retrieval, manipulation, and transformation of spatial data. Chang and Fu (1980; 1981) proposed QPE as a picture query language for IMAID an integrated relational database system interfaced with an image analysis system. Most queries are specified from the relations that are parts of a database of Landsat images and digitized maps. Processing sets are used to recognize boundaries of regions and connected line segments on digitized pictures. This QPE-based system included six classes of manipulation capabilities.

Another high level query language, PICQUERY, was designed to reside as a software layer above a pictorial database management system (PDBMS). It is intended to be the interface through which the user may access conventional relational databases using QBE and at the same time pictorial databases using PICQUERY. In concept, PICQUERY and QBE are seen as one single language by the user (Joseph and Cardenas 1988). PICQUERY is proposed by Joseph and Cardenas (1988) as a new tabular query language based on QPE. PICQUERY language commands may operate on the whole pictorial database or a set of picture-object identifiers. A picture is a distinctly identified, independent image stored in the pictorial database. It is built on top of a grid-based PICDMS.

6.2 Natural Language

It has been argued that since most users are competent in using natural language, it would be the ideal language for database interaction. The implication being that training would be reduced, or eliminated, and users would have little inhibition to using the computer (Schneiderman 1978). Thus, natural language query systems are an attempt to increase the *naturalness* of user interaction with GIS. Typical of some NL approaches in GIS-related applications is that described by Kasturi et al (1989) which is a natural language frontend (NLF) to a map data processing system that converts NL input to the syntactic form required by a query processor. The NLF consists of a preprocessor and a top-down parser. The preprocessor verifies that each word in the query is a part of its lexicon. The parser attempts to match its semantic grammar to the input query.

Pereira et al (1982) describe a system, ORBI, that is perhaps one of the more sophisticated NL efforts related to GIS. It is interrogatable in a subset of Portuguese which embodies and assimilates expert knowledge on environmental biophysical resource evaluation. ORBI's linguistic competence is achieved by means of a lexical and syntactic-semantic analysis, transforming a sentence into an optimally ordered directly evaluable list of Prolog goals (see Pereira and Warren 1980). The core grammar is independent of its application and is transportable to other domains. Much effort was devoted to incorporating elliptic and extraposed structures. Syntactic and semantic controls verify number and gender agreements,

designations of complex entities, and compatibility between nouns and verbs and their complements, pointing out any faults.

GIS users do not care to enter NL phrases via the keyboard. Hence there are significant research issues that could increase the utility of NL as a GIS query language. In particular, as technical capabilities for speech recognition as an input channel develop, voice input/output interface will have a profound impact (Straub and Wetherbe 1989) on how we interact with GIS. To support this interface, NLI will be required to have some increased measure of linguistic competence, especially in regards to the linguistics of spatial relations. NL for query in GIS require considerable GIS competence in order to interpret a user's query (Robinson 1991). Intelligent interfaces that permit users to express their needs in a form more natural to them will be critical in meeting the overall objective of providing GIS that have a high level of geographical competence while not burdening the user with undue cognitive load (Lundberg and Robinson 1988).

6.3 Logic-based Approaches

Mackay and Robinson (1992) raised the question of whether geographic data exchange formats (*e.g.* SDTS, CGIS) are required, or if spatial information management features can be supported using query languages based on powerful semantic models, or extended relational models (Robinson 1990). Roman (1990) suggests that logic-based data models provide the necessary tools to model space, time, and accuracy. In particular, databases supporting logical inference are able to handle queries on topologically connected spatial features, using transitive closure on spatial relations. Robinson and Zhang (1988) reported on the use of logic-programming to develop a system for managing consistency constraints during update transactions in a land information system.

Roman (1990) suggests that models based on logic are inefficient. This is particularly true of systems in which a Prolog inference engine is built on top of a relational database. The impedance mismatch that results from the tuple-oriented Prolog and set-oriented relational model leads to inefficiency. However, recent work on deductive database languages based on Datalog, such as Logical Database Language (LDL) (Naqvi and Tsur 1989), provide relatively efficient and powerful languages for complex queries. In an object-oriented framework, PROBE attempts to handle spatial information by providing specialized, embedded spatial operators, and a data structure for spatial access (Orenstein 1986; Orenstein and Manola 1988).

6.4 SQL

Widespread use of the SQL database language led to the ANSI (1986) standard. Prevalence of SQL as the database language of preference in organizations has had a profound effect upon how GIS products are being developed. There is a clear demand by organizations that GIS products be integrated with other software resources, particularly SQL-based database systems. Perhaps of more importance is the fact that organizations are beginning to consider GIS implementation to be an integral part of their existing corporate information resources

development strategy.

There have been many GIS-related developments using SQL or, SQL-like, languages (e.g., Abel and Smith 1986; Charlwood et al 1987; Frank 1982; Goh 1989; Herring et al 1989; Ingram and Phillips 1987; Keating et al 1987; Roussopoulos et al 1988; Tanaka and Ikeda 1989; Waugh and Healey 1986; Westwood and Brinkman 1988) which were reviewed by Robinson (1990). Extending SQL to deal with geographical queries has lead to the specification of classes of spatial operators. Terms used to denote spatial operators are similar across some of the above implementations. Robinson (1990) noted that there appears to be some convergence towards a common vocabulary that might serve as a useful basis for developing standards for defining spatial extensions to SQL. However, similarity of terms does not ensure that the operations are equivalent. One of the fundamental problems in determining equivalence is the heterogeneity in data models upon which the operators are based.

SDTS and CGIS (CGSB 1991) require the use of an internal data model to express spatial domains, and a second data model to specify the form of the data transfer. Unless these models are developed specifically to fit together, there are inevitable mismatches or heterogeneity between the models. The transfer specification could be simplified if the two models were made compatible. The CGIS specification suggests the use of an extended-SQL to represent the data for exchange purposes (CGSB 1991). SQL contains a full representation of relational algebra (*i.e.* it is Codd Complete). It is used to represent transactions involving both data retrieval and database updates. Future extensions to SQL for non-traditional applications such as GIS (Robinson 1990) need to consider aspects of extensibility to allow for spatial functions, abstract data types (ADT) for handling large, unformatted quantities of data, abstraction (*e.g. isa*) to preserve at least some of the semantics of concepts, and recursion or iteration to allow for transitive closure queries. Furthermore, if spatial data exchange is to evolve towards a more general framework of data sharing in multidatabase environments, then models of long transactions need to be incorporated into the database language. Key concerns of researchers and developers working on SQL/GIS issues include complex data types (*e.g.* arrays, lists, directed acyclic graphs), extensibility, spatial functions, and long transactions (Ashworth 1993; Halustchak 1993). Ashworth (1993) specifies a set of spatial functions (*e.g.* including EXTENT, BUFFER, LENGTH, OVERLAP) that may be included as part of a GIS/SQL. It seems unreasonable to pre-define all spatial relations that are considered significant by a one or a few vendors. However, if the expressiveness of SQL could be extended to that provided by logic languages, such as Datalog (Ullman 1988), then it may be possible to define spatial relationships using recursive expressions.

7 Concluding Comments

Transactions in a GIS environment often involve more than one database. The integration of GIS into an organization presumes, to some extent, the ability of the GIS and nonGIS database systems to effectively interact. It is the general surrounding IT environment which may in the end have more influence on GIS than GIS on IT. As SQL becomes more widely accepted in information systems in general and the performance of SQL-based GIS products improves it will be difficult for GIS users and suppliers to ignore the wider organizational imperatives that this portends.

Like Robinson (1991) it is suggested that an SQL database language with GIS extensions may provide a general and effective solution to the problems of interchanging and managing geographic data among several databases. Since SQL provides generalized query facilities for *ad hoc* queries on databases, it seems reasonable to expect that SQL, or another formal query language, is needed for building GIS that can be integrated with other information resources distributed about in other locations and systems as suggested by Gallagher (1993).

Bibliography

Abel D J and Smith J L (1986) A relational GIS database accommodating independent partitionings of the region. Proceedings of the Second International Symposium on Spatial Data Handling: 213-24

American National Standards Institute (1986) American National Standard for Information Systems - Database Language - SQL. ANSI X3.135-1986

Antonacci F, Bartolo L, Orco P and Spadavecchia N (1979) AQL: a relational database management system and its geographical applications. In: Blaser A (ed), Data Base Techniques for Pictorial Applications, Springer-Verlag: 569-99

Armstrong, M P and Densham P J (1990) Database organization for spatial decision support systems. International Journal of Geographic Information Systems 4: 3-20

Ashworth, M (1993) A geographic information systems perspective on spatial and object oriented extensions to SQL. In: Towards SQL Database Extensions for Geographic Information Systems. National Institute of Standards and Technology, Gaithersburg, Maryland

Ashworth, M (1992) System 9 data model and topology. presentation to seminar at University of Toronto, Toronto, Ontario

Baecker R M, William A and Buxton S (1987) Readings in Human-Computer Interaction: A Multidisciplinary Approach, Los Altos, California, Morgan Kaufmann

Band L E (1989) A terrain-based watershed information system. Hydrological Processes 3:151-62

Barrera R and Buchmann A (1981) Schema definition and query language for a geographical database system. In: Proceedings IEEE Conference on Computer Architecture for Pattern Analysis and Image Database Management: 250-56

Bracken I and Webster C (1989) Towards a typology of geographical information systems. Intern'l Jrnl Geog Info Sys 3(2): 137-152

CGSB (1992) Canadian Geomatics Interchange Standard: Formal Definition, 171-GP-1P. Canadian General Standards Board. Ottawa, Canada.

Chang N S and Fu K S (1981) Picture query languages for pictorial data-base systems. Computer 14 (11): 23-33

Chang N S and Fu K S (1980) Query-by-pictorial-example. IEEE Transactions on Software Engineering, SE-6(6): 519-24

Chang S-K and Kunii T L (1981) Pictorial data-base systems. Computer 14 (11): 13-17

Chang S-K, Reuss J and McCormick B H (1978) Design considerations of a pictorial database system. Policy Analysis and Information Systems, 1(2): 49-70

Charlwood G, Moon G and Tulip J (1987) Developing a DBMS for geographic information: a review. In: Proceedings Eighth International Symposium on Computer-Assisted Cartography, Baltimore, Maryland: 302-15

Chen, P P (1976) The entity-relationship model-toward a unified view of data. ACM Transactions on Database Systems, 1: 9-36.

Chock M (1981) Manipulating data structures in pictorial information systems. Computer 14 (11): 13-17

Codd E F (1970) A relational model of data for large shared data banks. Communications of Association of Computing Machinery, 13(6)

Date C J (1983) Database: A Primer. Addison-Wesley, Reading, Massachusetts

DMR (1989) Land Information Infrastructure, Volume 1, A Corporate Land Information Strategic Plan for the Government of British Columbia, DMR Group Inc.

Drabowski C E, Fong E N, and Yang D (1990) Object Database Management Systems: Concepts and Features. National Institute of Standards and Technology Special Publication 500-179, U S Government Printing Office, Washington, D.C.

Egenhofer M J (1992) Why not SQL!. International Journal of Geographic Information Systems 6(2): 71-85

ESRI (1992) ARC/INFO: GIS today and tomorrow. ESRI White Paper Series, Environmental Systems Research Instiute, Redlands, California

ESRI (1991) GIS Concepts Kit. Environmental Systems Research Instiute, Redlands, California

Foley J D and Wallace V L (1974) The art of natural graphic man-machine conversation. In: Proceedings IEEE 63, 4:462-71

Frank A U (1982) MAPQUERY: data base query language for retrieval of geometric data and their graphical representation. Computer Graphics 16 (3): 199-207

Gallagher L (1993) Database language SQL: integrator of data repositories for GIS applications. In: Towards SQL Database Extensions for Geographic Information Systems. National Institute of Standards and Technology, Gaithersburg, Maryland

GeoVision (1992) GeoVision Systems Incorporated Technical Background. GeoVision Systems Incorporated, Ottawa, Canada

Gogolla, M. and Hohenstein, U. (1991). Towards a semantic view of an extended entity-relationship model. ACM Transactions of Database Systems, 16, 369-416.

Goh P-C (1989) A graphic query language for cartographic and land information systems. International Journal of Geographic Information Systems 3(3): 245-55

Gotthard, W., Lockemann, P.C. and Neufeld, A. (1992). System-guided view integration for object-oriented databases. IEEE Transactions on Knowledge and Data Engineering, 4: 1-22.

Halustchak, O (1993) Proposed spatial data handling extensions to SQL. In: Towards SQL Database Extensions for Geographic Information Systems. National Institute of Standards and Technology, Gaithersburg, Maryland.

Hartson H R and Hix D (1989) Human-computer interface development: concepts and systems. ACM Computing Surveys, 21(1): 5-92

Herot C F (1980) Spatial management of data. ACM Transactions on Database Systems 5(4): 493-513

Herring J R, Larsen R C and Shivakumar J (1988) Extensions to the SQL language to support spatial analysis in a topological data base. In: Proceedings GIS/LIS'88 2: 741-50

Ingram K J and Phillips W W (1987) Geographic information processing using an SQL-based query language. Proceedings Eighth International Symposium on Computer-Assisted Cartography, Baltimore, Maryland: 326-35

ISO (1985) Information Processing - Specification for a Data Descriptive File for Information Interchange, ISO International Standard, ISO 8211-1985(E), International Organization for Standardization.

Jarke M and Vassiliou Y (1985) A framework for choosing a database query language. In: Mylopoulos J and Brodie M L (editors), Readings in Artificial Intelligence & Databases. Morgan Kaufmann, San Mateo, California: 363-376

Joseph T and Cardenas A F (1988) PICQUERY: a high level query language for pictorial database management. IEEE Transactions on Software Engineering 14 (5): 630-38

Katz R H (1990) Toward a unified framework for version modeling in engineering databases, ACM Computing Surveys, 22(4): 375-408

Katsuri R, Fernandez R, Amlani M L and Feng W (1989) Map data processing in geographic information systems. Computer 22 (12): 10-21

Keating T, Phillips W and Ingram K (1987) An integrated topologic database design for geographic information systems. Photogrammetric Engineering and Remote Sensing 53 (10): 1399-1402

Korth H F, Kim W, and Bancilhon F (1990) On long-duration CAD transactions. In: Readings in Object-Oriented Database Systems, Zdonik S B and Maier D (eds), Morgan Kaufmann Publishers, San Mateo, CA

Lafortune S (1988) Modeling and analysis of transaction execution in database systems, IEEE Transactions on Automatic Control, 33(5):439-447

Larson J A (1987) Chapter 2: user interfaces to database management systems. In: Larson J A (editor) Database Management Tutorial, IEEE Computer Society Press, Washington, D.C. 83-101

Lien W Y and Harris S K (1980) Structured implementation of an image query language. Pictorial Information Systems: 416-30

Lin B S and Chang S-K (1980) GRAIN-a pictorial database interface. In: Proceedings of IEEE Workshop on Picture Data Description and Management: 83-88

Litwin, W., Mark, L., and Roussopoulos, N. (1990). Interoperability of multiple autonomous databases. ACM Computing Surveys, 22, 265-293.

Lundberg G and Robinson V B (1988) Computers that know: a tutorial. Computers, Environment, and Urban Systems 12 (1): 49-72

Malone T (1982) Heuristics for designing enjoyable user interfaces: lessons learned from computer games. In: Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, Maryland: 63-68

Mackay, D S, Robinson V B, and Band L E (1993) On a knowledge-based approach to the management of geograhic information systems for simulation of forested ecosystems. In: Proceedings International Symposium on Environmental Information Management: Ecosystem to Global Scales, National Science Foumdation, Albuqurque, NM (in press)

Mackay, D S and Robinson V B (1992) Towards a Heterogeneous Information Systems Approach to Geographic Data Exchange. Discussion Paper 92/1, Institute for Land Information Management, University of Toronto, Mississauga, Ontario.

Manola F, Orenstein J, and Dayal U (1987) Geographic information processing in the PROBE database system. Proceedings Eighth International Symposium on Computer-Assisted Cartography, Baltimore, Maryland: 316-25

Mantey P E and Carlson E D Integrated geographic data bases. In: Blaser A (ed), Data Base Techniques for Pictorial Applications, Springer-Verlag: 173-98

McCann C A, Taylor M M and Touri M I (1988) ISIS: the interactive spatial information system. International Journal of Man-Machine Studies 28: 101-38

Menon, S and Smith T R (1989) A declarative spatial query processor for geographic information systems. Photogrammetric Engineering and Remote Sensing, 55(11): 1593-1600

Mohan L and Kashyap R L (1988) An object-oriented knowledge representation for spatial information. IEEE Transactions on Software Engineering 14(5): 675-680

Nagy G and Wagle S (1979) Geographic data processing. ACM Computing Surveys, 11(2): 139-81

Oberg R E (1988) The visual cartographic enquiry system for geodetic control data. GIS/LIS'88 Proceedings: Accessing the World. San Antonio, Texas: 484-89

Orenstein J A and Manola F A (1988) PROBE spatial data modeling and query processing in an image database application. IEEE Transactions on Software Engineering 14 (5): 611-29

Pereira L M P, Sabatier P and de Oliveira E (1982) ORBI-an expert system for environmental resource evaluation through natural language. Report FCT/DI-3/82, Departmento de Informatica, Universidade Nove de Lisboa (presented at First International Conference on Logic Programming, Marseilles, France)

Pizano A, Klinger A and Cardenas A (1989) Specification of spatial integrity constraints in pictorial databases. Computer 22 (12): 59-71

Reisner P (1981) Human factors studies of database query languages: a survey and assessment. ACM Computing Surveys 13(1): 13-31

Retz-Schmidt G (1988) Various views on spatial prepositions. AI Magazine 9 (2): 95-105

Robinson V B (1991) Spatial query languages, In: Muller J-C (ed) Advances in Cartography. Elsevier Applied Science on behalf of International Cartographic Association. p. 89-111

Robinson V B (1990) Structured query language (SQL) and geographic information systems (GIS): a briefing review, Research Agreement Report No. 1 (Draft), GIS Standards Laboratory, National Institute for Standards and Technology, United States Department of Commerce

Robinson V B (1989) Transaction recovery, referential integrity and the automated Canada lands information system(ACLIS), Working Paper 89/1, Institute for Land Information Management, University of Toronto

Robinson V B (1980) Urban Data Management Software (U.D.M.S.) Package - User's Manual (Draft), United Nations Centre for Human Settlements (HABITAT), Nairobi, Kenya

Robinson V B, Blaze M A and Thongs D (1985) Natural language in geographic data processing. Proceedings International Conference of the Remote Sensing Society and Center for Earth Resources Management, London. 67-73

Robinson V B and J C Coiner (1986) Characteristics and diffusion of a microcomputer geoprocessing software package: the urban data management software package. Computers, Environment, and Urban Systems, 10(3/4): 165-73

Robinson V B and Frank A U (1985) About different kinds of uncertainty in geographic information systems. Proceedings Seventh International Symposium on Computer-Assisted Cartography, Washington, D.C.: 440-50

Robinson V B and Frank A U (1987) On expert systems for geographic information systems. Photogrammetric Engineering and Remote Sensing 53 (10): 1435-41

Robinson, VB and Sani AP (1993) Modeling geographic information resources for airport technical data management using the information resources dictionary system (IRDS) standard. Computers, Environment and Urban Systems 17(2): in press.

Robinson V B and Wong R N (1987) Acquiring approximate representations of some spatial relations. In: Proceedings Eighth International Symposium on Computer-Assisted Cartography, Baltimore, Maryland: 604-22.

Robinson V B and Zhang G (1988) Modelling consistency preserving land database transactions in a logic programming environment. In: Proceedings GIS/LIS'88. San Antonio, Texas

Roman, G.C. (1990). Formal specification of geographic data processing requirements. IEEE Transactions on Knowledge and Data Engineering, 2, 370-380.

Rotzell, K. (1991) Transactions and versioning in an ODBMS.Computer Standards & Interfaces, 13(1): 243-248

Roussopoulos N, Faloutsos C and Sellis T (1988) An efficient pictorial database system for PSQL. IEEE Transactions on Software Engineering 14 (5): 639-50

Samet H (1990) QUILT: A geographic information system based on quad-trees. International Journal of Geographical Information Systems, 4(2): 103-132.

Samet J (editor) (1981) Query languages - a unified approach. Report of the British Computer Society Query Languages Group. Heyden University Press, Cambridge, England.

Shneiderman B (1978) Improving the human factors aspect of database interactions. ACM Transactions on Database Systems 3(4): 417-39

Small D W and Weldon L J (1983) An experimental comparison of natural and structured languages. Human Factors, 25(3): 253-63

Straub D W and Wetherbe J C (1989) Information technologies for the 1990's: an organizational impact perspective. Communications of the ACM, 32(11): 1328-39

Tanaka M and Ichikawa T (1988) A visual user interface for map information retrieval based on semantic significance. IEEE Transactions on Software Engineering 14 (5): 666-70

Tanaka S and Ikeda H (1989) An integrated data model for statistical and geographical data management. In: Kyung W L and Wu C (eds.), Proceedings of World Conference on Information Processing/Communication, June 14-16: 444-51

Tomlin C D (1990) Geographic Information Systems and Cartographic Modeling, Prentice-Hall, Englewood Cliffs, NJ.

Tuori M and Moon G (1984) A topographic map conceptual data model. In: Proceedings International Symposium on Spatial Data Handling, Zurich, 1: 28-37

USDI (1990) A Study of Land Information, United States Department of the Interior.

USDI (1989) Managing Our Land Information Resources, Bureau of Land Management, United States Department of the Interior.

van Roessel J W (1993 in press) Conceptual folding and unfolding of spatial data for spatial queries. Towards SQL Database Extensions for Geographic Information Systems. National Institute of Standards and Technology, Gaithersburg, Maryland.

Vassiliou Y and Jarke M (1984) Query languages - a taxonomy. In: Vassiliou Y (ed), Human Factors and Interactive Computer Systems, Ablex, Norwood, New Jersey: 47-81.

Ventrone, V. and Heiler, S. (1991). Semantic heterogeneity as a result of domain evolution. ACM SIGMOD Record, 20(4), 16-20.

Wang C C (1991) A strawman reference model in transaction processing for an object oriented database.Computer Standards & Interfaces, 13(1): 233-243.

Waugh T C and Healey R H (1986) The GEOVIEW design: a relational database approach to geographical data handling. In: Proceedings, Second International Symposium on Spatial Data Handling, Seattle, Washington: 193-212

Welty C and Stemple D W (1981) Human-factors comparison of a procedural and a non-procedural query language. ACM Transactions on Database Systems, 6(4): 626-49

Westwood K A and Brinkman J P (1988) Toward the successful integration of relational and quadtree structures in a geographic information system. In: Proceedings Urban and Regional Information Systems Association: 181-89

Worboys M F and Deen S M (1991) Semantic heterogeneity in distributed geographic databases, SIGMOD Record, 20(4):30-34.

Wu J-K, Chen T and Yang L (1989) QPF a versatile query language for a knowledge-based geographical information system. International Journal of Geographic Information Systems, 3(1): 51-59

Yang, J. Papazoglou, M.P. and Marinos,L. (1991) Knowledge-based schema analysis in a multi-database. In D. Karagiannis (ed.): Proceedings of the International Conference on Database and Expert Systems Applications, Springer-Verlag, Wien, Germany, p. 315-320.

Zloof M M (1975) Query-by-example. In: AFIPS Conference Proceedings, 44: 431-38.

Zobrist A L and Nagy G (1981) Pictorial information processing of Landsat data for geographic analysis. Computer, 14(11): 34-41

Vincent B. Robinson is Director of the Institute for Land Information Management at the University of Toronto. Over the past 15 years Dr. Robinson has conducted research and consulting projects for the National Science Foundation (USA), Natural Sciences and Engineering Research Council (Canada), United Nations Centre for Human Settlements, and many other government agencies in both the USA and Canada. He has served as a consultant to a variety of Canadian and international companies. Since 1988 he as been an Associate Professor of Surveying Science and on the graduate faculty of the University of Toronto. Currently he is involved in research on knowledegebased approaches to integrating GIS with ecological models and development of intelligent GIS transaction management tools. In 1978 he received his Ph.D. in Geography from Kent State University.

Dr. Robinson can be reached by phone at (416) 828-5459 and by electronic mail at the Internet addresses: vbr@geophagia.erin.utoronto.ca or vbr@esker.geog.utoronto.ca.

D. Scott Mackay is a PhD student in the Department of Civil Engineering and research assistant with the Institute for Land Information Management, at the University of Toronto. His dissertation topic considers the problem of managing spatial and temporal dynamics of ecosystem models integrated with spatial data management and geographical information systems. Other research interests includes knowledgebased classification of terrain, digital terrain analysis and development of machine vision tools for land/topographic databases. He has has been supported by the Natural Sciences and Engineering Research Council of Canada. Mr. Mackay has recently been awarded a Doctoral Research Fellowship under the Eco-Research Greenplan program coordinated by the Tri-Council Secretariat for interdisciplinary work related to management of ecosystem information for environmental management. Mr. Mackay holds BSc and MSc degrees from the Department of Geography, University of Toronto.

Mr. Mackay can be reached by electronic mail through Internet at the following address: scott@esker.geog.utoronto.ca .

Database Language SQL: Emerging Features for GIS Applications

Leonard Gallagher
National Institute of Standards and Technology
Information Systems Engineering Division
Technology Building, Room A266
Gaithersburg, MD 20899, USA
telephone: 301-975-3251
email: LGallagher@nist.gov

1 Introduction

Data management requirements for geographic information systems (GIS) often exceed the capabilities of existing database management systems. GIS requires a logically integrated database of diverse data often stored in geographically separated data banks under the management and control of heterogeneous data management systems. An over-riding requirement is that these various data managers be able to communicate with each other and provide shared access to data and data operations and methods under appropriate security, integrity, and access control mechanisms.

Database Language SQL [13] and its distributed processing counterpart, Remote Database Access (RDA) [15], are important International Standards that are able to address a significant portion of the above GIS requirements. SQL is particularly appropriate for the definition and management of data that is structured into repeated occurrences having common data structure definitions. SQL provides a high-level query and update language for set-at-a-time retrieval and update operations, as well as required database management functions for schema and view definition, integrity constraints, schema manipulation, and access control. SQL provides a data manipulation language that is mathematically sound and based on a first-order predicate calculus. SQL is self-describing in the sense that all schema information is queryable through a set of catalog tables. Features of the most recent SQL standard are discussed in [6], [7], and [18].

Early in 1991, technical committees for SQL standardization, operating under the procedures of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), committed to enhancing SQL into a computationally complete language for the definition and management of persistent, complex objects. This includes the specification of abstract data types, object identifiers, methods, inheritance, polymorphism, encapsulation, and all of the other facilities normally associated with object data management. Preliminary specifications for these facilities are contained in the most recent SQL3 Working Draft [14].

This paper focuses on the new object-oriented facilities proposed for inclusion in SQL3. It is adapted for GIS from [10]. Section 2 focuses on a three-schema data management architecture with an emphasis on using SQL and RDA to integrate local and remote data repositories, including both SQL and non-SQL data. Section 3 describes the preliminary status of object-oriented facilities proposed for SQL3; these facilities are subject to revision and improvement as they evolve over the next two or three years before final standardization. Section 4 looks at future opportunities and focuses on the potential benefits of defining a collection of standard "generic ADT packages" as the basis for SQL management of objects common to a number of application areas. Section 5 proposes a new SQL interface that would allow non-SQL data repositiories to make their data and special methods available, in a standard manner, to full-function SQL systems and to SQL applications. Section 6 draws some conclusions about the benefits of existing SQL and RDA standards as well as future opportunities.

2 Data management architecture

In an integrated GIS environment, we assume two or more remote GIS client/servers communicating with one another at the highest conceptual schema levels. This ensures that the communicating environments share a common understanding of the semantics of the data. This is a desirable goal that depends heavily on GIS standards not yet fully developed.

At the other extreme, in the absence of any data management standards, two sites can only communicate at the very lowest levels. Each site may be able to access files of data or "bulletin-board" views of data at other sites, or it may be able to pass parameters to application processes at remote sites, but it is not possible to access the external schema logical views of the data without knowing the external data model employed at the remote site and a syntax or protocol for invoking operations on that data model. Unless there are standards for schemas, access to remote sites may be effectively limited to the use of very low-level external schemas, thereby limiting common understanding to very simple structures. This low-level communication forces application programs to perform many of the data structuring and re-structuring tasks that could be performed more effectively by a database management system.

2.1 Three schema architecture

Data management has traditionally employed a three-schema architecture to place itself in a data processing environment. A conceptual schema represents a high-level, enterprise-wide view of all data, data relationships (including rules restricting updates or cascading the effects of updates to related data), and the GIS processes that use and update the data. Generally an enterprise has only one conceptual schema. The conceptual schema may or may not be directly implementable -- often it is represented via abstract diagrams devoid of the details necessary for actual implementation. Changes in the details of computer implementation or the specific human users and application programs that access the data have no effect on the conceptual schema.

An external schema represents a logical view of the data as accessible to a set of human users and application programs; an enterprise may have many external schemas. An external schema is generally a small subset of the conceptual schema, and may have application-oriented views of the data defined in the conceptual schema. An external schema is always

implementable, although minor changes in the physical implementation have no effect on its visibility to an application. This facilitates migration of the data to other hardware and software environments. An external schema may change to accommodate changes in the use of the data.

An internal schema represents a physical view of the data as stored on persistent storage devices. An enterprise may have many internal schemas to provide efficiency on a variety of hardware and software environments. Conceptual and external schemas are independent of the structures and access methods of any underlying file system; in contrast, an internal schema may be heavily dependent on file structures and access methods.

Each schema is constructed according to the rules of a data model. The data model prescribes not only the rules for defining data structures, but also the rules for interpreting and manipulating the data structures.

The conceptual schema may consist of a very large collection of object types and their interrelationships; no single application program wll require access to all the objects described by the conceptual schema. In contrast, an external schema may consist of a simple "record-oriented" view of only a few object types; a third generation programming language can easily process data described by such a schema. The conceptual schema itself may be so complex that it must be maintained by specialized software such as an Information Resource Dictionary System (IRDS). The IRDS may also be required to manage the mappings between the conceptual schema and the different external schemas, and the relationships among data and processes.

2.2 External schema communication

Standard communication among cooperating systems is possible at the present time using the government open systems interconnection profile (GOSIP). Currently, the application layer of GOSIP contains standards for association control (ACSE), file transfer (FTAM), virtual terminal (VT), and electronic mail message handling systems (MHS). The next version of GOSIP is expected in mid 1993 and will likely contain extensions of these facilities as well as remote database access (RDA) and additional facilities for handling documents (ODA/ODIF), electronic data interchange (EDI), and remote operations (ROS). Extensions to MHS and ROS should make it possible for user-defined objects at various remote sites to communicate their existence and provide access to their methods to application processors. Objects at remote sites may be able to "show themselves" to users at local workstations by using the graphical user interfaces proposed for future versions of VT.

We believe that the RDA component of GOSIP will provide the basis of distributed access to external schema definitions and to object data instances (Figure 1). In particular, existing and very near-term SQL and RDA standards will make it possible for various specialized, GIS data management systems to describe themselves to external processors and to provide "standard" access to the data they manage. With implementation of an External Repository Interface (ERI), discussed in Section 5 below, it is possible for each heterogeneous external schema to be "self-describing" in the sense that it can construct a "tabular" view of its logical data structures that can then be accessed and manipulated by all other sites. With longer-term

emerging data management standards that support object-oriented and knowledge-based features, an ERI interface can evolve into the desired high-level, GIS conceptual schema communication, with "seamless" integration of complex, structured data and supporting application services (Figure 1).

We begin with a "GIS Application Processor" that wishes to communicate with and access data at a number of different data repositories, some local and some remote. The application processor could use existing GOSIP protocols to connect to external processes or transfer files, but it would prefer not to have to manage its own communications links or worry about integrity, access control, remote transactions, or any number of different data manipulation functions; instead, it would rather communicate with a single, "familiar" GIS interface for both schema data and actual data occurrences. The "familiar" GIS could then connect itself to remote sites and access the desired data and data definitions, returning them to the accessing processor in a standard format. A remote object would still be able to use VT to "show itself" to the accessing process or use FTAM to transfer files containing objects or object definitions not under the control of the communicating data managers.

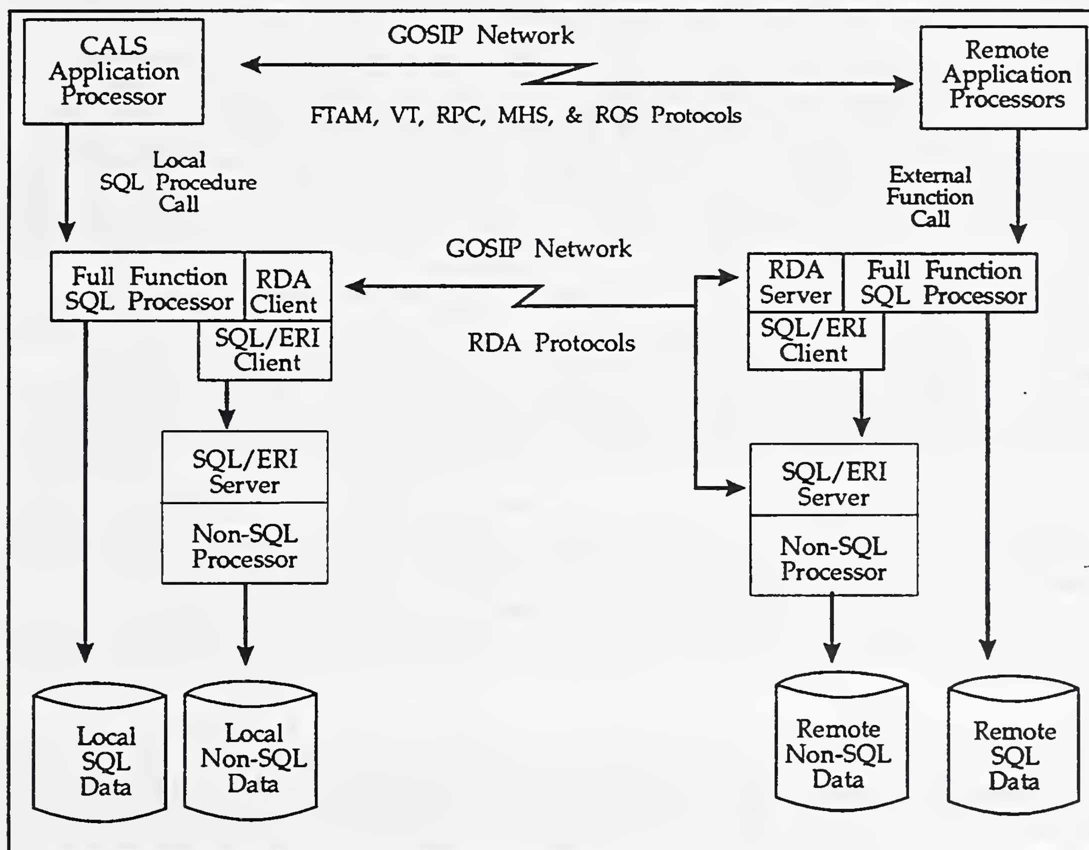


Figure 1. Data Integration Architecture.

We assume the existence of any number of heterogeneous data repositories, some at the local site and some at distributed sites. We assume a full-function SQL processor at all sites, but

not necessarily as the manager of the most important data. The non-SQL processors may control the maps, graphics images, or complex geographic structures that the application processor wishes to access. The local SQL processor conforms to Database Language SQL and has two integrated client components, one conforming to the RDA/SQL Specialization and one conforming to the SQL/ERI interface proposed in Section 5 of this report. Communications among the three SQL components are likely proprietary. The local site may have any number of non-SQL data repositories each controlled by a Non-SQL Processor having a component that conforms to the SQL/ERI interface. Communications among the internal components of the Non-SQL Processor are also proprietary. The local site has a proprietary local procedure calling mechanism and a proprietary local inter-process communications capability. Using these proprietary mechanisms the GIS Application Processor is able to issue standard SQL calls to the local full-function SQL processor, and the SQL/ERI Client component of the SQL processor is able to communicate, using an ERI specified subset of standard SQL, with the SQL/ERI Server of the Non-SQL Processor.

The local site is connected to one or more remote sites via a standard communications network that is able to connect to remote processes and allow "messages" or "calls" to be exchanged among processes (using ROS or MHS). Some messages may be sent directly from the application processor to processes or file stores at the remote site, but idealy, some local repository manager makes a connection and sends messages on behalf of the application processor. The Generic RDA and RDA/SQL Specialization standards specify protocols that allow the RDA Client component of the local SQL processor to send SQL statements to the RDA Server component of a remote SQL processor, or the SQL/ERI Server component of any Non-SQL processor, and receive data in return. All protocols and data are defined in the RDA standards and are transmitted as ASN.1 (ISO 8825) packages. If the GIS Application Processor is operating, interactively, on behalf of a human user, then any of the data repositories may use a local graphical user interface (GUI), or non-local GOSIP VT protocols, to present status information or a "menu of choices" to the human user. In this way an interactive "browsing" or "navigational" capability is provided to the human user without losing the standard RDA/SQL protocol communications used by the non-human processors.

At the remote site there exists a full-function SQL Processor as well as any number of Non-SQL Processors. Components of the SQL Processor conform to the SQL and RDA standards, and satisfy the proposed SQL/ERI Client requirements. Each Non-SQL Processor has a component that conforms to the SQL/ERI Server specification. The remote site handles internal communications and procedure calls in the same proprietary manner as does the local site.

At the present time the RDA standard specifies interchange protocols for transmitting records of data from a server site to a client site, provided that the data items in the records are either numbers or character strings. Near term RDA follow-on specifications will extend the data types handled to all of those specified in the SQL-92 specification, i.e. fixed and variable length character strings, fixed and variable length bit strings, fixed and floating point numerics, dates, times, timestamps, and intervals. Later RDA follow-on specifications will provide interchange mechanisms, in terms of ASN.1 elements, for the user defined abstract data types (ADTs) specified in the emerging SQL3 working draft. RDA protocols do not by themselves provide interchange mechanisms for other data objects, so interchange standards for images,

maps, topologies, and other spatial and geographic objects will remain critical for transmitting agreed object definitions among various sites.

SQL and RDA provide the basis for standard external schema communication. An SQL external repository interface (SQL/ERI) makes it possible for non-SQL data repositories to share their data with GIS applications. With emerging SQL enhancements for object-oriented and knowledge-based data management and emerging RDA extensions for distributed database, the ERI can evolve to support the GIS goal of direct conceptual schema communication.

3 Object Management in SQL3

Earlier versions of the SQL3 Working Draft specified object-oriented facilities such as User-Defined data types, Assertions, and Triggers to support the object notions of abstraction and encapsulation. They also included the generalization and specialization of tables in a table hierarchy that supported multiple inheritance. In each case, these object concepts were only partially supported without fully enforcing them as a discipline. There was no notion of object identifiers and no notion of class hierarchies for user-defined data types.

The newest SQL3 Working Draft [14] now addresses the requirement for true "objects" and "object identifiers" in SQL and also specifies supporting features such as encapsulation, subtypes, inheritance, and polymorphism. The following subsections briefly describe the current status of these features. Of course, as with any draft, the specifications are subject to revision and improvement as they evolve over the next two or three years before final standardization.

3.1 Abstract data types

The abstract data type facility provides the capability to define and manage persistent data type definitions, including structures and operations on those structures. A new ADT can be "constructed" from any existing data type, including previously defined abstract data types, known to the current SQL environment.

An abbreviated version of the current syntax is as follows:

```
CREATE TYPE <ADT name>
        [ <OID options> ]
        [ <subtype clause> ]
        [ <member list> ]
```

The <OID options> are discussed in Section 3.2 and the <subtype clause> is discussed in Section 3.5 below. The <member list> specifies the attributes, operations, and other methods applicable to the ADT definition.

Attributes contain an <encapsulation level>; otherwise, an attribute definition has the same syntax as a regular relational column definition. An <encapsulation level> is specified as either PUBLIC, PRIVATE, or PROTECTED. Public components form the interface of the ADT and are visible to all authorized users of the ADT. Private components are totally

encapsulated, and are visible only within the definition of the ADT that contains them. Protected components are partially encapsulated, being visible both within their own ADT and within the definitions of all subtypes of that ADT.

Operations on an ADT may include EQUALS and LESS THAN. The EQUALS clause identifies a function that specifies conditions under which two instances of the defined data type are considered to be equal. The LESS THAN clause identifies a function that specifies a comparison ordering over data type instances. The definitions of equality and ordering, taken together, specify the semantics to be used in the SQL comparison predicate when applied to ADTs. If the LESS THAN definition is not specified, then the ordering tests of the comparison predicate must return an unknown value.

Other methods associated with ADTs include FUNCTION definitions. FUNCTIONs operate on one or more ADT instances and return either Boolean, if the result is to be used as a truth value in a Boolean predicate, or a single value of a defined data type, if the result is to be used as a value specification. Functions may be SQL functions, completely defined in an SQL schema definition, or they may be external function calls to functions defined in standard programming languages (see 3.4).

Special "constructor" and "destructor" functions are defined to make or remove instances of an abstract data type. At the present time constructor and destructor functions are invoked implicitly through Insert or Delete operations on a table.

Special methods, identified as CAST functions, specify how an ADT may be mapped to other existing data types. For example, an IMAGE ADT may be mapped to a BIT STRING representation. With the ability to include CAST specifications in any ADT definition, a data type definer can define mappings to specific external representations. In this way the internal representation may be kept PRIVATE and not directly accessible, thereby allowing efficient implementation.

Abstract data type definitions and SQL or external function declarations are treated just like any other SQL objects as far as access control and other usage privileges are concerned. Access control is independent of encapsulation since encapsulation defines the structure of what is possible to see and access control determines who can see that structure. All names are qualified by the schema name of the containing schema and by the catalog name of the containing catalog. Each such object is "owned" by the authorization identifier of the schema in which it is defined and all privileges to use the object, to "see" its representation, or to modify its definition must be explicitly granted by the object owner. Privileges on existing ADTs may be GRANTed and REVOKEd and new ADTs or function declarations may be ADDed or DROPed from the schema as part of the SQL schema manipulation language.

3.2 Object identifiers

Object identity is that aspect of an object that never changes and that distinguishes the object from all other objects. Ideally, an object's identity is independent of its name, structure, or location. Object identity is therefore a unique identification of an object that is independent of the state of that object, and which persists over time. The identity of an object persists even after the object no longer exists (e.g. like a timestamp), so that it may never be confused with the identity of any other object. Other objects can use the identity of an object as a unique way of referencing it.

The <OID options> clause in an ADT definition allows several alternatives for object identifier (OID) specification:

> WITH OID VISIBLE,
> WITH OID NOT VISIBLE, or
> WITHOUT OID

If WITH OID is specified, then an OID value is generated when the object is created to give that object an immutable identity. The OID can be referenced by an "object reference" in constraints, queries, and other ADT definitions. If NOT VISIBLE is specified, then the OID value may not be passed as a parameter to functions or stored in a host language variable. If WITHOUT OID is specified, then the ADT does not have an object identifier; instead, each instance represents itself just like values of primitive data types do.

There is a continuing debate in the SQL standardization committees as to whether SQL should support all three of the above options, or if <u>every</u> new ADT definition should be assumed to carry a unique object identifier. The outcome of this debate will not affect the functionality of the new language, but it may influence its appearance and style.

PUBLIC components of an ADT are accessible to authorized users through a special "attribute reference" operator (i.e. <ADT reference>.<attribute name>). If the ADT has WITH OID specified, then the <ADT reference> will be to an object identifier of a specific ADT; if the ADT has WITHOUT OID specified, then the <ADT reference> will be to the ADT value itself. The attribute reference identifies a specific component of the ADT instance and permits the user to read or modify its value.

3.3 Object management

Object ADTs are subject to special "constructor" and "destructor" functions that either create a new instance of the ADT and make it part of the database or remove an instance of an ADT from the database. Since SQL is a "table-based" language, SQL designers have to address issues concerning whether or not SQL object instances may exist outside of table occurrences. If SQL objects are allowed to exist outside of tables, then new syntax to manipulate them and new structures to hold collections of them must become part of the language. Although these issues are still subject to debate and modification, the current status is to require that all object manipulation be achieved through table operations.

SQL3 currently allows specification of a "tabular" shell over an ADT class. In this way, constructor and destructor functions are automatically invoked when rows are inserted or deleted from the table, and the table itself is the collection of all object occurrences. SQL query and update statements may then be applied to the table without the need for any special language enhancements. If a requirement surfaces later to allow objects to exist and be managed independently of tables, then it can be handled as an upward compatible language enhancement.

In order to accommodate this implicit invocation of constructor and destructor functions, minor enhancements are needed to the syntax and semantics of the CREATE TABLE and INSERT statements (see [5]). The statement "CREATE TABLE T OF <ADT name>" creates a tabular envelope around the abstract data type specified by <ADT name> and all attributes of the abstract data type become columns of the new table T. Object instances may then be managed by the usual SQL Select, Update, and Delete statements.

The statement "INSERT INTO T <insert spec> ALIAS <name>", using the new ALIAS option with an Insert statement, returns the object identifier of any new object created by the Insert statement to the <name> variable. Syntax rules prohibit specification of an alias when the underlying ADT is specified WITHOUT OID.

3.4 Methods and functions

An abstract data type includes not only a collection of values or properties but also a set of operations (methods) on those values. Such operations are the procedures and functions that define the behavior of the abstract data type. Some of the operations associated with an ADT might be realized by means of data that is stored in the database, while other operations might be realized as executable code (functions). An implementation of an ADT is the stored data together with the data structures and code that implement the behavior of the ADT.

As seen in Section 3.1, methods may be encapsulated with the ADT definition. Specific methods for determining equality, and ordering when appropriate, are then usable in regular SQL comparison predicates. As we have seen, other methods can be defined as special operators on ADTs or as predicates that return truth values. A single value returned from a function call can be used any place in the SQL language that a single value is allowed. A truth value returned from a function call can be used as one of the terms in a boolean predicate.

Functions may be defined completely in SQL, or only their interface definition may be specified in SQL with the content of the function written in some programming language (e.g. Ada, C, or eventually C++). An SQL function may be "defined" as an independent schema element, as part of an ADT definition, or as part of a module definition. An external function may be "declared" in the same places. The syntax of an SQL function is:

```
[CONSTRUCTOR | ACTOR | DESTRUCTOR] FUNCTION <function name>
<parameter declaration list>
RETURNS <data type>
<SQL statement> ;
END FUNCTION
```

Only constructor or destructor functions may create or destroy new ADT instances; they have already been discussed above. An actor function is any other function that reads or updates components of an ADT instance or accesses any other parameter declared in the <parameter declaration list>. A parameter in the parameter list consists of a parameter name and an SQL data type. The RETURNS clause specifies the SQL data type of the result returned. Since all data types in an SQL function are SQL types that must accommodate Null values, it is not necessary to worry about "indicator" parameters to convey Nulls.

The <SQL statement> may be any SQL statement, including compound statements and control statements. Of particular importance here are the following:

- A NEW statement that allows creation of a new OBJECT ADT instance; it is only allowed in a CONSTRUCTOR function.
- A DESTROY statement that destroys the existence of an OBJECT ADT instance; it is only allowed in a DESTRUCTOR function.
- An ASSIGNMENT statement that allows the result of an SQL value expression to be assigned to a free standing local variable, a column, or an attribute of an ADT.
- A CALL statement that allows invocation of an SQL procedure.
- A RETURN statement that allows the result of an SQL value expression to be returned as the RETURNS value of the SQL function.

The syntax of an external function declaration is:

```
DECLARE EXTERNAL <external function name>
    <formal parameter list>
    RETURNS <result data type>
        [ CAST AS <cast data type> ]
    LANGUAGE <language name>
```

The <formal parameter list> is a list of SQL data types. If a data type in the parameter list is supported in the programming language identified by the LANGUAGE clause, then the corresponding programming language routine has two parameters for that data type; the second parameter is the "indicator" parameter to convey Null values. If a data type in the parameter list is an ADT not supported in the programming language identified by the LANGUAGE clause, then the corresponding programming language routine has two parameters for each base type in the ADT definition, recursively. Again, the second parameter in each case is an "indicator" parameter. The actual mapping from the <formal parameter list> in the external function declaration to the parameter list of the programming language routine can become quite complex, but is completely specified in the SQL3 working draft.

The CAST AS clause is a convenience to allow "encapsulated" casting from a programming language data type to an SQL data type. For example, an SQL DATETIME data type that appears in the formal parameter list is automatically cast to its character string literal

representation before it is passed to a programming language routine. The CAST AS clause could also automatically cause the character string RESULT to be re-cast into an SQL DATETIME value. Since every SQL data type has a defined CAST operation to and from character string representations, it is possible to pass any SQL data type to any programming language that supports character strings.

3.5 Subtypes and inheritance

Specification of "UNDER <ADT name>" in the <subtype clause> of an ADT definition (see 3.1) permits a new ADT to be defined as a subtype of an existing ADT. A type can have more than one subtype and more than one supertype. Thus a subtype is a specialized type of one or more supertypes and a supertype is a generalized type of one or more subtypes. A supertype shall not have itself as a proper subtype and a subtype family shall have exactly one maximal supertype.

Inheritance is an abstraction mechanism that adds to the power of data abstraction by allowing classes of objects to be related hierarchically. Inheritance allows classes to share definitions with other classes, thereby supporting newer, more specialized, data definitions without losing the existing properties and operations of the superclass.

Through inheritance, new types can be built over older, less specialized types rather than having to rewrite properties from scratch. Inheritance makes it possible to build a hierarchy of related ADTs, i.e. a "type hierarchy", that share the same interface, and possibly the same representation and implementations. As we move up in the inheritance hierarchy, types become more generalized; as we move down types become more specialized. These generalization/specialization capabilities allow more accurate and succinct modeling of applications.

The SQL implementation of a type hierarchy (see [16]) requires that an instance of a subtype is also an instance of all of its supertypes. Every instance is associated with a "most specific type" that corresponds to the lowest subtype assigned to the instance. At any given time, an instance must have exactly one most specific type. Note that the most specific type of an instance need not correspond to a leaf type in the type hierarchy.

As above, every column definition in an ADT has an encapsulation level specified as either PUBLIC, PRIVATE, or PROTECTED. Public and protected components are visible to the definitions of all subtypes of that ADT, but private components are not.

A subtype can define constructor, actor, and destructor operations just like any other ADT. All operations of the supertype are invocable from the subtype, so there is a high potential for name conflicts when the subtype defines more specialized operations. Name resolution rules, described in the following section, ameliorate this problem.

SQL provides "multiple inheritance", i.e. a subtype can have more than one direct supertype. With multiple inheritance, we can define a new type STUDENT-EMP which is a subtype of both STUDENT and EMPLOYEE. A person who is both a student and an employee can be modeled as an instance of the STUDENT-EMP type. In this way an instance will satisfy the

requirement to always have a "most specific type".

Multiple inheritance could lead to ambiguous inheritance of components from its supertypes, so SQL provides the disambiguity rules:

- If an attribute in more than one supertype is inherited from a common supertype higher in the hierarchy, then only the one from the common supertype is inherited.

- If an attribute with the same name in each of the supertypes is not inherited from a common supertype higher in the hierarchy, then the type definition is invalid unless the type definer renames the inherited components to remove the name clash.

These rules, and other related issues, are subject to improvement and evolution as the SQL ADT facility stabilizes over the next two or three years.

3.6 Polymorphic functions

Polymorphism is the ability to invoke an operation on any of several different objects and have that object determine what to do at run-time. A polymorphic function is one that can be applied in the same way to a variety of data objects. Support for polymorphism involves technical decisions concerning early or late binding among objects and the procedures that invoke their methods. To help address some of these technical decisions, a number of techniques have evolved, such as:

Overloading	The ability to assign the same name to more than one function or procedure -- name resolution is then determined by a set of rules, thereby allowing a processor to distinguish among different functions of the same name by examining the "type" of the input data.
Coercion	The ability to omit semantically needed type coversions -- we have already seen that SQL uses this technique in some of its parameter passing to external functions.
Inclusion	The ability to manipulate objects of a subtype "as if" they were objects of a supertype -- possibly with a function of the same name that calls different routines.
Generalizing	The ability to specify that a parameter should "take on" the type of some supertype during processing of a specific function call.

Resolution rules supporting polymorphism are derived from one basic concept, i.e. that for any particular function invocation, there must exist a single "best match" from the candidate functions that are "in scope". When a function call is executed, the unique "most specific

type" of the various input parameters is used to help define the "best match" according to the following rules, many of which are derived from those used by C++:

- Begin with the set of all functions that are "in scope" for a particular function call, i.e. those that are defined or declared with the calling function name in the statement, procedure, module, or schema associated with the function call.

- For each argument, determine the set of functions that is a "best match" for that argument, then take the intersection of these sets. Unless this intersection has exactly one member, the call is illegal. That is, the function selected must be a "strictly better match" for at least one argument than every other possible function, but not necessarily the same argument for each function.

- To decide which functions are the best match for each argument, agree: an "exact match" is better than one based on type coercion (i.e. CASTing), an implicit conversion to the "closest" supertype is better than SQL or user-defined type coercion, and an implicit SQL-defined CAST is better than an implicit user-defined CAST.

Consider an example where the following three functions are defined:

```
FUNCTION F(:p1 X, :p2 INTEGER)
FUNCTION F(:p1 X, :p2 REAL)
FUNCTION F(:p1 Y, :p2 REAL)
```

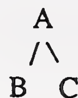
Suppose that X and Y are abstract types defined in the same schema and that X has a user-defined CAST clause that defines a conversion from INTEGER to X. A function call F(1,1), where both 1's are integer literals would result in the following analysis:

- For the first argument, there is no exact match, and no implicit SQL conversion, so the user-defined conversion from INTEGER to X is used. The first two function definitions are in the set of "best matches".

- For the second argument, the INTEGER type of the literal is an exact match to the INTEGER parameter. Only the first function is in the set of "best matches".

Based on this analysis, the first function is the "best match" so it is the one invoked.

As a second example (also from [3]), suppose you have a type hierarchy in which A is a supertype, B and C are subtypes of A, and D is a subtype of both B and C.



\/
D

In this case, suppose that three functions F are defined as:

F(:p A) RETURNS A
F(:p B) RETURNS B
F(:p C) RETURNS C

Based only on the compile time rules described so far, a function call F(d), where the value d is of type D, would be ambiguous because, with an implicit conversion to the supertypes B and C of D, both the second and third functions would be in the set of "best matches". A function call F(x), where the value x is of type A, would also be ambiguous at compile-time because the x might really be of type D at run-time.

As currently specified, the SQL Syntax Rules require that an implementation consider all possible cases that might occur at run-time. For each case, i.e. for each of the four possible "most specific types" that x might have at run-time in the above example, the above rules must hold and identify for each case, a unique function. In addition, the set of identified functions must specify a RETURNS data type such that they all share a "common" supertype. This common supertype is the one returned in all cases.

There are a number of issues associated with polymorphism. In some cases the rules for resolving function calls are arbitrary and not always the best choice for every application scenario. Other issues concern the run-time overhead associated with the "late" binding required to support inheritance of properties to all subtypes of a given type.

3.7 Control structures

In Section 3.4 we saw that several "control" statements have already been introduced into the SQL language, e.g. ASSIGNMENT, CALL, and RETURN. The next step is to consider if more control statements and other "programming language" facilities should be added to SQL. In particular, we need to consider the appropriateness of:

- sequences of SQL statements in a procedure instead of the single SQL statement allowed in the current SQL standard,
- flow of control statements, such as looping, branching, etc.,
- exception handling, so that when an exception is raised, the SQL function or procedure can resolve the issue internally, or propagate the exception to the next outermost exception handler, rather than always returning control to the main calling routine.

These 3GL programming language facilities are valuable because they allow procedural encapsulation and they allow complete behavior to be specified within an ADT definition without the need to escape to a procedure written in some other language. Complex behavior can be made available to the host application program via a single call. This offers benefits in

both cost and control. In SQL'92, all procedures are single SQL statements so multiple calls must be made to address complex problems. All temporary state and flow of control belong to the host language application, thereby adding complexity to the application program that could be encapsulated in the SQL procedure. The following facilities from [8] are included in the current SQL3 working draft.

3.7.1 Compound statement. A compound statement is a statement that allows a collection of SQL statements to be grouped together into a "block". A compound statement may declare its own local variables and specify exception handling for an exception that occurs during execution of any statement in the group. Its syntax is as follows:

```
[ <beginning label>: ]
[ <variable declaration list> ]
BEGIN
        [ <SQL statement list> ]
        [ <exception handler> ]
END [ <ending label> ]

<exception handler> ::=
        EXCEPTION [ {WHEN <condition> THEN <SQL statement list>}... ]

<condition> ::= <exception name list> | OTHER
```

An <exception name> is unique within a <module> and may be declared with an <exception declaration>.

3.7.2 Exception handling. An exception declaration establishes a one-to-one correspondence between an SQLSTATE error condition and a user-defined exception name. It's syntax is

```
DECLARE <exception name> EXCEPTION FOR SQLSTATE <SQLSTATE
literal>.
```

The exception handling mechanism under consideration for SQL3 is based very strongly on the mechanism defined in Ada. Each compound statement is assumed to have an exception handler; if one is not explicitly defined, then a default handler is provided by the system. When the execution of a statement results in an active exception condition, then the containing exception handler is immediately given control. Ultimately, the exception handler terminates with one of the following behaviors:

- The compound statement terminates with the active exception condition still active, or
- The compound statement terminates with a new active exception condition, or
- The compound statement terminates successfully, as though no exception occurred, and there is no outstanding active exception condition.

The exception handler may execute SIGNAL or RESIGNAL statements to identify a new exception name or to pass on the existing exception name. If an exception condition occurs in the exception handler itself, then the compound statement is terminated and that exception condition becomes the "active" exception condition.

3.7.3 Flow of control statements. Program flow of control statements is currently specified in the draft SQL3 document as

- A CASE statement to allow selection of an execution path based on alternative choices. A <value expression> is executed and, depending on the result, control is transferred to the appropriate block of statements.

- An IF statement with THEN, ELSE, and ELSEIF alternatives to allow selection of an execution path based on the truth value of one or more conditions.

- A LOOP statement, with a WHILE clause, to allow repeated execution of a block of SQL statements based on the continued true result of the <search condition> in the WHILE clause. A LOOP statement is also allowed to have a statement label.

- A LEAVE statement to provide a graceful exit from a block or loop statement.

3.8 Stored procedures

In the existing SQL-92 standard a module is a persistent object created by the Module Language. It is a named package of procedures that can be called from an application program, where each procedure consists of exactly one SQL statement. However, there is no requirement that an implementation be able to execute Module Language (the alternative is Embedded SQL) and the resulting persistent module is not stored as part of the SQL schema, is not reflected in the information schema tables, and cannot be passed across an RDA connection to a remote site.

In the emerging SQL3 specification, ANSI and ISO standardization committees have recognized the requirement for some "standard" capability to define persistent modules that "live" in the SQL schema and whose procedures may be called from any SQL statement in the same processing environment. In SQL3 the CREATE MODULE statement has the same status as any other schema definition statement. The result of execution is a module that is managed by SQL rather than by the proprietary facilities of the host operating environment. Module definitions are reflected in the Information Schema just like any other schema object and they are subject to ownership and access control declarations.

The primary benefit of supporting stored procedures is that implementations are able to optimize groups of statements rather than just individual statements. Entire packages of SQL procedures can be sent to a remote SQL conforming site, be optimized at that site, and then be executed with a single call when needed.

3.9 Parameterized types

The ability to define abstract data types does not by itself provide the capability to define "parameterized" types. A parameterized type is really a "type family" with a new data type for each value of an input parameter. For example, an ADT definition for VECTOR(N) can be thought of as a family of data types, one for each positive integer value of N. This idea is not new as we already have parameterized, predefined types in the existing SQL standard, e.g. CHARACTER(N) and DECIMAL(P,S), and it is a common feature in programming languages that support user-defined types. Reference [4] adds the ability to specify parameterized ADT definitions in SQL.

We may think of a "parameter" as any value of a data type known to the SQL environment, e.g. an integer value in the examples above. We may also think of a "parameter" as a reference to an exisitng data type, rather than a value of that type. For example, we may wish to specify that VECTOR(N) is really a vector of integers, or reals, or decimals with fixed precision and scale. This can be achieved by passing a data type name to the ADT definition instead of just a data type value.

The syntax for specifying a parameterized type in SQL3 is very similar to that for specifying a regular ADT, namely:

```
CREATE TYPE TEMPLATE <template name>
      ( { <template parameter declaration> }... )
      <abstract data type body>

<template parameter declaration> ::=
      <template parameter name> { <data type> | TYPE }
```

The keyword TEMPLATE indicates that the specification is for a paramaterized ADT rather than a regular ADT. The keyword TYPE indicates that a parameter is a data type name rather than a data type value. The <abstract data type body> is analogous to the body of a regular ADT definition.

A parameterized type is referenced by specifying the type template name and an actual parameter list. Each actual parameter must be a value, or a data type, that can be determined at syntax evaluation time, i.e. usually a literal or a data type name. If the actual parameter is a data type name, then the formal template parameter must specify TYPE.

You are allowed to define more than one type template with the same name, just as you may define more than one <SQL function> with the same name. For example, it is legal to define two POINT data types, one for 2-dimensional points and one for 3-dimensional points. The rules for matching a parameterized type reference to a parameterized type definition are the same as the rules for matching overloaded functions.

<u>3.9.1 Distinct types</u>. Sometimes it is desirable to be able to distinguish between table or ADT attributes that have the same underlying ADT definition. For example, table T1 might have a column named Cartesian_Coordinate that is defined to have the data type POINT and table T2 might have a column named Polar_Coordinate that is also defined to have the data

type POINT. The POINT data type may have a DISTANCE function defined to calculate the distance between any two points, but clearly the calculation

DISTANCE(T1.Cartesian_Coordinate,T2.Polar_Coordinate)

may not be a meaningful calculation.

The SQL3 draft provides a facility for the user to declare that two otherwise equivalent ADT declarations are to be treated as "distinct" data types. The keyword DISTINCT used in an ADT declaration indicates that the resulting type is to be treated as "distinct" from any other declaration of the same ADT. In the above example, if two new types are declared

```
CREATE DISTINCT TYPE CARTESIAN_POINT AS POINT
CREATE DISTINCT TYPE POLAR_POINT AS POINT
```

and if Cartesian_Coordinate and Polar_Coordinate are declared to have the data types CARTESIAN_POINT and POLAR_POINT respectively, then both kinds of coordinant points would have all the methods and operations for POINT, but any attempt to apply the DISTANCE function between them would result in an error.

The DISTINCT facility in SQL3 is currently only applicable to abstract data types, not to pre-defined data types. For example, it is not legal to declare the following:

```
CREATE DISTINCT TYPE PART_NBR AS INTEGER
CREATE DISTINCT TYPE EMP_ID AS INTEGER
```

It is possible to extend the definition for "distinct" types from abstract types to pre-defined types. This is an issue that will be addressed in the near term.

3.9.2 CLID generator types. The emerging Common Language Independent Datatypes (CLID) specification [12], under development in ISO JTC1/SC22/WG11, also specifies facilities for parameterized and distinct data types. However the syntax is slightly different. CLID uses the keyword GENERATOR instead of TEMPLATE and NEW instead of DISTINCT, but the effect is essentially identical.

Some of the following sections in this paper were written before parameterized types were added to the SQL3 working draft, so they are written using the CLID syntax. Thus in the following sections

CREATE GENERATOR TYPE ⇔ CREATE TYPE TEMPLATE and
CREATE NewType = NEW OldType ⇔ CREATE DISTINCT TYPE NewType AS OldType

3.10 Generator types

At the present time SQL3 only defines a limited number of data types, including: fixed and variable length character strings, fixed and variable length bit strings, fixed and floating point numerics, dates, times, timestamps, intervals, Boolean, and enumerations. The components of

an ADT must therefore be defined as one of these base data types or as a previously defined ADT.

There is a need to extend the pre-defined, base data types to include "generator" data types such as those specified in the emerging ISO common language-independent datatypes (CLID) specification [12]. The CLID specification includes, among others, the generator types:

ARRAY {[<lower>..<upper>]}... OF <base type>
LIST OF <base type>
SET OF <base type>
CHOICE ({<identifier>:<base type>}...)
RECORD ({<identifier>:<base type>}...)
RANGE Subtype Generator
SIZE Subtype Generator
EXTEND Supertype Generator
Declared Generator

ARRAY creates a new data type whose values are fixed-length sequences of values from the <base type>. Values in the sequence are in a one-to-one correspondence with a value in the product space of the <lower> to <upper> limits for each index component. Operations defined for an array are:

Equal — a Boolean predicate on two arrays that returns true if their corresponding components are pairwise equal, and returns false otherwise.

Select — operates on an array and on an element of the index product space to return the appropriate value from the <base type>.

Replace — operates on an array, an element of the index product space, and a value from the <base type> to produce a new array with the given value substituted into the appropriate position.

LIST creates a new data type whose values are ordered sequences of values from the <base type>, including the empty sequence. The following operations are defined for lists:

Equal — Equal is a Boolean predicate on two lists that returns true iff the two lists have the same length and all components are pairwise equal.

IsEmpty — IsEmpty is a Boolean predicate on a single list that returns true iff the sequence is empty.

Head — Head operates on a list to return the first element from the sequence.

Tail — Tail operates on a list to return a new list consisting of all elements except the first.

Append — Append operates on a list and a single value from the <base type> to produce a new list with the value as the last element of the sequence.

Empty — Empty is a niladic operation yielding the empty sequence.

SET creates a new data type whose values are taken from the power set (i.e. the set of all subsets) of the <base type>, with operations appropriate to the mathematical set algebra. In order to ensure uniqueness of representation, the <base type> is required to be discrete, meaning it cannot have a distance function defined that yields any limit points that are elements of the <base type>. Operations on sets consist of the following:

Equal	a Boolean predicate that returns true iff two sets are equal.
IsIn	a Boolean predicate that operates on an element of the <base type> and a set to return true iff the element is a member of the set.
Subset	a Boolean predicate that operates on two sets and returns true iff the first is a subset of the second.
Union	operates on two sets to return their set union.
Intersection	operates on two sets to return their set intersection.
Complement	operates on a set to return its set complement.
SetOf	operates on a single value of the <base type> to return the singleton set consisting of just that element.
Empty	a niladic operation that returns the empty set.
Universe	a niladic operation that returns the set of all values from the <base type>.
Select	operates on a set to return an arbitrary single value from that set.

SET OF <base type> : SIZE (<min>, <max>)
LIST OF <base type> : SIZE (<min>, <max>)

For sets, the size declaration specifies constraints on the minimum and maximum cardinality of sets that are allowed as part of the subtype. For lists, the size declaration specifies constraints on the minimum and maximum length of the sequence of elements that determines the list.

EXTEND is a supertype generator that acts on a <base type> to create a new data type that has the <base type> as a subtype. The syntax is:

<base type> : EXTEND (<value list>).

The value space of the extended data type consists of all values in the <base type> plus those additional values specified in the <value list>. The extend generator can be used with ENUMERATION or other data types to extend the value space to include the values in the value list. The operations defined for the <base type> are not automatically extended.

4 Generic ADT packages

With the existence of abstract data types (ADTs) and generator types in SQL3, there is a new opportunity to consider standardization of the SQL interface to packages of high-level data objects for use in various application areas. It makes sense to standardize packages for science and engineering, full-text and document processing, or methods for the management of multimedia objects such as image, sound, animation, music, and video. A new standardization project has been proposed by ISO/IEC JTC1/SC21/WG3 to develop a "companion standard" for SQL3 that would specify a class library of multimedia and other useful ADT packages. This proposed standard, to be called SQL/MM, would provide an SQL language binding for multimedia objects defined by other standardization bodies (e.g. SC18 for documents, SC24 for images, and SC29 for photographs and motion pictures). The main intent of standardization is to allow applications to use the same ADTs across different application areas, thereby promoting interoperability and the sharing of data, and encouraging performance optimization over a manageable collection of types.

Some packages, e.g. geographic information structures, have broad appeal across different application areas and could benefit from "generic" standardization. The difficult part, and the most important, is for user groups to agree on the desired types and methods that are most useful in a specific application area. Once an application package is well-defined and accepted by a significant user population, implementations will follow rapidly.

This section is gives examples of a few generic ADT packages that may have value in some GIS application areas. If others agree that it makes sense to define named ADTs and syntax for accessing them in a standard specification, then they can be pursued with the proper mechanism to make that standardization happen.

The capabilities discussed in Section 3 need to be available before one can define robust "generic" application packages with "well-chosen" definitions. Since Abstract Data Type facilities are already in the emerging SQL3 specification, we assume that the needed additional data types and data type generators will either also be included as part of SQL3 or as the "base requirement" of the SQL/MM project.

n the following subsections we assume that all SQL-defined types are able to accommodate null values. All operations must be able to handle input and output parameters that may have null values. In some cases we will want the aggregate type (e.g. an array) to handle null types in the components and sometimes we won't. This will be a decision that needs to be made for each newly defined type. To accommodate null values, we must sometimes replace the CLID BOOLEAN data type by an SQL data type that recognizes true, false, and unknown. We achieve this by using the CLID data type STATE(true, false, unknown) wherever a truth value is expected in SQL.

4.1 Vector spaces

We know from linear algebra [11, 17] that any finite-dimensional vector space defined over the field of real or complex numbers is isomorphic to a real or complex product space. In addition, if a finite-dimensional vector space has an inner product defined, then that inner

product space is isomorphic to the real or complex product space with the well-known scalar product for vectors of real or complex numbers. Many geometric properties of any inner product space are closely related to geometric properties on real or complex product spaces. For these reasons, the finite-dimensionsal real and complex product spaces play an important practical role in many geoscience and engineering applications. It would be an important contribution to these applications if the base operations were standardized and incorporated into all "scientific" database management systems.

We assume the existence of a COMPLEX data type

TYPE COMPLEX = RECORD (real:REAL, imag:REAL)

with operations, using the notation of [12], defined as follows:

Zero():COMPLEX
One():COMPLEX
Add(u:COMPLEX,v:COMPLEX):COMPLEX
AddInv(u:COMPLEX):COMPLEX
Mult(u:COMPLEX,v:COMPLEX):COMPLEX
MultInv(u:COMPLEX):COMPLEX where u not equal to ZERO()
Conjugate(u:COMPLEX):COMPLEX

where Zero and One are the additive and multiplicative identities, Add and Mult are the addition and multiplication operations for complex numbers, AddInv is the additive inverse, and MultInv is the multiplicative inverse. Under these operations the non-null values of COMPLEX satisfy the axioms of a mathematical field.

The following additional COMPLEX operations are defined in terms of the base operations for Record-type:

Equal(u:COMPLEX,v:COMPLEX):STATE(true, false, unknown)
Aggregate(x:REAL,y:REAL):COMPLEX Note: x or y null implies result is null
RealPart(u:COMPLEX):REAL
ImagPart(u:COMPLEX):REAL

4.2 Product Spaces

Let FIELD be any numeric data type that has the operations of a mathematical field, i.e. addition, subtraction, multiplication, and division. We define the following general product space to represent vectors defined over FIELD:

CREATE GENERATOR TYPE VECTOR(N) OF FIELD = NEW ARRAY [1..N] OF FIELD

We are particularly interested in the cases where FIELD is either REAL or COMPLEX, so define the following:

CREATE GENERATOR TYPE REALVECTOR(N) = VECTOR(N) OF REAL

CREATE GENERATOR TYPE COMPLEXVECTOR(N) = VECTOR(N) OF COMPLEX

Let VECTOR(N) be a shorthand notation for VECTOR(N) OF FIELD and define the following operations for VECTOR(N):

```
Zero():VECTOR(N)
Add(x:VECTOR(N),y:VECTOR(N)):VECTOR(N)
AddInv(x:VECTOR(N)):VECTOR(N)
ScalarMult(a:FIELD,x:VECTOR(N)):VECTOR(N)
ScalarProd(x:VECTOR(N),y:VECTOR(N)):FIELD
Norm(x:VECTOR(N)):REAL
Dist(x:VECTOR(N),y:VECTOR(N)):REAL
```

With Zero as the zero vector in N-space, Add as vector addition, AddInv as the additive inverse for vectors, ScalarMult as multiplication of a vector by a scalar, and ScalarProd as the scalar product of two vectors, VECTOR(N) OF FIELD will satisfy the axioms of an inner product space over the base field. In addition, if Norm(x) is defined to be the positive square root of ScalarProd(x,x), and if Dist(x,y) is defined as Norm(Add(x,AddInv(y))), then VECTOR(N) OF FIELD becomes a complete metric space in the norm topology.

The following operations, analogous to Record-type operations, allow equality comparison of vectors, construction of vectors from base components, and reference to individual vector components:

```
Equal(x:VECTOR(N),y:VECTOR(N)):STATE(true, false, unknown)
Aggregate(a1:FIELD,...,aN:FIELD):VECTOR(N)
Project(x:VECTOR(N),i:(INTEGER:RANGE(1..N))):FIELD
```

There is no LESS THAN operation defined for vectors, so it is not possible to test for order relationships in comparison predicates.

4.3 Cross Products in Real 2-Space and 3-Space

We note that REALVECTOR(2) and REALVECTOR(3) have important vector applications in many engineering and physics problems. In each of these data types, it makes sense to define an additional operation, the vector cross product, as follows:

```
CrossProd(x:REALVECTOR(2),y:REALVECTOR(2)):REALVECTOR(3)
CrossProd(x:REALVECTOR(3),y:REALVECTOR(3)):REALVECTOR(3)
```

We could consider generalizing the CrossProd to be a vector product on VECTOR(N), but the definition becomes very complex. It might be better to wait until after matrix algebra operations have been defined to represent linear transformations of vectors. One might also consider adding other vector operations, e.g. Gradients and Curl, etc., that are important in physical science applications.

4.4 Matrix algebra

Many practical applications can be modeled in terms of linear transformations on finite-dimensional real or complex vector spaces. Such linear transformations can be represented as two-dimensional matrices over real or complex fields. Addition and scalar multiplication of matrices are defined to coincide with addition and scalar multiplication of linear transformations, and multiplication of matrices is defined to coincide with composition of linear transformations. Standardization of matrices and matrix operations as SQL data types would be an important contribution to many areas of applied mathematics.

Let RING be any numeric data type that has the operations of a mathematical ring, i.e. addition, subtraction, and multiplication, but not necessarily division. We define the following general product space to represent matrices defined over RING:

```
CREATE GENERATOR TYPE MATRIX(M,N) OF RING = NEW ARRAY
[1..M][1..N] OF RING
```

We are particularly interested in the cases where RING is either REAL or COMPLEX, so define the following:

```
CREATE GENERATOR TYPE REALMATRIX(M,N) = MATRIX(M,N) OF
REAL

CREATE GENERATOR TYPE COMPLEXMATRIX(M,N) = MATRIX(M,N)
OF COMPLEX
```

Let VECTOR(N) be a shorthand notation for VECTOR(N) OF FIELD and let MATRIX(M,N) be a shorthand notation for MATRIX(M,N) of RING. Then define the following operations for MATRIX(M,N):

```
Zero():MATRIX(M,N)
Identity():MATRIX(M,N)
Add(x:MATRIX(M,N),y:MATRIX(M,N)):MATRIX(M,N)
AddInv(x:MATRIX(M,N)):MATRIX(M,N)
ScalarMult(a:RING,x:MATRIX(M,N)):MATRIX(M,N)
MatrixProd(x:MATRIX(M,S),y:MATRIX(S,N)):MATRIX(M,N)
Transpose(x:MATRIX(M,N)):MATRIX(N,M)
ConjugateTranspose(x:MATRIX(M,N)):MATRIX(N,M)
Trace(x:MATRIX(N,N)):RING
Determinant(x:MATRIX(N,N)):FIELD
Adjoint(x:MATRIX(N,N)):MATRIX(N,N)
MatrixInv(x:MATRIX(N,N)):MATRIX(N,N)
Rank(x:MATRIX(M,N)):(INTEGER:RANGE(1..Min(M,N)))
Reduce(x:MATRIX(M,N)):MATRIX(M,N)
```

With Zero as the all-zero matrix, Add as matrix addition, AddInv as the additive inverse for matrices, and ScalarMult as multiplication of a matrix by a scalar, MATRIX(M,N) satisfies the axioms of a vector space over the base field. In addition, with Identity as the diagonal identity matrix, MatrixProd as the product of two product-compatible matrices, and MatrixInv as the

inverse of square, non-singular matrices, REALMATRIX(N,N) and COMPLEXMATRIX(N,N) are linear algebras, i.e. they satisfy the properties of a non-commutative mathematical ring with identity.

The operations Transpose, Conjugate, Trace, Determinant, and Adjoint (see [11]) all provide helpful tools when matrices are used to analyze linear transformations. The ConjugateTranspose is intended only for COMPLEXMATRIX(M,N), but it is well-defined for all MATRIX(M,N). Rank and Reduce are important operations whenever an MxN-Matrix is used to represent a system of M linear equations in N variables.

The following operations, analogous to Record-type operations, allow equality comparison of matrices, construction of matrices from base components, and reference to individual matrix components:

```
Equal(x:MATRIX(M,N),y:MATRIX(M,N)):STATE(true, false, unknown)
RowAggregate(x1:VECTOR(N),...,xM:VECTOR(N)):MATRIX(M,N)
ColumnAggregate(x1:VECTOR(M),...,xN:VECTOR(M)):MATRIX(M,N)
Project(x:MATRIX(M,N),(i,j):(RECORD
    (rowid:(INTEGER:RANGE(1..M)),colid:(INTEGER:RANGE(1..N))))):RIN
    G
ProjectRow(x:MATRIX(M,N),i:(INTEGER:RANGE(1..M)):VECTOR(N)
ProjectColumn(x:MATRIX(M,N),j:(INTEGER:RANGE(1..N)):VECTOR(M)
```

4.5 Euclidean geometry

Any instantiation of the values of REALVECTOR(N) is said to be a Euclidean space. Every Euclidean space can support the structures and operations of Euclidean geometry. Such structures include, but are not limited to, point, line, segment, polysegment, polygon, convex polygon, angle, angular measure, distance from point to line, distance from point to polygon, distance between two disjoint polygons, planes, hyperplanes, etc. Such notions are important in GIS as the foundations upon which spatial data structures can be defined. The notion of convex hull and optimization of linear functions defined on convex sets is important in operations research and applied economics. It would be an important contribution to these fields if certain fundamental Euclidean structures were standardized in SQL.

```
CREATE GENERATOR TYPE EUCLIDEAN(N) UNDER REALVECTOR(N)
```

All of the vector operations from REALVECTOR(N) would be inherited by EUCLIDEAN(N) and new structures and operations are defined as follows:

```
CREATE TYPE POINT UNDER EUCLIDEAN(N)

CREATE TYPE LINE = NEW SET OF POINT : SIZE(2,2)

CREATE TYPE HYPERPLANE = NEW ARRAY [1..N+1] OF REAL

CREATE TYPE HALFSPACE = NEW ARRAY [1..N+1] OF REAL

CREATE TYPE SEGMENT = NEW RECORD(start:POINT,stop:POINT)
```

```
CREATE TYPE POLYSEGMENT = NEW LIST OF POINT

CREATE TYPE CONVEX_POLYHEDRON = NEW SET OF HALFSPACE :
SIZE(1,M)
```

Note: M represents the number of "faces" of the convex polyhedron.

Define CAST operations so that a convex polyhedron can be represented as a set of M linear inequalities in N variables, i.e. $AX <= B$ where A is an MxN matrix and X and B are elements of VECTOR(M). Also be able to CAST a conve polyhedron as an Mx(N+1) matrix so that reduction algorithms can be applied to "solve" the inequalities.

```
CREATE TYPE LINEAR_FUNCTION = NEW VECTOR(N) OF REAL
```

Note: A linear function is of the form $f(X) = AX$ where A is a 1xN vector and X is an Nx1 vector variable.

The following operations might be the start of a long list of operations of interest to operations research professionsals:

```
Distance(p:POINT,l:LINE):REAL
Distance(p:POINT,h:HYPERPLANE):REAL
Maximum(f:LINEAR_FUNCTION,cp:CONVEX_POLYHEDRON):REAL
Minimum(f:LINEAR_FUNCTION,cp:CONVEX_POLYHEDRON):REAL
etc.
```

Of particular importance in many engineering and cartographic applications is the two-dimensional Euclidean plane and the three_dimensional Euclidean space. These instantiations of EUCLIDEAN(2) and EUCLIDEAN(3) deserve special notation for they are likely candidates for optimization in database systems catering to these application areas.

```
CREATE TYPE EUCLIDEAN_PLANE = EUCLIDEAN(2)

CREATE TYPE EUCLIDEAN_SPACE = EUCLIDEAN(3)
```

In the EUCLIDEAN_PLANE, it also makes sense to define the following new structures and operations:

```
CREATE DOMAIN POLYGON AS POLYSEGMENT
      CHECK(Equal(Head(VALUE),Last(VALUE))

CREATE DOMAIN CONVEX_POLYGON AS POLYGON
      CHECK( ScalarProd of all successive points is ...)
```

[Continue with needed functions -- especially those dealing with distances, conic sections, surveying, navigating, mapmaking, etc.] For example see [19].

4.6 Geographic regions on Earth surface

Regions on the surface of the Earth could be defined as part of a "geographic" package as follows:

```
CREATE TYPE DEGREE = NEW INTEGER:RANGE(-360..360)
CREATE TYPE LONG_DEGREE = DEGREE:RANGE(-179..180)
CREATE TYPE LAT_DEGREE = DEGREE:RANGE(-90..90)
CREATE TYPE MINUTE = NEW INTEGER:RANGE(0..59)
CREATE TYPE SECOND = NEW INTEGER:RANGE(0..59)
CREATE TYPE PRECISION = NEW INTEGER:RANGE(0..9999)
```

Note: This definition of precision is biased toward a decimal representation. As an alternative, one might define precision as a positive integer, p, coded as a bit string of length n, to represent the fraction p/2**n.

```
CREATE TYPE LATITUDE = RECORD
        (deg:LAT_DEGREE,min:MINUTE,sec:SECOND,prec:PRECISION)

CREATE TYPE LONGITUDE = RECORD
        (deg:LONG_DEGREE,min:MINUTE,sec:SECOND,prec:PRECISIO
        N)

CREATE TYPE MEASURE = RECORD
        (deg:DEGREE,min:MINUTE,sec:SECOND,prec:PRECISION)
```

With these definitions the normal field extraction operations on RECORD can be used to extract "degree", "minute", "second", and "precision" from LATITUDE and LONGITUDE even if values of these types are stored in vendor specific formats. As currently defined, PRECISION is a partition of SECOND into 10,000 subunits. Reference [1] recommends this level of precision as adequate (i.e. within 1/8-th inch) for identifying any point on the surface of the Earth.

Other functions can be defined on LATITUDE and LONGITUDE to extract Universal Transverse Mercator System (UTM) units or other units for accepted ways to identify points on the surface of the Earth (see [1]).

Define Addition/Subtraction operations so that LATITUDE, LONGITUDE and MEASURE are additive groups.

```
CREATE TYPE LOCATION = NEW
RECORD(lat:LATITUDE,long:LONGITUDE)
```

Determine if it makes sense to define arithmetic operations on LOCATION. Define distance between LOCATIONs via great arcs on the surface of the earth (or use other accepted GIS distance measurements).

```
CREATE TYPE REGION = NEW SET OF LOCATION
```

Define Earth_Surface = Universe()

Note that Earth_Surface has a "finite" domain (i.e. approximately 2**70 elements). All instances of REGION that represent contiguous geographic entities (e.g. countries, cities, bodies of water) can be represented efficiently and the set operations of Union, Intersection, Complement can be optimized using the methods of [9].

Define CAST operation from LOCATION to REGION.

Define QUADRANT as
RECORD(base:LOCATION,lat:MEASURE,long:MEASURE)

Define CAST operation from QUADRANT to REGION.

A "quadrant" represents a "rectangular" region on the surface of the earth, i.e. all locations within the rectangle determined by the "base" and the "lat" and "long" measures.

Define CAST operations between QUADRANT and RECTANGLE in EUCLIDEAN space.

Import the Definitions of Euclidean Geometry for 2-space, especially segment, polysegment, polygon, and convex polygon.

Define CAST operation from POLYGON to REGION to be the set of all locations that: 1) fall on the boundary or 2) fall in the interior of the polygon. Note that the definition of interior may be tricky unless the polygon is a convex polygon.

Define other important transformations between Euclidean 2-space and the Earth_Surface.

With the above data structures to represent Earth_Surface geography, many simple queries can be answered quite easily. For example, the query "find the closest international airport to Chesapeake Bay" or the query "find the locations of all hotels within 3 miles of Interstate 95 between Washington and New York" can be answered by treating all such geographic objects as REGIONs. The answer to the first query is the distance between the set of all locations of international airports and the set of locations comprising the region Chesapeake Bay, and the answer to the second query is the intersection of the three regions: 1) set of all hotel locations, 2) 3-mile BUFFER around Interstate 95, and 3) Washington-NewYork-Corridor.

Another typical geographic query is to ask if Region A lies to the NorthWest of Point B. This is easily modeled by defining NorthWest of Point B as a Quadrant and then determining if Region A is contained in the Region determined that Quadrant.

4.7 Spatial data types

There is a need to add new abstract data types to model appropriate operations for spatial data management. A detailed effort in this area is reported in [2]. Among the suggested ADTs is a geometry ADT which manages geometric information and topologic relations.

4.8 ASN.1 data type

The Abstract Syntax Notation (ASN.1) standards (ISO 8824 and ISO 8825) define a number of data types along with an underlying representation for each of those data types as an Octet String, i.e. LIST OF OCTET where OCTET = ARRAY [0..7] OF BIT. SQL developers should make sure that all SQL data types can be cast to some ASN.1 data type (preferably a predetermined one), and that all ASN.1 data types are representable in SQL. With the appropriate cast operations to Octet Strings, the RDA standard can be used for interoperability, even if the communicating systems don't fully support one another's data types.

5 External repository interface (ERI)

GIS applications require access to multiple data repositories, not all of which are managed by SQL conforming processors; in fact, most are probably managed by non-SQL processors. It is not unusual for GIS applications to require data from the operating system, from graphics repositories, from CD-ROM's, from CAD/CAM databases, or from libraries of catalogued data. We need to consider a new conformance alternative in SQL so that such non-SQL data repositories can make their data available, in simplified but standard form, to SQL systems or SQL applications. NIST representatives have proposed development of such new interface specifications to ANSI and ISO standardization committees. The next year will see if we can persuade others to see the benefits.

This "consciousness-raising" discussion is intended to heighten awareness to user requirements for better integration among heterogeneous data repositories. It may make sense to specify a "client" and a "server" interface to external repositories so that non-SQL systems can act as servers to SQL requests for data. It may make sense to propose a new standardization project to develop the conformance requirements needed for non-SQL systems to provide SQL views of their data and for SQL systems to provide fully functional views over that data to SQL users. This interface might be specified as part of the emerging SQL3 standard [14], or processed as a completely separate standard. In any case, it would repackage functionality from the SQL and RDA standards and give new conformance requirements for both SQL and non-SQL systems.

If we label this new interface as the SQL external repository interface (SQL/ERI), then it might consist of a "server" part and a "client" part. Non-SQL systems could claim conformance to the "server" part and SQL systems could claim conformance to the "client" part. A wide range of non-SQL products and services might be able to claim conformance as SQL/ERI Servers. They could provide high level abstract data types with application specific methods and operations. They might even be required to provide an SQL schema definition to describe data that will be externally available. In addition, SQL/ERI Servers would be required to evaluate "simple" SQL queries over individual objects defined in the schema. The exact meaning of "simple" can be specified in the ERI definition -- possibly at different levels of service. The SQL processor can then think of the external repository as an SQL_CATALOG that can be CONNECTed to, but that can only respond to whatever SQL statements are specified for that level of service.

In turn, SQL systems might be able to claim conformance as SQL/ERI Clients. If an SQL system claims conformance as an SQL/ERI Client, then it agrees to provide SQL functionality, at whatever level of the SQL standard it conforms to, over any table or other object provided by an SQL/ERI Server. This may require that the SQL system automatically create a temporary table whenever the external view is referenced in a query, and then populate that table using the limited capabilities provided by the "server" interface so that it can guarantee the ability to perform nested queries, or searched updates and deletes, or recursive queries, or whatever is requested by its application.

With the SQL/ERI "client" and "server" definitions, non-SQL systems would be able to provide services to SQL-based applications even though they might not be able to provide the expected query flexibility, access control, concurrency control, or updatability required of a full-function SQL data manager. Full-function SQL processors could provide these expected data management facilities and, in addition, provide user access to data repositories not otherwise accessible via the SQL language. Section 2.2 describes how the SQL/ERI definitions might be used to provide uniform GIS application access to both SQL and non-SQL data at local and remote sites.

The SQL/ERI standard might provide several different conformance packages for non-SQL systems. Certainly one kind of conformance package might coincide with the services provided by read-only tables on CD-ROM. Data on the CD-ROM would "conform" to this SQL/ERI server package if it includes a data manager kernel that is executable on a wide range of workstations and responds correctly to an SQL CONNECT statement using the call interface provided by that workstation. Another kind of conformance package might coincide with the services provided by online databases. Such systems would be required to respond to RDA requests as if they were remote SQL servers conforming to the RDA/SQL-Specialization. Other conformance packages might correspond to the services required in a multi-vendor environment with some updatability requirements and other requirements to "read" SQL data in other remote catalogues.

In each of these cases, a conforming SQL/ERI server would be required to be "self-describing" as if it were a separate SQL catalog. It would be required to supply an SQL Information_Schema describing all available tables and other objects and the equivalent SQL data types for all attributes. In particular, the following tables from the Information_Schema would probably be required, although some might be empty: TABLES, COLUMNS, DOMAINS, VIEWS, and SCHEMATA. If the ERI Server provides new abstract data types not defined in the SQL standard, then it would also be required to provide an SQL ADT interface definition as specified in the emerging SQL3 specification.

6 Conclusions

Database Language SQL provides standardized facilities for defining, managing, and protecting data. With implementations available on all sizes and makes of computing equipment, the SQL standard is leading the way toward unprecedented portability of database applications. The emerging SQL3 specification includes object management capabilities over abstract, user-defined data types, thereby making SQL3 a complete language for creating, managing, and querying persistent objects. The emerging RDA standard promises to complete the link among SQL products from different vendors, leading to true open systems interconnection and interoperability among conforming SQL systems. Emerging specifications for future revisions of SQL and RDA promise enhanced facilities to support intelligent objects and knowledge-based applications in a distributed processing environment.

GIS requirements for integration of SQL and non-SQL data repositories can be met with the new user-defined data types in SQL and emphasis on a new common external repository interface (ERI) linking SQL to non-SQL data managers. An SQL/ERI standard, based on an appropriate subset of SQL and RDA capabilities, will provide new opportunities for integration of heterogeneous data repositories into GIS applications. Non-SQL data repositories would be able to use the object-oriented ADT definition facilities in SQL to present a standardized, yet functionally complete, external schema to GIS applications. With support from full-function SQL processors on one side of the ERI interface, and standardized access to data and data operations on the other side, applications can take full advantage of high-level data structures and operations provided by specialized, non-SQL processors while at the same time depend on the availability of full-function SQL statements to access and manage the data.

Armed with full-function SQL and RDA implementations at each remote site, and ERI access to specialized tools and data repositories, GIS applications will be able to specify, via an SQL statement, what data is to be analyzed, and will be able to direct that data to a chosen application tool, analyze data through the eyes of that tool (e.g. sophisticated design analysis tools), and specify where the result should be directed for further access by other tools. The interoperability capabilities provided by SQL and RDA allow integration of data and applications from various processing sites. With these data management standards, and with other capabilities provided by emerging specifications, the GIS goals of *seamless* interoperability are within reach.

References

1. ANSI, *American National Standard for representation of geographic point locations for information interchange*, ANSI X3.61-1986, approved 23 June 1986.

2. Ashworth, Mark. "A Geographic Information Systems Perspective on Spatial and Object-Oriented Extensions to SQL," NIST monograph on SQL and Geographic Information Systems, edited by V.B. Robinson, 1993.

3. Bauer, Jonathan, *et al.* "Polymorphic functions", document ISO/IEC JTC1/SC21/WG3 DBL KAW-7 (X3H2-91-135), June 1991.

4. Bauer, Jonathan, *et al.* "Parameterized types", document ISO/IEC JTC1/SC21/WG3 DBL KAW-41 (X3H2-91-274), October 1991.

5. Beech, David and Hasan Rizvi. "Tables and subtables as sets of objects", document ISO/IEC JTC1/SC21/WG3 DBL KAW-8 (X3H2-91-142rev), June 1991.

6. Cannan, S.J. and G.A.M. Otten. *SQL - The Standard Handbook*, McGraw-Hill Book Co, Berkshire SL6 2QL England, October 1992.

7. Date, C.J. with Hugh Darwen. *A Guide to the SQL Standard*, Addison-Wesley Publishing, Reading, MA 01867 USA, October 1992.

8. Eisenberg, Andrew and B. Johnston. "Additional control statements for SQL", document ISO DBL KAW-14 (X3H2-91-179), June 1991.

9. Gallagher, Leonard. *Computer implementation of a discrete set algebra*, NIST technical report, NISTIR 4637, July 1991.

10. Gallagher, Leonard and Joan Sullivan. *Database Language SQL: Integrator of CALS Data Repositories*, NIST technical report, NISTIR 4902, September 1992.

11. Hoffman, Kenneth and Ray Kunze. *Linear Algebra 2nd Ed*, Prentice-Hall, 1971.

12. ISO/IEC CD 11404. *Common Language-Independent Datatypes (CLID)*, 2nd Committee Draft, document ISO/IEC JTC1/SC22 N1305, December 1992.

13. ISO/IEC 9075. *Database Language SQL*, International Standard ISO/IEC 9075:1992, American National Standard X3.135-1992, American National Standards Institute, New York, NY 10036, November 1992.

14. ISO/IEC SQL Revision. *ISO-ANSI Working Draft Database Language SQL (SQL3)*, Jim Melton - Editor, document ISO/IEC JTC1/SC21 N6931, American National Standards Institute, New York, NY 10036, July 1992.

15. ISO/IEC 9579. *Open Systems Interconnection - Remote Database Access (RDA), Part 1: Generic Model and Part 2: SQL Specialization*, International Standard ISO/IEC 9579:1993, American National Standard ANSI/ISO/IEC 9579, American National Standards Institute, New York, NY 10036, publication expected mid-1993.

16. Kulkarni, Krishna, *et al.* "Inheritance for ADT's", document ISO/IEC JTC1/SC21/WG3 DBL KAW-6 (X3H2-91-133), June 1991.

17. Lichnerowicz, Andre. *Linear Algebra and Analysis*, Holden-Day, 1967.

18. Melton, Jim and Alan Simon. *Understanding the New SQL: A Complete Guide*, Morgan Kauffman Publishers, San Mateo, CA 94403, October 1992.

19. Halustchak, O. Proposed Spatial Data Handling Extensions to SQL, in Robinson, V.B. and Tom, H. (ed.s): *Towards SQL Database Language Extensions for Geographic Information Systems*,NISTIR, National Institute of Standards and Technology, Gaithersburg, MD, p. 69-84.

Leonard Gallagher is a computer scientist in the Information Systems Engineering Division at NIST. He is responsible for data models, database standardization, and integration of database technology with new approaches to information management such as knowledge and object-oriented systems and hypertext. He has been a member of the ANSI/X3 technical committee on Database, X3H2, since 1979 and chairs the ISO/IEC JTC1/SC21/WG3 rapporteur group for follow-on enhancements to the ISO SQL standard. At NIST, he leads an inter-division project for research and experimentation into hypertext and hypermedia information management systems. Dr. Gallagher received the BA degree in mathematics from St. John's University of Minnesota in 1965 and the Ph.D. in mathematics from the University of Colorado in 1972. He taught mathematics at the Catholic University of America for 6 years and has been involved in database research at NIST for the past 15 years.

Dr. Gallagher can be reached by telephone at +1-301-975-3251 or by electronic mail using the Internet address LGallagher@nist.gov.

Proposed Spatial Data Handling Extensions to SQL

Orest Halustchak, Senior Product Architect
GeoVision Systems Incorporated
1600 Carling Avenue, Suite 200
Ottawa, Ontario K1Z 8R7
Canada

1 Introduction

The use of relational database management system (RDBMS) technology within the existing GIS user community is virtually universal. It is likely that RDBMS will be the major commercial data management technology for the remainder of the decade, even with the current emerging interest in object-oriented database technology. The Structured Query Language (SQL) will be the major language for interaction with relational databases and is being supported by many of the new object-oriented databases as well. Furthermore, the investment by organizations in training staff to use SQL is quite significant. Consequently, a large body of expertise is available both within the Geographic Information Systems (GIS) user community and computing industry as a whole.

The use of RDBMS and SQL by (GIS) has emerged as a *de facto* industry standard. Even though SQL is a well defined standard, its implementation by GIS companies in their software products varies considerably. Most GIS companies have made significant investments into the design and development of proprietary interfaces and spatial data extensions to support SQL. These GIS companies include Kork (Ingram and Phillips, 1987), Intergraph (Herring, Larsen, and Shivakumar, 1988), Prime (Charlwood, Moon, and Tulip, 1987), and GeoVision (Westwood, 1989). Each has attempted to provide facilities supporting spatial predicates and spatial data manipulation facilities within SQL (or SQL-like query languages).

Since the acceptance of SQL as a de facto GIS standard, the GIS user community has developed specific expectations about portability, functionality, and performance. Unfortunately, the basic query language in most cases is an incomplete implementation of both the ANSI and FIPS standards (ANSI, 1989; NIST, 1987). Each company provides its own set of fairly minimal and elementary spatial extensions. These extensions generally do not maintain consistent syntactic and semantic constructs with the rest of SQL, e.g. spatial predicates are not in general supported within the WHERE clause, but rather within a separate clause.

Many leading government agencies are starting to advocate the certification of SQL support by GIS software products. The key to certification is to define an interface specification that may be used to design and develop validation tests. This paper is intended to be an initial framework that may be used to develop such a specification into a standard set of spatial extensions that may be eventually embedded into the SQL ISO/ANSI standard.

Our approach is to identify a set of enhancements that would be of general use to all GIS vendors and in many cases to non-GIS vendors as well. The proposed enhancements provide significant capability over the current SQL standard for GIS applications. One could define many more enhancements that would be useful for GIS, but we, at GeoVision, decided to identify a set of enhancements each of which would have a good chance of being accepted by all GIS vendors and also by many non-GIS users.

Detailed information is provided in the Functional Requirements section. Functions identified include long term lock and transactions, custom functions, auto-sequence numbering, data types, triggers, and n-dimensional indexing. These descriptions are intended as starting points for discussion and as such are not rigorous definitions of data types and functions. The specific syntax examples included in this proposal are intended as initial suggestions. The discussion on long transaction support presents the long transaction concept, but does not include specific SQL implementation suggestions.

2 Functional Requirements

The functional requirements described in this section are for extensions to SQL that would be of major benefit to a GIS application using a relational database management system (RDBMS) as its only storage mechanism. This section defines enhancements that are not necessarily GIS specific, but could be used for other applications. Even the location data type might have uses outside a GIS specific application, for example in a CAD application. In some cases, such as the data types sections, SQL syntax and semantic changes are suggested. In other cases, in particular, the long transactions section, concepts and requirements are outlined with no specific SQL syntax suggestions.

2.1 Data Types

We have identified two sorts of implementation of data types, ones that are intrinsic to SQL like the current character and numeric types, and ones that users could define for their specific applications. The two intrinsic types described here are *location* and *array* data types.

2.1.1 Location Data Type. . Geographic information systems manage data about objects in a spatial context. The geographic location of an object is an important datum. The concept of "location" almost always is expressed as a value with multiple dimensions. This concept has uses not only in GIS where location represents a location on the ground, but also in CAD, mathematics, and various other engineering and scientific applications.

The data type is represented by n numbers, where n is either two or three. To allow flexibility, it should not restrict n to two or three, but allow higher dimensional locations. For GIS, the basic requirement is for 2D or 3D data. For example:

```
create table spotheight (
        id          numeric primary key,
        coords      three_d_coordinate(12,2),
        ...
);
```

Above, the parameters for three_d_coordinate are meant to be the same as for the numeric type, number of digits and precision. Instead of having specific types for 2D and 3D coordinates, there could be one type with the dimension as a parameter.

```
create table spotheight (
        id          numeric primary key,
        coords      coordinate(3,12,2),
        ...
);
```

Of course the reason for having a location data type is not merely to have a compact way to represent them. The SQL operators would have to support the location data type.

```
select ... where location_1 between location_ll and location_ur;
```

The section on user defined functions later in this document has more examples of the use of the location data type.

2.1.2 List or Array Data Type. . A problem in modeling GIS data in a relational database is how to represent long ordered coordinate strings or large arrays of raster data. Representing this data in normalized tables is inefficient both in terms of processing and in terms of storage overheads.

```
create table contours (
        id          numeric primary key,
        feat_code   char(10),
        z           numeric(10,4),
        ...
);

create table coords (
        id          numeric references contours (id),
        pt_num      numeric,
        coord       coordinate(2,12,2),
        ...
        primary key (id, pt_num) );
```

Here, id is the unique id of the feature. The pt_num field is required to maintain the correct order of points for the feature. These two fields are repeated for every single point. If one considers that the primary key requires an index for quick retrieval, then the overhead of identifying a single point uses more storage than the actual point itself takes.

Almost all queries that require the coordinates for a contour line require the coordinates in order, for example to draw the contour line on a plot. This requires an order by clause.

```
select coord from coords where id = n order by pt_num;
```

This is not too bad for most database systems which are fairly efficient in performing the sort if the field being sorted is indexed. However, in most cases when coordinates are to be retrieved, a join is involved also. For example, the following query is required to retrieve 100 metre contours.

```
select contours.id, contours.feat_code, coords.coord
from contours, coords
where z = 100 and contours.id = coords.id order by coords.id, pt_num;
```

This not only includes a sort of all selected rows, but also returns the id and feat_code along with every single point. Some features, especially contour lines, require hundreds or thousands of points to represent them accurately. Furthermore, in more realistic examples, more than two attributes would be retrieved with every feature. This query is usually processed as two separate queries, one to retrieve the basic feature attributes (id and feat_code in this example) and a second query which returns only the coordinates.

Several relational database vendors now support a "long" data type to address this and similar problems. It is a field that can hold a large amount of data with limits large enough that they could be used for small images or for coordinate strings. The advantages with using that type of field are: a separate coordinates table is not require saving the join; the overhead for identifying each point is not needed; and the query can be processed with no sort required. A significant disadvantage is that these fields cannot be used in expressions or comparisons. The internal detail of these fields is completely unknown to the database management system, so this type is not ideal. What is required is an ordered list or ordered array data type, for example

```
create table contours (
        id          numeric primary key,
        feat_code   char(10),
        z           numeric(10,4),
        coords      two_d_coordinate(12,2) array(1...unlim),
        ...
);
```

In the example, the field *coords* is an array of elements, where each element is a location data type called *two_d_coordinate*. It is intended that the number of elements be variable and unlimited. The relational database vendors, however, likely would have some limit on the size

of arrays, the same way they have limits on the maximum number of rows per table. The query for all 100 metre contours is as follows.

```
select id, feat_code, coords from contours where z = 100;
```

The provision of arrays requires further syntactic enhancements for inserting, updating, and selecting elements of the array.

2.2 User Defined Functions

The proposed SQL3 standard includes "External Procedures" which is what I refer to as user defined functions. The ability to write a function in a language like C and have it available as a function within SQL is a very powerful capability. It allows GIS vendors to embed many of their spatial functions right into SQL.

Currently, during data selection, if there are spatial functions involved in the selection, extra data needs to be retrieved from the database. The selected data is then further filtered by GIS software implementing the spatial functions. For example, consider a query "Find all parcels within 4 kilometers of a given location." Different vendors have different ways of handling this, but in general, the query cannot be phrased accurately enough in SQL. Parcels in the general geographical area are retrieved and then the spatial clipping functions are performed afterwards.

Benefits include less data retrieved from the database and more powerful phrasing of queries. User defined functions might include geographic distance, projection transformation (eg. UTM to Mercator), unit conversion (eg. feet to metres), and more accurate selection (clipping).

Functions can also be written to perform specific operations on the previously described location or array data types. For instance, a function could define a location data item out of separate coordinates, or a function could extract a coordinate out of a location data type such as

```
insert location(308219.45, 6209483.75, 450) into loc_table;
select ordinate(1, start_locn) from loc_table;
```

Or another example is

```
select * from homes
where distance(location(308219, 6209483), homes.locn) < 20;
```

In the above example, *homes.locn* is a location data type, but it could either be a single location field or it could be an array of location fields, in which case the distance function returns minimum distance.

In

```
select building.name from building, parks
where inside(building.locn, parks.perimeter)
and parks.name = 'Algonquin';
```

the function *inside* would perform a point-in-polygon operation to determine if the building's location is inside the area enclosed by the park's perimeter.

Some GIS functions that would be required could be implemented as operators rather than functions. They would be spatial predicates in SQL such as *inside, connected, crosses,* and *north_of.*

In

```
select road.name, rail.lineno from road, rail
    where road.coordinates crosses rail.coordinates;
```

coordinates would be arrays of location data type. This could also be implemented as user defined functions, so the requirement for user defined operators is not as strong as the requirement for user defined functions.

```
select road.name, rail.lineno from road, rail
where crosses(road.coordinates, rail.coordinates);
```

For the following examples using the location data type, the database management system would need to support the semantics of the following types of expressions.

```
update buildings set locn = locn + location(500, 100);
select building.locn - fire.locn from ...;
```

2.3 Triggers

The integrity enhancement with the current SQL standard is very useful for maintaining the integrity of relationships between rows of data in different tables. However, in GIS there are cases where the relationship of information goes beyond referential integrity. For example, the area of a polygon feature is dependent on the boundary coordinates of that polygon. If a user alters the boundary information, a trigger could initiate another user function which, in this example, would recompute the area value.

A trigger function could also be used to perform complex validation of input data. For polygons, a trigger function could validate that an updated centroid point is inside the bounds of the polygon area. This trigger capability is included in the proposed SQL3 standard.

2.4 Long Term Transactions and Locks

Currently, a transaction is assumed to be fairly short in duration, seconds or minutes, but certainly not hours. In GIS, the situation is more complicated. Yes, it still makes sense to have the same sort of transaction concept. A user digitizes a feature, enters some related attributes, and updates some related features. That then is committed as a unit of work. It makes sense that if something goes amiss during that unit of work that the transaction is rolled back.

However, users of GIS also have a longer duration *transaction.* A user who may be designing

changes to a part of a GIS may take hours or days to make all the necessary changes. This often will span several sessions with the user logging out between sessions. Some examples include:

- A planner is designing a new road network for a proposed subdivision. This may require adding or changing hundreds or thousands of features.

- An engineer is designing changes to a telephone network in a local area as a work order.

- An engineer is analyzing a *what-if* scenario in an electrical network. The task analyses the effect of a downed power line on the rest of the electrical network.

- A user is analyzing the potential impacts of a hypothetical forest burn area on wildlife habitats.

Users expect to have similar capabilities with these *long transactions* as they do with the current definition of transaction, but there are differences.

Similarities:

- The *long transaction* can be rolled back if the user decides not to apply any of the changes.
- The user will commit the *long transaction* when done. After commit, all users will be able to see the changes.

Contrasts:

- The user still requires transaction control at a small unit of work level. In the example of a planner designing a new road network, the rollback capability is still necessary to rollback errors made to the most recently entered feature, but without rolling back the whole design effort. So, the requirement is not just a matter of adding capability to extend the life of a transaction. It requires a two-level mechanism. Many instances of the current concept of transaction, the *short* transaction are nested sequentially inside a *long transaction.*
- If the system crashes, the rollback is not to the beginning of the *long transaction*, but only to the beginning of the small unit of work, the *short* transaction.
- A long transaction is not terminated automatically when the database is closed. A long transaction can span multiple sessions.
- Changes being made during the *long transaction* can be made available selectively to other users before the *long transaction* is committed. This is necessary in a design environment where several engineers collaborate on a single design project. The visibility of the long transaction is controlled, so not all users see it. Rules of transaction serializability do not apply to long transactions.

- Data that is being updated or referenced by the *long transaction* is subject to different locking rules. For short transactions, locking is used to enforce concurrency based on a serializable transactions model. For long transactions, persistent locks which can last as the length of the long transaction are required. A user may be designing modifications to a piece of equipment and wishes to maintain an exclusive lock against update of that object. Another type of lock is a shared lock where more than one user may be changing the object within their transactions. This lock is used when users are working in overlapping areas and each user potentially may need to update the same object. More information on this follows later.

I present an approach to supporting long transactions which is based on a multi-version management system. The approach requires the support of multiple versions of any given row in the database. The description lays out some main requirements that need to be supported by the database management system.

2.4.1 Multiple Versions. In the course of designing the new set of features, the designer may go through several *what-if* scenarios. An engineer designing a change to a telephone network may need to examine several different cable configurations. During the whole design process, the database stores all variations of the data. The database stores several *versions* of a given *object*, where an object may be represented by a set of rows in one or more tables. Different users have a requirement to view different sets of versions. Some users may wish to see only data that is a representation of the data as it is in the field. Other users may wish to see different stages of the designs. In a multi-user database, there will be many engineers doing design work, sometimes in different geographic areas, and sometimes in overlapping geographic areas.

The support of versions has been an issue in CAD applications (eg. Kemper, Wilkes, and Schlageter, 1991; Ahmed and Navath, 1991), and is becoming important in GIS applications, especially in utility applications (eg. Easterfield, Newell, and Theriault, 1990). A version management mechanism supports the storage of many versions of a single object with different users working on different versions. However, a single *view* of the database will contain only one version of any given object. A given version of an object may be seen in many views. Permanent objects are seen in all views of the data.

Different projects are often independent of each other. For example, two engineers are designing road changes to opposite ends of the city where there is no overlap between the data being changed for the two projects. But, this is not always the case. Sometimes an engineer designs changes in an area that overlaps with the area being worked on by another engineer. In fact, the changes made by one engineer may depend on specific changes being made by the other engineer. There is a dependency of one project on another project. For example, one engineer may be designing the placement of a cable that attaches to a switch being placed within another engineer's project. The first engineer then has a concern whenever a design change happens that affects the dependency.

For any object in the database, there can be multiple versions. A version of an object may depend on some other version of the object. Objects are also related to one another. Let us take a very simple example of a cable attached to a switch. If a cable is represented as a row in a CABLE table, and a switch is represented in a SWITCH table, then for a many-to-two relationship, the switches to which a cable is attached are represented using foreign keys in a CABLE_SW table.

```
CABLE (
      cable_id    char(10) primary key,
      type        char(8),
);


SWITCH (
      switch_id   numeric(5),
      type        char(8),
);

CABLE_SW (
      cable_id    char(10) references cable (cable_id),
      end_ind     char(1);    /* S = start, E = end */
      switch_id   numeric(5) references switch (switch_id),
primary key (cable_id, end_ind) );
```

In a database where one is not concerned about versions of objects, the switch_id can refer to one and only one row of the SWITCH table. However, in a database supporting multiple versions of any given row, the switch_id foreign key may refer to several versions of the switch. The referential integrity is more complicated. I am going to attach version numbers to each row of these tables so as to be able to identify them. Let us say that there is a switch with id 100 and type A. Let us call it version 0, meaning that it is a permanent version. An engineer is designing a change to the network that requires a new cable, id C12, type X, and version 1. The engineer attaches start end of the cable to the switch. Another engineer determines that for his project, the switch must be replaced with a type B switch. Fortunately, the type B switch can support the type X cable required by the first engineer. We now have another switch with id 100, but of type B and version 2. A third engineer is basing her design on the work of the other two engineers. She needs to upgrade the type X cable to a type Y cable. The type Y cable can still attach to the type B switch. The type Y cable is version 3.

SWITCH

switch_id	type	version
100	A	0
100	B	2

CABLE

cable_id	type	version
C12	X	1
C12	Y	3

CABLE_SW

cable_id	end_ind	switch_id	version
C12	S	100	1

Each engineer has a different view of this data. The third engineer sees a type Y cable attached to a type B switch. The second engineer sees a type X cable attached to a type B switch. The first engineer sees a type X cable attached to a type A switch. Finally, a user that needs to see an "as-built" view sees a type A switch, but sees neither the type A nor the type B cable.

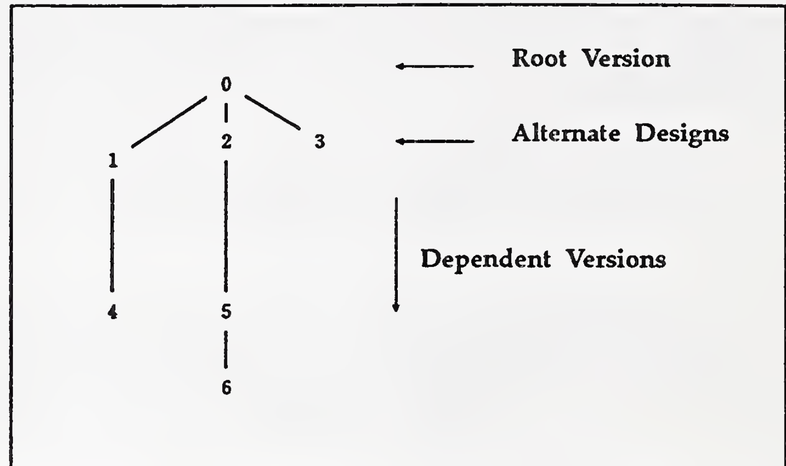


Figure 1. Version Dependencies.

The various versions of a given object can be represented as a tree structure (Figure 1). The branches leaving the object represent design alternatives based on that object. Each alternative may in turn have their own branches representing other alternatives which are dependent on that alternative.

A task usually involves changing many objects in the database. Continuing with the scheme where we assign a version number to each change, let us assume that all changes made for a single task are assigned a single version number. The task may be a simple work-order or it may be a what-if analysis scenario. Often, a user will need to explore several what-if scenarios, comparing the results of each of the scenarios. Each scenario is a separate task. An entire project would be composed of many tasks and often would be the result of collaboration between several users.

Although the changes made within a task are made using a single version number, changes may be made to data with other version numbers. For example, in Figure 2 switches S1 and S2 are modified using versions 7 and 8 respectively. Cable C30 is modified under version 8. Then, under version 9, cable C20 is removed and replaced by cables C21 and C22 and switch S3. Version 9 is directly dependent on versions 1, 7 and 8. Notice that the version 9 view contains data from versions 0, 1, 7, and 8.

Representing the task dependencies in a graph form, the graph is a directed graph. (Figure 3). Contrast this with figure 1 which showed the version dependencies only for a single object. For a single object, the dependency graph is a tree since a single version of an object is dependent directly on only one other version of that object. However, for a task, a single version may be dependent directly on many other versions.

The view of the database that one has with a single version is based on a complex relationship between versions. The specific version of a given object that is within the view depends on the task dependency graph. This is an important concept in a multi-user design database. Each user can select the view of the database that they require. Their view is chosen by the design project and stage within that project. Once a view is established, the user will perform analysis without needing to specify which version of which object needs to be seen. The view appears as a separate copy of the database. Other users working on other projects will not disturb that view. Similarly, any what-if changes made by the user will not disturb other users of the database who are working on different projects.

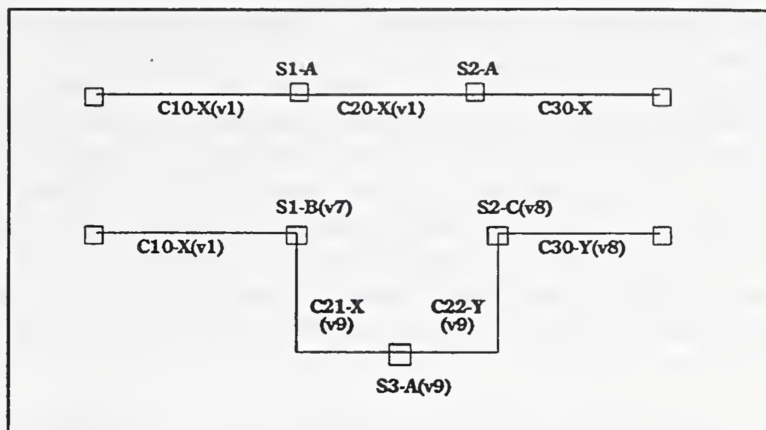


Figure 2. Example of Task Dependency.

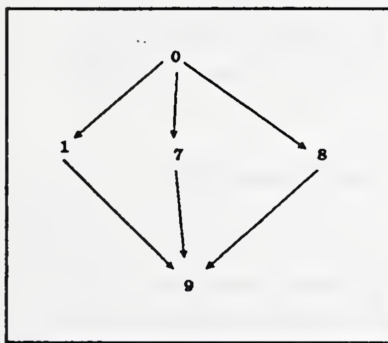


Figure 3. Task Dependencies.

There is one exception to this view mechanism. In work-order processing, it is often necessary to display graphically on a plot an indication of what changes are to be made. This type of plot is used in the field to guide the workers. For example, if a work-order removes a cable, the old version of the cable is displayed with an x'ed out symbolization. If a cable is to be changed, the old and new attributes of the cable are displayed, but using different colours. So, even within a view of a database, applications do need to access previous versions of the data.

As with short transactions, there are commit and rollback operations for long transactions. A commit operation makes versions of objects, which represent conditional changes, permanent objects. Conditional changes made to permanent rows replace those permanent rows. Those rows then become available to all users working on the permanent versions of data. Similarly, a rollback of a version removes all rows of that version without affecting the rows on which they were dependent.

Commits of versions usually do not happen until the design change has been approved, installed in the field, and compared against what was really installed in the field. Occasionally, the field installation will need to make changes that were not anticipated by the designers. For example, a telephone pole may be installed a few feet from the intended spot due to an unanticipated obstruction.

We have seen how a user performs a single task using the long transaction mechanism. All changes made within that task are assigned a single version identification. In practice, a task on its own is insufficient. A project is made up of many tasks, often with different users working on different tasks that make up the project. It is usually not until the whole project is complete that the changes made within the tasks of the project are made available to all users. With tasks supported by the database management system using long transactions, the application can manage the project concept by treating it as an aggregate of tasks. Each task can be assigned application specific attributes, such as "initial design", "initial approval", "corrected design", and "final approval". Committing a project would involve committing each task within the project. In a very elaborate application, this aggregation can be nested to even higher levels.

2.4.2 Referential Integrity Constraints. The presence of multiple versions of rows clearly complicates the matter of referential integrity. Each project view of the database has its referential integrity. The foreign key reference from one end of a cable to the switch identifies a single switch. The project view sees only one version of that switch. However, the database management system must realize that the foreign key may reference many instances of the switch rows. Foreign key references can be ambiguous, unless a version indicator is included as part of the foreign key.

- Unique key constraints - A key is no longer unique within a whole table. It is unique only within a single version.

- Foreign key references - A foreign key may refer to different versions of a primary row depending on the task view that is in effect. Refer to the earlier switch / cable example, above. The view from version 2 is that cable C12 version 1 refers to switch 100 version 2. However, the view from version 1 is that the same cable version, cable C12 version 1 refers to switch 100 version 0. So, while the logical data model requires a one-to-many relationship between SWITCH and CABLE_SW, physically the relational database must represent this as a many-to-many relationship.

2.4.3 Concurrency Control. In a multi-user environment, concurrency control is certainly important. However, in a graphic design environment, the traditional concept of a transaction is too restrictive. Traditionally, within a transaction, a user will examine some records, make a set of changes, and then commit the transaction. In a graphic design environment, the time spent with a set of data is too long to force it to be within a database transaction. For example, an engineer may work with all the electrical outside plant data in a 10 square mile area for a period of a week. The engineer may decide that during that time he does not want any other user editing selected data within that set. This requires a lock mechanism that lasts an extended period of time. The locking done by relational database management systems for concurrency is insufficient.

Most systems accomplish this by adding lock information columns to each table that contains data that could be locked. Then, the lock control is handled by the application or user. It works, but there are some drawbacks

- If a user accesses the data through another application, then the lock controls are not effective.

- All updates by the user must be monitored by the application to verify that there are no lock conflicts. This adds overhead. For example:

```
update parcels set value = value * 1.1 where street = 'pine';
```

The application could modify the query as follows:

```
update parcels set value = value * 1.1 where street = 'pine' and lock is NULL;
```

This update will skip locked parcels, but often that is not the desired effect. Usually the application will want to inform the user if there are any lock conflicts. The application will have to generate a separate query to verify that no locks will conflict.

```
select parcel_id, lock from parcels where street = 'pine' and lock is not NULL;
```

If there are subqueries, it further complicates the application's processing of the operation.

Locking to maintain concurrency control is usually an exclusive type of locking. One user updates an object and other users cannot update that object until the first user releases the lock. The lock can be released prior to the end of a transaction. However, as pointed out earlier, in a design environment with long transactions, exclusive locks are sometimes too restrictive. If several users are working in overlapping areas, an exclusive lock would allow only one user to make changes in the overlapping area. Often that is not appropriate, so users prefer to use shared locks in those situations. A shared lock can be placed on an object by any number of users. When a user eventually decides to update the object, the user escalates his or her shared lock to an update lock. All other shared locks on the object then are discarded. Usually, the other users are notified by the application when their locks have been discarded. Depending on application requirements, the notification can be immediate, or can be done only when the other user attempts to update the object.

Long transactions can be rolled back. This has implications on users who have their tasks which depend on those transactions. In a normal transaction, the database management system prevents users from seeing changes before they are committed. In a long transaction, however, it is necessary for several designers to see the transaction effects prior to the transaction being committed permanently in the database. A view of the transaction is not provided automatically to all users. The user must identify the versions of data that they wish to be in their view. In some applications, the owner of the transaction must grant access to the version to a particular user before that user can see the transaction. After a rollback operation, the other users must be informed so that they can update their dependencies.

Concurrency control for long transactions requires a lock mechanism whereby rows can be locked for extended periods of time, under the control of the user. Both exclusive and shared locks are required. Concurrency control also requires the ability for a user to control what versions within his or her transaction are viewable by other users.

2.4.4 Versioning Requirements Summary. In summary, long transactions can be implemented using a database supported versioning mechanism. These are the main requirements of the database management system to support multiple versions of rows.

Support multiple versions of rows in a table (based on primary keys).

Support dependencies between versions.

Manage referential integrity which takes versions into account.

Support views based on versions and version dependencies.

Provide commit and rollback operations at the version level.

Manage persistent locks at the row level.

I have not suggested a SQL syntax to support these concepts. Instead, I have tried simply to present the concepts of long transactions in a GIS application and to outline the requirements of a database management system to support these concepts.

2.5 Auto Sequence Numbering

It is common among GIS vendors to assign unique identification numbers to features added to a GIS database. One cannot assume that all features that a user would create would have a unique user key. In fact, often the only thing unique about a feature is it's location and data type. Spot heights, contour lines, and boundary features usually do not have unique user keys. Even when the user has keys, they often are of different type for different feature (eg. part number, street name) which makes it difficult to identify features in a general way. Thus, a unique sequence of numbers is generated.

In a multi-user database and potentially in a distributed environment, generating unique sequence numbers is tricky, although it can be done. It would be more useful and efficient if the database management system could generate unique sequence number automatically. For example, consider a "feature" table which contains attributes feature_code (eg. street, parcel), type (eg. line, point, polygon), and perhaps others. A further attribute would be feature_number and it would be defined as a unique sequence number. When one would insert a row into the "feature" table, one would include all columns except feature_number.

```
insert into features (feature_code, type, ...) values ('street', 'line', ...);
```

The database management system would compute a new feature_number.

2.6 N-Dimensional Indexing

The "Create Index" statement is not part of the SQL standard. However, it is generally expected that vendors provide this sort of command and therefore I have included a requirement here. Most indexes available are B-Tree, hashing, or other method that handles only one dimensional data. One can index several fields, but it becomes a concatenated index rather than a multi-dimensional index. In GIS, it is essential to be able to provide at least a two-dimensional indexing capability on coordinate location. For some applications, a three-dimensional index would be desirable. There are many published spatial indexing algorithms. Two commonly known spatial index structures are R-Trees and Quadtrees. One of these or some other multi-dimensional index structure is required. For example:

```
create index geo_index on spotheights(x, y);
```

or using the location data type,

```
create index geo_index on spotheights(coords);
```

Note that the syntax is no different than what many vendors support currently. However, the intention is that the index is treated as a 2-D index and not as a 1-D concatenated key index.

3 Conclusions

I have presented some of ideas for basic enhancements to SQL to support GIS data types and functions. We at GeoVision feel that these enhancements would provide significant capability over the current SQL standard, but at the same time would not hamper the various GIS vendors from developing distinctive products or inventing new ways of modeling and manipulating GIS data. We also feel that all the concepts described in this paper can be used for other non-GIS applications. Location and list data types can be used in any application that deals with multi-dimensional data such as CAD / CAM. Versioning can be used in any application that requires the support of on-line design work and that includes most engineering or architecture types of applications.

It is hoped that ideas presented in this paper provide insight into some of the data management issues that GIS needs to solve and that the ideas can be used as a starting point for further discussions.

References

Ahmed, R., and Navathe, S. B. (1991), "Version Management of Composite Objects in CAD Databases", *Proceedings 1991 ACM SIGMOD*, May, pp. 218-227.

ANSI X3.135-1989, *Database Language - SQL with Integrity Enhancement*, 1989

Charlwood, G., Moon, G., and Tulip, J. (1987). Developing a DBMS for Geographic Information: A Review. **Proceedings Auto-Carto 8.** Baltimore, Maryland.

Easterfield, M. E., Newell, R. G., and Theriault, D. G. (1990), "Version Management in GIS - Applications and Techniques", *European Conference on Geographic Information Systems*,April, pp. 288-297

Gallagher, L. (1991). **SQL3 Support for CALS Applications.** NISTIR 4494, National Institute of Standards and Technology, US Department of Commerce: Gaithersburg, Maryland.

Herring, J. R., Larsen, R. C., and Shivakumar, J. (1988). Extensions to the SQL Query Language to Support Spatial Analysis in a Topological Data Base. **Proceedings GIS/LIS '88.** Orlando, Florida.

Ingram, K. J. and Phillips, W. W. (1987). Geographic Information Processing Using a SQL-based Query Language. **Proceedings Auto-Carto 8.** Baltimore, Maryland.

Kemper, F., Wilkes, W., and Schlageter, G. (1991), "Basic Mechanisms to Support Versioning in the Database Component of a CAD Framework", *Electronic Design Automation Frameworks, Proceedings of the 2nd IFIP Space WG 10.2 Workshop*, pp. 171-178

NIST, (1987). **Database Language SQL.** FIPS Publication 127, National Institute of Standards and Technology, US Department of Commerce: Gaithersburg, Maryland.

Robinson, V. B. (1990). **Structured Query Language and Geographic Information Systems: A Briefing Review.** Institute for Land Information Management, University of Toronto: Toronto, Ontario

Strand, E. (1991). Standards Conformance or System Conformance. **GIS World.** May, pp. 64, 66, 67.

Westwood, K. (1989). Toward the Successful Integration of Relational and Quadtree Structures in a Geographic Information System. **Proceedings National Conference on GIS - Challenge for the 1990's.** Ottawa, Ontario.

Orest Halustchak is a senior product architect with GeoVision Systems Inc. Currently, he leads the R&D team for the database related products of GeoVision's Vision* product line. As senior architect, he also provides technical input into the architecture design of future products and product releases. He has been with GVSI and its predecessor company since 1977. Orest received a BSc. Honors Physics degree from the University of Manitoba in 1976.

A Geographic Information Systems Perspective on Spatial and Object Oriented Extensions to SQL

Mark Ashworth
Computervision GIS
45 Vogell Road, 7th Floor
Richmond Hill, Ontario
CANADA L4B 3P6

1 Introduction

With growing interest in Geographic Information System (GIS) technology, there has developed a need to provide standard mechanisms to interact with the data so information can be combined in analysis and reported in an easy and efficient manner. From a database perspective, GIS integrates spatial and attribute information. This paper outlines a proposed extension to the SQL standard which includes spatial and object oriented extensions required in the GIS environment. SQL has been chosen as the base to build on, because of its wide spread acceptance in the current mainstream information processing environments. The SQL standard already defines many Information Systems facilities that are needed in GIS, e.g. transaction management, integrity constraints, privileges, a data definition language and a data manipulation language.

SQL with Integrity Enhancement [1, 9] is the basis for the extensions presented in this paper. They are presented in such a way that they can be applied to the new SQL standard [2] that is about to be adopted. The extensions are based on previous work done with the SYSTEM 9 Query Language (S9QL) [21] and the Analytical Tool Box product offerings [4, 6]. The complete grammar for *GIS/SQL* can be found in Ashworth [5].

The sections, Object-Oriented Data Modeling and Extended Transaction Model, outline the needs of GIS. The sections describing the *GIS/SQL* language extension are Object Oriented Extensions, Spatial Extensions, Data Dictionary Extensions, Distributed Processing/Multibase Extensions, Language Extensions and Input, Output and Graphic Display. The language grammar is presented in BNF; please refer to [1] for a description of the notation used in this paper. An Appendix of examples follows Concluding Remarks, a Glossary and References.

2 Object Oriented Data Modeling

2.1 Object Oriented Approach

The real world can be modeled using an object oriented approach. This paradigm supports objects, class definitions, subclassing, methods, inheritance, polymorphism and data encapsulation.

The table definition has been extended to include the components of a class definition. Class definitions are used to define similar classifications of real world phenomena. An object can be modeled as a row in the table.

Class definitions are structured into a class hierarchy that maintains inheritance information. The *Object* class is the root of all object classifications. *Entity* is a subclass of *Object* used to model any phenomena. *Spatial_Entity* is a subclass of *Entity* used to model any spatial phenomena. *Simple* and *Composite* are subclasses of *Spatial_Entity*. This portion of the hierarchy is system defined. The user begins to build the spatial components of their GIS model by subclassing the *Simple* or the *Composite* class definitions. The non-spatial components of their GIS model can be modeled either by subclassing the *Object* class definition or by subclassing the *Entity* class definition if it will be part of a composite directed acyclic graph (DAG) discussed below. Similar models have been developed using entity-relationship models such as [20].

In Figure 1 the user has defined a class definition called *building* to describe a man-made structure. A *building* is a subclass of *Simple*. *Office_building* and *dwelling* are more specialized types of *building*, but still share the same basic properties so they have been subclassed from *building*.

Classes are organized in a DAG. The class definition, *Simple*, is the super class designed to model a single real world spatial phenomenon. The class definition, *Composite*, is the super class used to model entities that are composed of two or more related real world entities. Sometimes this is called the assemble or has-a relationship. Only subclasses of *Entity* may be referenced in the composite DAG as in Figure 2.

In Figure 2 *City_Block* and *Parcel* are subclasses of the *Composite* class definition. *Road, Parcel_Boundary* and *Building* are subclasses of the *Simple* class definition. Class definitions are organized into a directed acyclic graph (DAG) with subclasses of the *Composite* class definition at internal nodes and subclasses of the Simple class definition at terminal points.

Subclasses of the *Spatial_Entity* class definition can be defined to maintain no topology, node topology (0 Cell), linear topology (1 Cell), surface topology (2 Cell), solid topology (3 Cell) or raster information.

A class definition includes a set of attributes and a set of methods. Attributes may be inherited from the super class as well as having new ones defined specifically for the class definition. New attributes cannot redefine the data types of inherited attributes but changes to the attribute

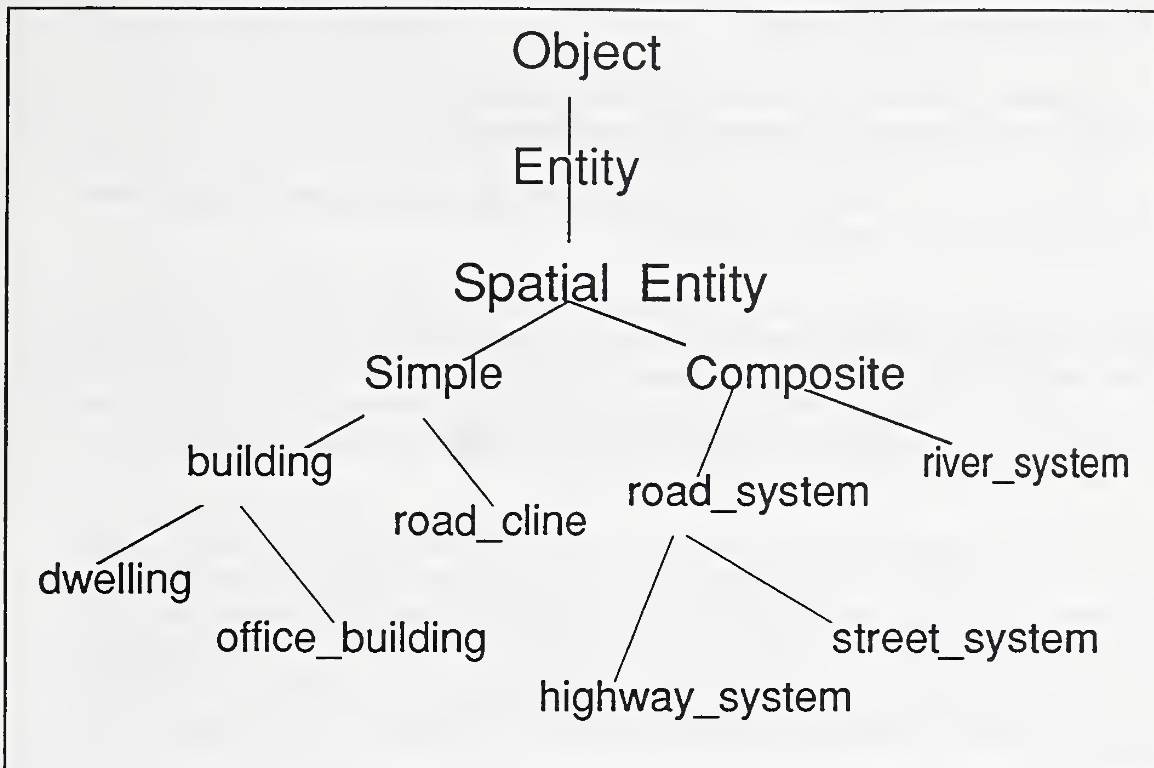


Figure 1. Example of class hierarchy.

constraints are allowed.

In addition to the set of attributes defined in a class definition, a class definition has a set of methods that an object in the class will respond to. A subclass will inherit all the attributes and methods defined by the parent (or superclass). A subclass can define new attributes and methods. The behavior of a subclass's method can be changed by redefining the method in the subclass definition. This is referred to as polymorphism.

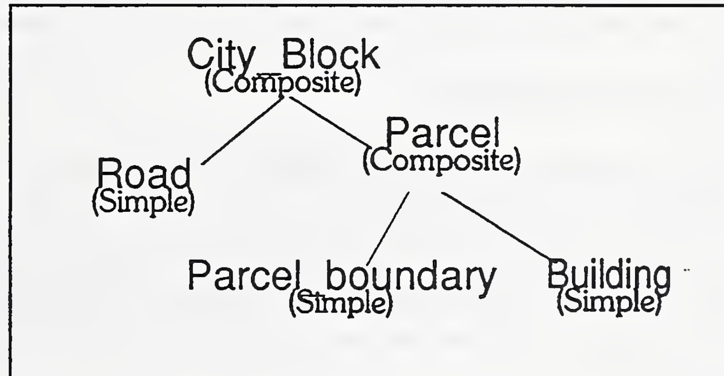


Figure 2. Example of class definitions organized as acyclic directed graph (DAG).

2.2 Extensibility

This paper outlines some ways to extend the GIS/SQL language be the ability to define new methods, operators and abstract data types (ADTs). New operators and methods may be

defined to work on a specific ADT.

2.2.1 Operators. Operators perform monadic or dyadic operations between one or two <value expression> and the results can be used in <value expression>.

2.2.2 Functions. Functions perform some work with a set of parameters. The result of the function can be used in <value expression>.

2.2.3 Abstract Data Types. New data types called ADTs can be created from the basic set of data types (CHAR, INTEGER, etc.) or from other existing ADTs. Methods can be defined to work with the new data types. Existing operators may be redefined to work with an ADT; this is commonly referred to as operator overloading. In addition to redefining existing operators, new operators may be declared. A constructor is used to generate a literal for a given ADT. The result can be treated as a <value specification>.

An ADT called GEOMETRY has been added to SQL for maintaining the geometric or both geometric and topologic information. Many of the operators and functions in this paper have their roots in topologic relationships between features [8]. If topological relationships are not being maintained but are required for a spatial operator, spatial function or spatial predicate, then they will be derived during the query processing phase of the request. This would have to be done if the data being analyzed came from separate systems not supporting the topologic model [16].

3 Extended Transaction Model

This section briefly outlines some specific aspects of transactions in the GIS environment.

3.1 Long Transactions

A long transaction has all the properties of a transaction as defined in [1] except for its duration. This transaction may be active for weeks or months. A typical application using this type of transaction is found in a municipal planning department where a new subdivision is planned or site changes need an approval process. The changes must be approved before being finally committed to the database. If the changes are approved then the changes are committed to the database, otherwise they are rolled back. Sometimes this work is done by contract work off site. Thus the system must support the ability to execute the transactions from remote systems.

3.2 High Volume Read Operations

A typical business application may read up to one hundred records in an operation to fill a textual screen with data. By contrast, a typical GIS graphic display will have up to ten thousand entities in a single spatial area that is displayed. Similarly adding a new subdivision to the database involves a large numbers of update operations. The database management system must be tuned to handle this type of behavior.

3.3 Increased Data Interaction

Maintaining the topologic relationships among entities does increase the amount of work to perform even a simple operation. For example, a single line may be used to define a land parcel boundary, a forest boundary, a lake shoreline and a political boundary. To update this line, there is an overhead in checking the privileges on all four classes.

3.4 Privileges on Spatial Area

There is a need to have privileges based on a spatial criterion. Metro Toronto for example, is made up of five municipalities. There is a need to share information between the centres but some data within the corporate boundary may only be updated by each centre. The ideal solution is to have the capability to define privileges based on spatial area.

4 Object Oriented Extensions

This section describes the object oriented extensions for GIS/SQL.

4.1 Inheritance

The <isa clause> specifies a super class where the attributes and methods are inherited from in this class. Many <isa clause>s can be used in a single table definition to specify multiple inheritance.

```
<table element> ::=
                        <isa clause>
<isa clause> ::=
                        ISA <table name>
```

The <table name> in the <isa clause> is the super class and must already exist.

4.2 Composite Specification

The <composed of clause> specifies the classes that may be used to assemble elements in class being defined.

```
<table constraint definition> ::=
                        <composed of clause>
<composed of clause> ::=
                        COMPOSED [OF] <composed of item>
<composed of item> ::=
                        <table name>
                   |    ( <table name> [{, <table name> }...] )
```

The <table name> in <composed of item> must be a subclass of Entity.

4.3 Composite Predicate

The <composite predicate> enables the user to specify a predicate based on the composite relationship.

```
<predicate> ::=
                    <composite predicate>
<composite predicate> ::=
                    <value expression> [ NOT ] <composite op>
                        { <value expression> | <subquery> }
<composite op> ::=
                    REFERS [TO]
                |   REFERRED [BY]
```

The operator constructors are:

REFERS [TO]	returns TRUE if the LHS entity composes the RHS entity.
REFERRED [BY]	returns TRUE if the LHS entity is used to compose the RHS entity.

4.4 Abstract Data Type Support

The <create adt statement> can be used to create a new data type. This data type can be used by any subsequent <create table statement>'s.

```
<create adt statement> ::=
                    CREATE ADT <adt name> <create adt body>
<create adt body> ::=
                    ( <adt item> [{, <adt item> }...] )
<adt item> ::=
                    STRUCTURE ( <data type item> [{, <data type item> }...] )
                |   FORMAT <value expression>
                |   PARAMETERS ( <data type> [{, <data type> }...] )
                |   ISA <adt name>
                |   <linker clause>
                |   <method clause>
                |   <operator definition>
                |   <constructor definition>
<data type item> ::=
                    <data type>
                |   PRIVATE <data type>
                |   VARIABLE
```

The <data type item> is used to specify the structure of the abstract data type. The VARIABLE keyword may be used for variable length data types like the GEOMETRY ADT (Section 5.1) The FORMAT directive can be used to specify how to display the data type. PARAMETERS specifies the values required to define the ADT attribute. The ISA directive lets the user specify a sub type of another type.

The <linker clause> defines a loader module containing entry points for functions, operators, constructors and methods that can be loaded dynamically or installed as part of an installation procedure. The <value expression> is system and implementor specific.

```
<linker clause> ::=
                    LINKER <value expression>
```

In the UNIX environment, the <value expression> may be a string containing an object file such as - LINKER "/GIS/adt/my_functions.o"

4.5 Method Specification

A <method clause> defines a method for a class or ADT. A <linker clause> and <method clause> can be specified in the <table element> of a class definition. The previous section specified a <method clause> in the <adt item>.

```
<table element> ::=
                    <linker clause>
                  | <method clause>
<method clause> ::=
                    METHOD <method name> ( <method parameters> , <method returns> ,
                        <entry point> [ , <linker clause> ] [ , <method error> ] )
<method parameters> ::=
                    PARAMETERS ( <data type list> [ , .. ])
<data type list> ::=
                    <data type> [{, <data type> }...]
<method returns> ::=
                    RETURNS ( <data type> )
<method error> ::=
                    ON ERROR <value expression>
```

A <method parameter> list can end with a '..' for variable length argument lists. The <method parameters> define the data types used as parameters to the method. The <method returns> defines the data type of the returned value of the method. If it is not specified, the method does not return a value. The <value expression> in <method error> must result in an integer. It is used for error handling in Embedded SQL.

The <entry point> defines the label in the linker module. Like the <linker clause> , the <value expression> is system and implementor specific.

```
<entry point> ::=
                    ENTRY <value expression>
```

In the UNIX environment, the <value expression> may be a string containing an entry point: ENTRY "my_method".

4.6 Operator Definition

The <operator definition> is used to overload existing operators for an ADT. It is specified in the <adt item> when defining an ADT.

```
<operator definition> ::=
                FUNCTION <function name> ( PARAMETERS ( <data type>
                    [{, <data type> }...] ), RETURNS ( <data type> ), <entry point>
                    [ , <linker clause> ] [ , ON ERROR <value expression> ] )
            |   OPERATOR <operator>
                    <entry point> [ <linker clause> ]
```

The OPERATOR directive will overload the operator and the FUNCTION directive creates a function specific to the ADT.

4.7 Constructor Definition

A <constructor definition> how to specify a literal for an ADT.

```
<constructor definition> ::=
                CONSTRUCTOR <constructor name> ( PARAMETERS ( <data type>
                    [{, <data type> }...] ) , RETURNS ( <data type> ) ,
                    OPTION ( <option list> ) , <entry point>  [ , <linker clause> ]
                    [ , ON ERROR <value expression> ] )
<option list> ::=
                <constructor name>
            |   <constructor name> +
            |   <constructor name> *
            |   <constructor name> ( <unsigned integer> [ , <unsigned integer> ] ).
```

Constructors may be specified recursively. The <option list> specifies the limits for the number of constructors at a given level. A + following the constructor's name indicates one or more may be specified; a * specifies zero or more and two integers specify an allowable range.

4.8 Built In ADT Management Functions

The <adt management function specification> provides some facilities to extract data values from an ADT.

```
<adt management function specification> ::=
                        <adt management function key word> <value expression list>
<adt management function key word> ::=
                                NUMBEROF
                            |   EXTRACT
<value expression list> ::=
                ( <value expression> [{ , <value expression }...] )
```

The function definitions in <adt management function key word> are:

NUMBEROF returns the number of elements in an ADT value. The first parameter is the ADT value. The second parameter is the type of element.

EXTRACT extracts a given element from an ADT value. The first parameter is the ADT value. The second parameter is the name of element. The optional third parameter is the name of the element to extract. The default is the first element.

4.9 Object Identity

The GENERATED keyword applies to only single primary key of integer type. This will cause a unique key to be generated automatically. This is used to give object an identity automatically and is useful when there is no attribute value to uniquely identify the spatial entity.

```
<unique specification> ::=
                        [ GENERATED ] UNIQUE
                |       [ GENERATED ] PRIMARY KEY
<unique column list> ::=
                        <column name> [{, <column name> }...]
```

4.10 Extended Data Types

To use the extended object oriented data management facilities, changes must be made to <data type>.

```
<data type> ::=
                        <temporal type>
                |       <extended type>
<temporal type> ::=
                        ANSI SQL 3.135-199X Compatible [2].
<extended type> ::=
                        <adt type>
                |       <method type>
                |       <formula type>
<adt type> ::=
                        <adt name> [ ( <value expression> [{, <value expression> }...] ) ]
<method type> ::=
                        <method name> ( <column name> )
<formula type> ::=
                        FORMULA ( <value expression> )
```

The <adt type> enables the user to create a column in a table with a previously defined data type. Virtual columns can also be created with the <function type> or <formula type>. The <method type> enables the user to create a column that comes from the results of executing a

method on an ADT in another column. In <formula type>, an expression is specified. This expression is evaluated when the column is referenced. Columns created with either <method type> or <formula type> cannot be updated and may be implemented with a query rewrite facility [23]. Columns of this type are useful in definition of information in a graphical display.

4.11 Values and Targets

The <value specification> and <target specification> have been extended to allow the specification of constructors to be used as literals, functions, class methods and ADT methods.

```
<value specification> ::=
                        <constructor specification>
                    |   <function specification>
                    |   <class method specification>
                    |   <adt method specification>
 <function specification> ::=
                        <spatial function specification>
                    |   <adt management function specification>
                    |   <generic function specification>
<generic function specification> ::=
                        <function name> <value expression list>
<class method specification> ::=
                        <table name> :: <method name> <value expression list>
<adt method specification> ::=
                        <column specification> : <method name> <value expression list>
<constructor specification> ::=
                        <temporal constructor>
                    |   <geometry constructor>
                    |   <style constructor>
                    |   <generic constructor>
<generic constructor> ::=
                        <adt name> : <constructor name>
                              [ ( <constructor item>
                              [{, <constructor item> }...] ) ]
<constructor item> ::=
                        <value expression>
                    |   <coordinate list>
                    |   <coordinate>
                    |   <subquery>
<temporal constructor> ::=
                        ANSI SQL 3.135-199X Compatible [2].
```

In the <class method specification> and the <adt method specification>, the values in the <value expression list> must result in the appropriate data type as specified in the method definition.

The <geometry constructor> is defined in Section 5.3 and the <style constructor> is defined in Section 9.5.

5 Spatial Extensions

5.1 Geometry ADT

The geometry ADT provides facilities to manage geometric information and topologic relationships.

<data type> ::=
<geometry type>
| GEOMETRY [(<geometry type item> [{, <geometry type item> }...]
<geometry type item> ::=
<dimensionality>
| <integrated>
<dimensionality> ::=
SOLID
| SURFACE
| LINEAR
| NODE
| RASTER
| <value expression> CELL
<integrated> ::=
[NOT] INTEGRATED

5.2 Coordinate Constructor

A <coordinate> is used to specify a position in space and often time. A <coordinate list> is an ordered list of two or more <coordinates>.

<coordinate> ::=
(<x_val>, <y_val> [, <z_val>][, <time>])
<x_val> ::=
<value expression>
| NULL
<y_val> ::=
<value expression>
| NULL
<z_val> ::=
<value expression>
| NULL
<time> ::=
<value expression>
| NULL
<coordinate list> ::=
(<coordinate> {, <coordinate> }... [, <z>][, <time>])

Coordinates may be specified in any combination of 2d, 3d, 2d/time or 3d/time. The <time> specification in the <coordinate list> will apply to all coordinates in the list that do not have a <time> specified. The <z_val> specification in the <coordinate list> will apply to all

coordinates in the list that do not have a <z_val> specified.

5.3 Geometry Constructor

The <geometry constructor> is used to specify <literal>'s of type GEOMETRY.

```
<geometry constructor> ::=
                              <geometry constructor item>
                        |     AMALGAMATE ( <geometry constructor item>
                                  [{, <geometry constructor item> }...] )
<geometry constructor item> ::=
                              <node constructor>
                        |     <linear constructor>
                        |     <directed linear constructor>
                        |     <surface constructor>
                        |     <solid constructor>
                        |     <triangle constructor>
                        |     <raster constructor>
                        |     <eaf constructor>
                        |     <centroid>
<shared body> ::=
                              SHARED <subquery>
```

The <shared body> specifies one or more objects to build up topology. AMALGAMATE enable multiple geometries to be grouped into one. This is useful when an entity may be split into more than one piece.

<u>5.3.1 Node Constructor</u>. A <node constructor> specifies a single point.

```
<node constructor> ::=
                              NODE ( [ <node constructor item> [{, <node constructor item> }...] )
<node constructor item> ::=
                              COORDINATE <coordinate>
                        |     <shared body>
```

<u>5.3.2 Linear Constructor</u>. A <linear constructor> specifies a line. There are a number of linear types supported: simple polyline (line), curves and arcs.

```
<linear constructor> ::=
                              <line constructor>
                        |     <curve constructor>
                        |     <arc constructor>
                        |     LINEAR ( [ <linear constructor item>  [{, <linear constructor item> }...] )
<linear constructor item> ::=
                              <line constructor>
                        |     <curve constructor>
                        |     <arc constructor>
                        |     START_NODE <subquery>
                        |     END_NODE <subquery>
```

```
                        |   <shared body>
<line constructor> ::=
                            LINE <coordinate list>
                        |   RECTANGLE <coordinate list>
<curve constructor> ::=
                        |   BSPLINE <coordinate list>
                        |   BEZIER <coordinate list>
<tangent> ::=
                            <value expression>
<arc constructor> ::=
                            ARC <coordinate list>
                        |   ARC ( <center>, <radius> , <start direction>, <end direction> )
                        |   CIRCLE ( <center>, <radius> )
<center> ::=
                            COORDINATE <coordinate>
<radius> ::=
                            <value expression>
<start direction> ::=
                            <value expression>
<end direction> ::=
                            <value expression>
```

The <line constructor> is a simple polyline consisting of a starting point, zero or more intermediate points and an end point.

The <curve constructor> is a parametric cubic curve definition. BEZIER type curves are defined as Bezier curves [12] p. 488. BSPLINEs are defined as Uniform Non Rational B-spline curves [12] p. 491. NURB and MESH types may be added in the future. The <value expression> in <tangent> is an angle.

The <arc constructor> is a segment of circle. Arcs can be specified by three coordinates that are not collinear. An alternate way of specifying an ARC is by a center point, radius and two angles. This specification will be extended to include elliptical arcs.

5.3.3 Directed Linear Constructor. The direction of a line is from the first point to the last point in the coordinate list. The REVERSED directive in <directed linear constructor> will cause the line direction to be from the last point to the first point.

```
<directed linear constructor> ::=
                            <linear constructor>
                        |   REVERSED ( <linear constructor> )
```

5.3.4 Surface Constructor. The <surface constructor> is used to specify polygons, or polygons with islands or holes.

```
<surface constructor> ::=
                            SURFACE ( <surface constructor body> [{, <island body> }...] )
<island body> ::=
                            ISLAND_SURFACE ( <surface constructor body> )
```

```
<surface constructor body> ::=
                    <surface constructor item> [{, <surface constructor item> }...]
<surface constructor item> ::=
                    <directed linear constructor>
            |       <shared body>
```

In the <surface constructor> the exterior surface may be created from one or more <line constructors>. A surface may have zero or more <island constructors>. The islands are created with <surface constructor body>.

Surface geometries have the exterior boundary formed in a clockwise fashion. The island boundaries are formed in a counter-clockwise fashion (i.e. the right hand rule). The linear geometries may have their direction logically reversed to maintain this property.

5.3.5 Solid Constructor. The <solid constructor> specifies a solid geometry. Research is currently focused in the area of 3D topologic models [19].

```
<solid constructor> ::=
                    SOLID ( <solid constructor body> [{, <hollow body> }...]
<hollow body> ::=
                    SOLID_HOLLOW ( <solid constructor body> )
<solid constructor body> ::=
                    <solid constructor item> [{, <solid constructor item> }...]
<solid constructor item> ::=
                    <surface constructor>
            |       <shared body>
```

5.3.6 Triangle Constructor. The <triangle constructor> is used to specify a triangular irregular network (TIN). A set of triangles can be used to model a surface in 3D. Digital Elevation Modeling can be down using the RASTER constructor (Section 5.3.7). In this case, the value of each cell would contain the z coordinate.

```
<triangle constructor> ::=
                    TRIANGLE <coordinate list>
            |       TRIANGLE ( <triangle constructor item>
                          [{, <triangle constructor item> }...] )
<triangle constructor item> ::=
                    LINE <coordinate list>
            |       ADJACENT_TRIANGLE <subquery>
            |       TRIANGLE_NODE <subquery>
            |       <shared body>
```

In the <triangle constructor>, 3 points in a coordinate list are specified. The coordinates cannot be collinear.

5.3.7 Raster Constructor. The <raster constructor> is used to specify a raster image.

```
<raster constructor> ::=
                    RASTER ( <raster constructor item> [{, <raster constructor item> }...] )
              |     PIXEL <value expression list>
<raster constructor item> ::=
                    ORTHOGONAL_RASTER ( <value expression> )
              |     RAW_RASTER_DATA ( <value expression> )
              |     RASTER_TRANSFORM ( <value expression>
              |     RASTER_FORMAT ( <value expression> )
              |     RASTER_SIZE <value expression list>
              |     RASTER_LOCATION <coordinate>
              |     RASTER_RESOLUTION <value expression list>
              |     RASTER_NPLANES ( <value expression> )
              |     RASTER_DEPTH ( <value expression> )
              |     <surface constructor>
              |     <colour constructor>
              |     <shared body>
```

The RASTER_SIZE directive contains the size of the image in the x, y and optional z axis. The <coordinate> in the RASTER_LOCATION directive contains the x, y and optional z elements for the location of the image. The RASTER_RESOLUTION directive contains the pixel size of the image in the x, y and optional z axis. The <value expression> in the RASTER_FORMAT directive defines the type of raster image such as ORTHOGONAL_RASTER, RAW_RASTER_DATA either contain the raster data or reference a file containing the raster information. The data should preferably be copied into the database so that all the transaction management facilities are available.

5.3.8 EAF Constructor. An Edit Attention Flag (EAF) is used to show the location of a potential data problem area. They are typically used in the data cleanup phase of data capture.

```
<eaf constructor> ::=
                    EAF ( <eaf constructor item>
                          [{, <eaf constructor item> }...] )
<eaf constructor item> ::=
                    EAF_LOCATION <coordinate list>
              |     EAF_CAUSE ( <value expression> )
              |     EAF_CODE ( <value expression> )
              |     EAF_DESCRIPTION ( <value expression> )
              |     <shared body>
```

5.3.9 Centroid. The centroid is used to specify a single location for an entity. This is primarily used as a location to display attribute values for a spatial entity.

```
<centroid> ::=
                    CENTROID <coordinate>
```

5.4 Between Predicate

The <between predicate> has been extended to handle GEOMETRY typed attribute. It will return TRUE if the first <value expression> is spatially between the second <value expression> and third <value expression>.

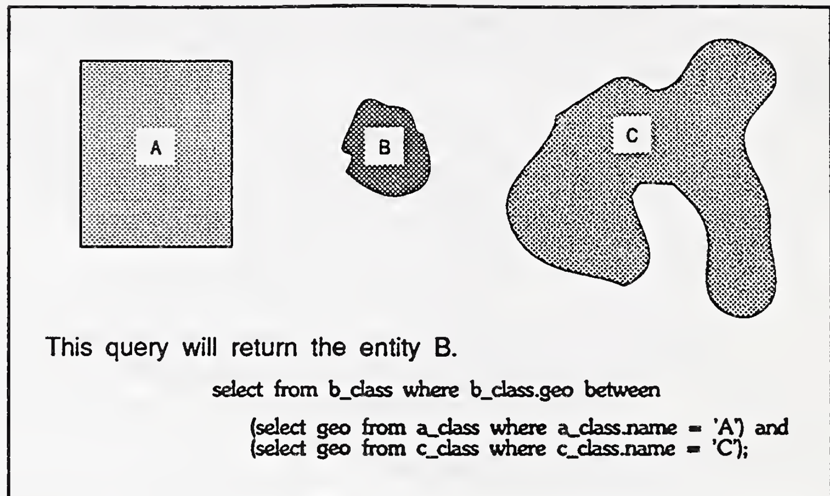


Figure 3. Concept of <between predicate>.

5.5 Spatial Predicates

The <spatial predicate> specifies a set of spatial predicates for use between values.

```
<predicate> ::=
                      |    <spatial predicate>
<spatial predicate> ::=
                           <value expression> [ NOT ] <spatial op>
                                 { <value expression> | <subquery> }
<spatial op> :: =
                           <spatial op key word> [ <spatial op parameters> ]
<spatial op parameters> ::=
                           PARAMETER <value expression list>
                      |    PARAMETERS <value expression list>
<spatial op key word> ::=
                           [AT] START [OF] | [AT] END [OF] | CONNECTED [TO]
                      |    OVERLAPS | [IS] OUTSIDE [OF]
                      |    CONTAINS | [IS] CONTAINED [BY]
                      |    [IS] ADJACENT [TO]
                      |    [IS] <direction> [OF]
                      |    [IS] LEFT [OF] | [IS] RIGHT [OF]
                      |    [IS] NEAREST [FROM] | [IS] FARTHEST [FROM]
<direction> ::=
                           NORTH | SOUTH | EAST | WEST
                      |    NORTH EAST | NORTH WEST | SOUTH EAST | SOUTH WEST
                      |    EAST NORTH | EAST SOUTH | WEST NORTH | WEST SOUTH
                      |    ABOVE | BELOW | DIRECTION
```

Definitions of predicates follows. Note that not all conditions or assumptions are included in this paper.

Formal discussion of many spatial predicates may be found in [8] and [11]. The predicates work on columns of type GEOMETRY. A full implementation should handle all geometric types ((n) node, (l) linear, (s) surface, (o) solid) defined for predicate. Allowable types are specified as a series of letters separated by or bars (|) in parentheses. Additional predicates are defined in [24]

[AT] END [OF] — returns TRUE if the LHS entity (n|l|s|o) is at the end of the RHS entity (l). An optional tolerance parameter may be specified.

CONNECTED [TO] — returns TRUE if the LHS entity (n|l|s|o) is connected to the RHS entity at a point (l). An optional tolerance parameter may be specified.

OVERLAPS — returns TRUE if the LHS entity (n|l|s|o) overlaps the RHS entity (n|l|s|o). An optional tolerance parameter may be specified.

[IS] OUTSIDE [OF] — returns TRUE if the LHS entity (n|l|s|o) does not overlap the RHS entity (n|l|s|o). An optional tolerance parameter may be specified. An optional tolerance parameter may be specified.

CONTAINS — returns TRUE if the LHS entity (s|o) contains the RHS entity (n|l|s|o). An optional tolerance parameter may be specified.

[IS] CONTAINED [BY] — returns TRUE if the LHS entity (n|l|s|o) is contained by the RHS (s|o). An optional tolerance parameter may be specified.

[IS] ADJACENT [TO] — returns TRUE if the LHS entity (n|l|s|o) is adjacent to the RHS entity (n|l|s|o) by sharing a point or line segment. An optional tolerance parameter may be specified.

[IS] <direction> [OF] — returns TRUE if the LHS entity (n|l|s|o) is in the direction from the specified RHS entity (n|l|s|o). A 45 degree pie shape is used that is centered on the direction (22.5 degrees either sided) to determine the result. The DIRECTION predicate has the azimuth as a parameter. An optional parameter may be specified to change the default 45 angle to the specified value.

[IS] LEFT [OF] — uses the line direction to return TRUE if the LHS entity (s|o) is to the left of the RHS entity (l).

[IS] RIGHT [OF] — uses the line direction to return TRUE if the LHS entity (s|o) is to the right of the RHS entity (l).

[IS] NEAREST [TO] — returns TRUE if the LHS entity (n|l|s|o) is the nearest to the RHS entity (n|l|s|o).

[IS] FARTHEST [FROM] — returns TRUE if the LHS entity (n|l|s|o) is furthest from the RHS entity (n|l|s|o).

5.6 Spatial Functions

The <spatial function specification> specifies a set of spatial functions.

<spatial function specification> ::=
<spatial function key word>
<value expression list>

<spatial function key word> ::=
EXTENT
| MER
| BUFFER
| DETERMINE_CENTROID
| SKELETON
| LENGTH
| AZIMUTH
| SLOPE
| AREA
| PERIMETER
| VOLUME
| TRANSFORM
| OVERLAP
| CONTAIN
| ADJACENT
| SEPARATION
| GEOMETRY_UNION
| AXIS_PROJECT
| THIESSEN_POLYGON

The function definitions for <spatial function key word> are:

EXTENT — returns a surface geometry that is the convex polygon that would contain the specified geometry (n | l | s).

MER — returns a minimum enclosing rectangle or box that would contain the specified geometry (n | l | s | o). An optional parameter may specify the dimensionality: 2D for a rectangle or 3D for a box.

BUFFER — returns a geometry representing an expanded region around the given geometry (n | l | s | o). The required second parameter is the width of the expansion. An optional third parameter is the filtering tolerance and optional forth parameter is the dimension of the calculation: 2D or 3D.

DETERMINE_CENTROID — returns the node geometry representing the centroid of the given geometry. If the centroid is not specified in the geometry, it is calculated. An optional third parameter is the dimension of the calculation: 2D or

3D.

SKELETON	returns the linear geometry representing the skeleton of the given geometry (s l o).
LENGTH	returns the length of the given geometry (l l s). The optional second parameter specifies the dimension of the calculation: 2D or 3D.
AZIMUTH	returns the bearing (azimuth) of the given linear geometry. The optional second parameter specifies degrees, radians or grads.
SLOPE	returns the slope of the given 3D geometry (l l s).
AREA	returns the area of the given geometry (s l o).
PERIMETER	returns the exterior perimeter length of the given geometry (s). The optional second parameter specifies the dimension of the calculation: 2D or 3D
VOLUME	returns the volume of the given geometry (o).
TRANSFORM	transforms a given geometry through a given transformation. The second parameter is the transform to perform (See section 6.6.3). The third parameter specifies the direction of the transformation: 0 is from the base to the target and 1 is from the target to the base.
OVERLAP	returns a geometry of the overlapping portion of the two specified geometries (n l l l s l o). NULL is returned if there is no overlap. Boundaries are considered part of the entity.
CONTAIN	returns a geometry of the first geometric entity (n l l l s l o) if the first geometry is contained in the second geometric entity (s l o). NULL is returned otherwise. Boundaries are considered part of the entity.
ADJACENT	returns a geometry of the adjacent portion of the two specified geometries (n l l l s l o). NULL is returned if the two geometries are not adjacent. An optional third parameter can specify node or linear results. Boundaries are considered part of the entity.
SEPARATION	returns the shortest distance between the two given geometries (n l l l s l o).

GEOMETRY_UNION — unites the specified geometries (n|l|s|o) into one geometry.

AXIS_PROJECT — returns a geometry representing the geometry projected on the given axis. The optional second parameter is the axis specification. It is a string containing "x", "y", "z" or "t". The default is "xy". The values can be swapped by changing the order of "x", "y" or "z". There is no order in the time dimension.

THIESSEN_POLYGON — returns geometries representing the thiessen polygons around the set of specified geometries (n).

6 Data Dictionary Extensions

The following data dictionary extensions are presented to maintain more details about the geographic information in the database (project), definitions of long transactions (partitions), display styles, (styles), thematic display (theme, dynamic theme), transformations (ellipsoid, projection, transformation, control point, orientation) and accuracy. These extensions are maintained in database tables that can be controlled by the privilege mechanism.

6.1 Project

The project defines the spatial extent for the database. The definition includes the transformation, unit system, themes and class definitions used in the GIS project.

```
<project management statements> ::=
                    CREATE PROJECT <project name> <project body>
                |   ALTER PROJECT <project name> <alter project>
                        [{, <alter project> }...]
                |   DROP PROJECT <project name>
                |   COPY PROJECT <project name> TO <project name>
                |   RENAME PROJECT <project name> TO <project name>
                |   DISPLAY PROJECT [ <project name> ]
<project body> ::=
                    ( <project item> [{, <project item> }...] )
<project item> ::=
                    <surface constructor>
                |   UNIT [ METRIC | IMPERIAL ]
                |   HOST <value expression>
                |   PORT <value expression>
                |   LOCATE IN <value expression>
                |   TRANSFORMATION <transformation name>
                |   [ NOT ] COMPRESSED
<alter project> ::=
                    ADD <project item>
                |   CHANGE <project item>
                |   DELETE <project item>
```

6.2 Partition

A partition defines a subset of classes and attributes that may be used in a long transaction. The long transaction is started when the partition is checked out of the project. At this time the partition may be used locally or moved to a remote site. At the remote site, the partition must contain all the information needed because the project may not be accessible. The long transaction is committed when the partition is checked in.

```
<partition management statements> ::=
                    CREATE PARTITION <partition name> <partition body>
                |   ALTER PARTITION <partition name> <alter partition>
                                        [{, <alter partition> }...]
                |   DROP PARTITION <partition name>
                |   COPY PARTITION <partition name> TO  <partition name>
                |   RENAME PARTITION <partition name> TO  <partition name>
                |   DISPLAY PARTITION [ <partition name> ]
<partition body> ::=
                    ( <partition item> [{, <partition item> }...] )
<partition item> ::=
                    <project item>
                |   CLASS <table name>  |  CLASS <table name> . <column name>
                |   CLASS <table name ( <column name>  [{, <column name> }...] )
<alter partition> ::=
                    ADD <partition item>
                |   CHANGE <partition item>
                |   DELETE <partition item>
```

The checkout or checkin of a partition is handled by the following statement.

```
<cico statement> ::=
                    CHECK { IN|OUT } <db name>
```

A check out command starts a long transaction. A check in will commit the long transaction. The <db name> must contain a partition specification that it defined.

6.3 Style

A style defines the display characteristics for spatial and attribute information. A fill style defines how surface primitives are displayed. A linear style defines how linear primitives are displayed. A symbol style defines how node primitives are displayed. An annotation style defines how to display an attribute value. The style constructor is defined in Section 9.5.

```
<style management statements> ::=
                    CREATE STYLE <style name> <style constructor>
                |   DROP STYLE <style name>
                |   COPY STYLE <style name> TO <style name>
                |   RENAME STYLE <style name> TO <style name>
                |   DISPLAY STYLE [ <style name> ]
```

6.4 Theme

A theme defines one way to graphically display attributes in a set of tables. It contains a set of thematic display elements that match display styles with a set of entities in a class. A default map scale and masking order definition is provided primarily for physical output devices (for example plotters).

```
<create theme statement> ::=
                        CREATE THEME <theme name> <create theme body>
                    |   ALTER THEME <theme name> <create theme command>
                            [{, <create theme command> }...]
                    |   DROP THEME <theme name>
                    |   COPY THEME <theme name> TO <theme name>
                    |   RENAME THEME <theme name> TO <theme name>
                    |   DISPLAY THEME [ <theme name> ]
<create theme body> ::=
                        ( <create theme item> [{, <create theme item> }...] )
<create theme item> ::=
                        <table name>
                    |   <table name> . <column name>
                    |   <table name ( <column name> [{, <column name> }...]
                    |   MAPSCALE ( <value expression> )
                    |   MASKING [ <create theme where> ] <create theme set>
<create theme where> ::=
                        OVER <table name>
                    |   UNDER <table name>
<create theme set> ::=
                        <create theme group>
                    |   ( <create theme group> [{, <create theme group> }...] )
<create theme group> ::=
                        ( <table name> [{, <table name> }...] )
<alter theme command> ::=
                        ADD <create theme item>
                    |   CHANGE <create theme item>
                    |   DELETE <create theme item>
```

A table name without a column specification will, by default, include all attributes for the table.

The item MASKING will define one masking set. Its location in the masking order is determined by the <create theme where> clause. Any class not in the MASKING clause will default to a set at the end of the masking order.

The OVER directive indicates that the masking group will mask everything in the group that contains the <table name> and any groups lower in the list. The UNDER directive indicates that the masking group will be masked by everything in the group that contains the <table name> and any groups higher in the list.

The MAPSCALE is used as the default scaling factor for physical devices.

6.5 Thematic Display

A thematic display controls the style of a specific entity being displayed, based on the current theme, current scale and the constraint specification (<search condition>).

<thematic display management statements> ::=
CREATE THEMATIC DISPLAY <thematic name> <create thematic body>
| ALTER THEMATIC DISPLAY <thematic name> <create thematic command>
[{, <create thematic command> }...]
| DROP THEMATIC DISPLAY <thematic name>
| COPY THEMATIC DISPLAY <thematic name> TO <thematic name>
| RENAME THEMATIC DISPLAY <thematic name> TO <thematic name>
| DISPLAY THEMATIC DISPLAY [<thematic name>]
<create thematic body> ::=
(<create thematic group item> [{, <create thematic group item> }...])
<create thematic item> :=
FOR <column specification> <style constructor>
| IN <theme name>
| BEFORE <thematic name>
| AFTER <thematic name>
| <where clause>
<alter thematic command> ::=
ADD <create thematic item>
| CHANGE <create thematic item>
| DELETE <create thematic item>

Only one thematic definition with no <search condition> is allowed for each unique pair of <table name> and <theme name>. This construct is used to define the default style for the class in the theme. The BEFORE and AFTER directives control the display priority.

6.6 Transformation

A transformation consists of a group of extensions to manage the transformation and projection information for the spatial data.

<u>6.6.1 Ellipsoid Management</u>. An ellipsoid is required by NOAA GCTP/II transformation package [17].

<ellipsoid management statements> ::=
CREATE ELLIPSOID <ellipsoid name> <create ellipsoid body>
| ALTER ELLIPSOID <ellipsoid name> <alter ellipsoid command>
[{, <alter ellipsoid command> }...]
| DROP ELLIPSOID <ellipsoid name>
| COPY ELLIPSOID <ellipsoid name> TO <ellipsoid name>
| RENAME ELLIPSOID <ellipsoid name> TO <ellipsoid name>
| DISPLAY ELLIPSOID [<ellipsoid name>]

```
<create ellipsoid body> ::=
                    ( <create ellipsoid item> [{, <create ellipsoid item> }...] )
<create ellipsoid item> ::=
                    SEMIMAJORAXIS ( <value expression> )
                |   ECCENTRICITYSQUARED ( <value expression> )
<alter ellipsoid command> ::=
                    ADD <create ellipsoid item>
                |   CHANGE <create ellipsoid item>
                |   DELETE <create ellipsoid item>
```

<u>6.6.2 Projection Management</u>. A projection specifies a projection system in the NOAA GCTP/II transformation package [17].

```
<projection management statements> ::=
                    CREATE PROJECTION <projection name> <create projection body>
                |   ALTER PROJECTION <projection name> <alter projection command>
                        [{, <alter projection command> }...]
                |   DROP PROJECTION <projection name>
                |   COPY PROJECTION <projection name> TO <projection name>
                |   RENAME PROJECTION <projection name> TO <projection name>
                |   DISPLAY PROJECTION [ <projection name> ]
<create projection body> ::=
                    ( <create projection item> [{, <create projection item> }...] )
<create projection item> ::=
                    ELLIPSOID <ellipsoid name> [ <create ellipsoid body> ]
                |   PROJECTION <value expression list>
                    PARAMETER <literal> <value expression list>
                |   ZONE ( <value expression> )
<alter projection command> ::=
                    ADD <create projection item>
                |   CHANGE <create projection item>
                |   DELETE <create projection item>
```

The <value expression> for the PROJECTION and PARAMETER item must be an unsigned integer.

<u>6.6.3 Transformation Management</u>. Transforms are used to manage many types of transformations for a project. They may be a projection (BASE PROJECTION, TARGET PROJECTION), non-linear transformation (NONLINEARCELLS, NONLINEARXFORM), linear transformation (SCALE, SHIFT, ROTATION), polynomial transformation by coefficients (POLYNOMIAL) and control points (CONTROL). The orientation can be changed within a transformation by the BASE ORIENTATION and TARGET ORIENTATION.

```
<transformation management statements> ::=
                    CREATE TRANSFORMATION <transformation name>
                        <create transformation body>
                |   ALTER TRANSFORMATION <transformation name>
                        <alter transformation command>
                        [{, <alter transformation command> }...]
```

| DROP TRANSFORMATION <transformation name>
| COPY TRANSFORMATION <transformation name> TO
<transformation name>
| RENAME TRANSFORMATION <transformation name> TO
<transformation name>
| DISPLAY TRANSFORMATION [<transformation name>]

<create transformation body> ::=
(<create transformation item> [{, <create transformations item> }...])

<create transformation item> ::=
BASE PROJECTION <projection name> [<create projection body>]
| TARGET PROJECTION <projection name> [<create projection body>]
| NONLINEARCELLS (<value expression>)
| NONLINEARXFORM (<value expression>)
| LINEARSCALE (<value expression> [, <value expression>
[, <value expression>]])
| LINEARSHIFT (<value expression> [, <value expression>
[, <value expression>]])
| LINEARROTATION (<value expression>
[, <value expression> [, <value expression>]])
| POLYNOMIAL <value expression list>
| CONTROL <control point item> , <control point item>
| BASE ORIENTATION <orientation name>
| TARGET ORIENTATION <orientation name>

<alter transformation command> ::=
ADD <create transformation item>
| CHANGE <create transformation item>
| DELETE <create transformation item>

<control point item> ::=
<coordinate>
| <control point name>

6.6.4 Control Point Management. The control points are used to calculate a polynomial transformation.

<control point management statements> ::=
CREATE CONTROL POINT <control point name>
<create control point body>
| DROP CONTROL POINT <control point name>
| COPY CONTROL POINT <control point name> TO
<control point name>
| RENAME CONTROL POINT <control point name> TO
<control point name>
| DISPLAY CONTROL POINT [<control point name>]

<create control point body> ::=
<coordinate>

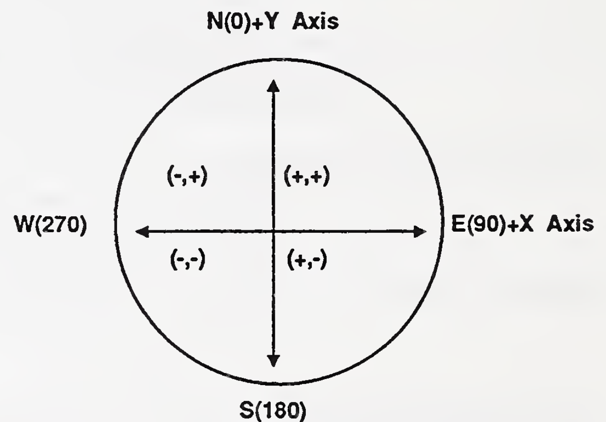
6.6.5 Orientation Management. The orientation controls the coordinate system and azimuth settings.

<orientation management statements> ::=
CREATE ORIENTATION <orientation name> <create orientation body>
| DROP ORIENTATION <orientation name>
| COPY ORIENTATION <orientation name> TO <orientation name>
| RENAME ORIENTATION <orientation name> TO <orientation name>
| DISPLAY ORIENTATION <orientation name>

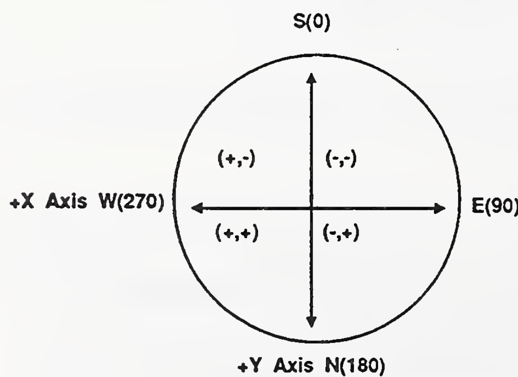
<create orientation body> ::=
(<create orientation item> [{, <create orientations item> }...])

<create orientation item> ::=
X_AXIS (<value expression>)
| Y_AXIS (<value expression>)
| NORTH (<value expression>)
| EAST (<value expression>)
| XY_ORDER
| YX_ORDER

The orientation specification always works in the following system to specify the new system:



If the following is specified in an orientation, X_AXIS(270), Y_AXIS(180), NORTH(180), EAST(90), the resulting orientation of the coordinate system is:



XY_ORDER is used for (x, y) coordinate systems and the default, YX_ORDER is for (y, x) coordinate systems.

6.7 Data Accuracy

The data accuracy for a region can be specified by the survey number, completion date, sigma number and value of sigma for the survey. Additional information can be specified in the description.

```
<accuracy management statements> ::=
                CREATE ACCURACY <accuracy name> <create accuracy body>
        |       DROP ACCURACY <accuracy name>
        |       COPY ACCURACY <accuracy name> TO <accuracy name>
        |       RENAME ACCURACY <accuracy name> TO <accuracy name>
        |       DISPLAY ACCURACY <accuracy name>
<create accuracy body> ::=
                <column name>
        |       SIGMA <value expression>
        |       SIGMA NUMBER <value expression>
        |       SURVEY NUMBER <value expression>
        |       REGION <geometry constructor>
        |       DESCRIPTION <value expression>
        |       COMPLETION <value expression>
```

7 Distributed Processing/Multibase Extensions

The need for distributed systems increases as data volumes grow from year to year and the data management becomes more decentralized [15]. With standards like RDA [3] and SDTS [22] becoming more prominent in the database industry, the need for GIS extensions to SQL so that external database management system can use geographic information transparently, is in high demand.

The management of the distributed parts of a database should be straight forward for the DBA to manage. A new data dictionary extension has been proposed to manage the distributed parts. The fragment may be based on spatial extent (a spatial variant of horizontal fragmentation), set of classes (vertical fragmentation) or a combination of both [18].

```
<fragment management statements> ::=
                CREATE FRAGMENT <fragment name> <fragment body>
        |       ALTER FRAGMENT <fragment name> <alter fragment>
                     [{, <alter fragment> }...]
        |       DROP FRAGMENT <fragment name>
        |       COPY FRAGMENT <fragment name> TO <fragment name>
        |       RENAME FRAGMENT <fragment name> TO <fragment name>
        |       DISPLAY FRAGMENT [ <fragment name> ]
<fragment body> ::=
                ( <fragment item> [{, <fragment item> }...] )
<fragment item> ::=
                <project item>
        |       REPLICATION [ OF ] <fragment name>
        |       CLASS <table name>
        |       CLASS <table name> . <column name>
```

| CLASS <table name (<column name> [{, <column name> }...])

<alter fragment> ::=

ADD <fragment item>
| CHANGE <fragment item>
| DELETE <fragment item>

See Section 6.1 for a definition of <project item>.

An additional mechanism to relocate tables in other locations in the system is necessary to distribute the tables across many partitions. The <locate in clause> is one mechanism that may be used. The <value expression> is implementor-defined.

<locate in clause> ::=

LOCATE IN <value expression>

The <value expression> must result in a valid disk location specification, it is system and implementor specific.

For federated or multibase systems, extensions have been added to allow specification of a table in another system. If the specified database is remote, an implementation of RDA [3] or SDTS [22] could be used to exchange the information.

<table name> ::=

| DB <db name> . <table identifier>
| DB <db name> . <authorization identifier> . <table identifier>

<db name> ::=

<project name>
| <project name> (<partition name>)

8 Language Extensions

8.1 Create Function Statement

The <create function statement> provides an extensible mechanism to create new general purpose functions.

<create function statement> ::=

CREATE FUNCTION <function name> (
PARAMETERS (<data type> [{, <data type> }...]),
RETURNS (<data type>), <entry point>
[, <linker clause>] [, ON ERROR <value expression>])

The value of the ON ERROR directive is set in the SQLCODE on an error in Embedded SQL.

8.2 Create Aggregate Set Function Statement

Beside normal functions, it should be possible to extend the set functions that are available in SQL so that functions like shortest path can be added to the system easily.

```
<distinct set function> ::=
                        {AVG|MIN|MAX|SUM|COUNT|<set function>}
                            ( DISTINCT <column specification> )
<all set function> ::=
                        {AVG|MIN|MAX|SUM|<set function>} ( [ALL] <value expression> )
<set function> ::=
                        <function name>

<create aggregate statement> ::=
                        CREATE AGGREGATE <function name> (
                            PARAMETERS ( <data type> [{ , <data type> }...] ),
                            RETURNS ( <data type> ),
                            ENTRY <value expression list> [ , <linker clause> ]
                                [ , ON ERROR <value expression> ] )
```

This definition is similar to a function definition except that a number of entry points are defined to provide facilities to initialize the set function, start a group, process an attribute value, end a group, and cleanup the set function.

8.3 Indexing

There are no index specification statements in [1] but are a common mechanism used to improve query performance. The <create index statement> and <drop index statement> have been added to control how attributes are indexed. Unique indices should be applied to attributes defined with the <unique constraint definition> automatically. In addition to indexing provided in many common commercial relational database systems, a spatial index can be defined on attributes with the GEOMETRY data type. R-Trees [14, 7] or RT-Trees [13] are types of spatial indexing methods available.

```
<create index statement> ::=
                        CREATE <create index type> INDEX [ <index name> ]
                        ON <create index specification>
<create index type> ::=
                        SPATIAL
                    |   UNIQUE
                    |   ( <value expression> [ , <value expression> ] )

<create index specification> ::=
                        <table name> . <column name>
                    |   <table name ( <column name> [ {, <column name> }...] )
```

SPATIAL indices can only be created on GEOMETRY type attributes. UNIQUE indices cannot be created on attributes that allow NULL values.

8.4 Alter Statement

The data model for a working system must be changed from time to time, as new requirements are placed on the system. It must be possible to modify the database schema while the database is on-line. Schema changes must be performed within the transaction model so that it can be rolled back when necessary. During a historical query, the schema of the table should be presented at the time specified in the query.

```
<alter table statement> ::=
                    ALTER TABLE <table name> <alter table command>
                        [{, <alter table command> }...]
<alter table command> ::=
                    ADD <table element>  [ AFTER <column name> ]
                |   CHANGE <table element>
                |   DELETE <table element>
```

8.5 Drop Statement

When a component in the schema is no longer necessary, it should be dropped from the database using the <drop statement>. Like the alter statement, schema changes must be performed within the transaction model.

```
<drop statement> ::=
                    DROP SCHEMA <schema authorization identifier>
                |   DROP MODULE <module name>
                |   DROP TABLE <table name>
                |   DROP VIEW <table name>
                |   DROP INDEX <index name>
                |   DROP INDEX <create index specification>
                |   DROP ADT <adt name>
                |   DROP CONSTRUCTOR <constructor name>
                |   DROP FUNCTION <function name>
                |   DROP AGGREGATE <function name>
```

8.6 Display Statement

A facility to display information in the data dictionary is provided by the <display statement>.

```
<display statement> ::=
                    <display item>
                        [ ALL ]
                        [ <destination clause> ]
<display item> ::=
                    DISPLAY MODULE [ <module name>]
                |   DISPLAY PROCEDURE [ <procedure name>]
                |   DISPLAY TABLE [ <table name>]
                |   DISPLAY VIEW [ <table name> ]
                |   DISPLAY FUNCTION [ <adt name> ]
```

```
            |   DISPLAY AGGREGATE [ <adt name> ]
            |   DISPLAY PRIVILEGE [ <table name> ] [ FOR <grantee> ]
            |   DISPLAY PRIVILEGE DB [ <table name> ] [ FOR <grantee> ]
            |   DISPLAY TRANSACTION [ <transaction id> ]
            |   DISPLAY VERSION
            |   DISPLAY DB
<transaction id> ::=
                Implementor-defined transaction indicator
```

If the name of the entity is not specified, all the entities of the type are displayed. The ALL clause causes a more verbose output to be generated. For example, all references in the composite DAG, class definition hierarchy will be displayed for tables and all style references for style.

If a <destination clause> (defined in Section 9.2) is not specified, the output will be printed on the standard output device.

Indices are displayed in the <display table statement>.

8.7 Rename Statement

The <rename statement> will rename a data dictionary element.

```
<rename statement> ::=
                RENAME <table name> TO <table name>
            |   RENAME <table name> . <column name> TO <column name>
            |   RENAME FUNCTION <function name> TO <function name>
            |   RENAME AGGREGATE <function name> TO <function name>
```

8.8 Copy Statement

The <copy statement> will copy a data dictionary element.

```
<copy statement> ::=
                COPY <table name> TO <table name>
            |   COPY FUNCTION <function name> TO <function name>
            |   COPY AGGREGATE <function name> TO <function name>
```

8.9 Comment Statement

The <comment statement> provides a mechanism to document a table or column.

```
<comment statement> ::=
                COMMENT ON <table name>
            |   COMMENT ON <table name> <character string literal>
            |   COMMENT ON <table name> . <column name>
            |   COMMENT ON <table name> . <column name>
                                        <character string literal>
```

8.10 Like Predicate

The <like predicate> has been extended to perform pattern matches based on phonetics.

```
<like predicate> ::=
                    |   <column specification> [ NOT ] SOUNDEX <pattern>
```

9 Input, Output, Graphical Display

The current SQL standard does not specify facilities to control input units or output destinations and formats. With geographic information it is very important to handle the graphical display of the data. Since the definition of how the information looks is separate from the geometric and topologic data, a new ADT, STYLE, is defined to manage the specification. There have been proposals in the past [10] that have proposed a separate language for graphic display. The specification of the graphics display has been included in this definition to provide one consistent interface to managing GIS information.

9.1 Input Control

This GIS/SQL accommodates units for spatial information. There are numerous unit systems that are handled in a global sense and a local sense. For example, the units of the coordinate system may be specified in meters but for a specific SQL query the results are presented in miles. This is implemented by a postfix unary operator such as 'km' that could convert from kilometers to meters.

9.2 Output Control

The <destination clause> controls the destination of the results from a select or data dictionary display command. By default, if the <destination clause> is not specified, the results are formatted and printed on the terminal.

```
<destination clause> ::=
                        INSERT [INTO] <table reference>
                    |   INTO <destination file>
                    |   DUMP [[INTO] <destination file> ]
                    |   PIPE [INTO] <destination command>
                    |   DUMP PIPE [ INTO ] <destination command>
<destination file> ::=
                        <value expression>
<destination command> ::=
                        <value expression>
```

The INSERT keyword directs the results into a table. The table specified in <table reference> will be created if it does not exist. If the table exists, the results of the query must match the <table reference> otherwise an error will result.

The INTO keyword directs the results into a specified file. The <value expression> of <destination file> and <destination command> are implementor-defined. The DUMP keyword specifies that a dump format representation is output. The dump format should be standard across all systems and include information such as units. This will provide a simple mechanism to move data from one system to another. If the <destination file> is not specified, standard output is used. The PIPE keyword specifies the results to be written to the standard input of the command specified. This command may be any implementor-defined OS command.

9.3 Select Many Statement

The <select statement: many> statement has been added to select many rows from one or more tables into an optional destination.

```
<select statement: many> ::=
                    <query specification> [ <order by clause> ] [ <destination clause> ]
```

9.4 Default Clause

The <default clause> specifies the default graphical display style for a <column definition>.

```
<default clause> ::=
                |   DEFAULT <style constructor>
```

9.5 Style Constructor

The <style constructor> specifies a display style for geometry or attribute data.

```
<style constructor> ::=
                    <colour constructor>
                |   <pattern constructor>
                |   <unit format constructor>
                |   <fill constructor>
                |   <stroke constructor>
                |   <symbol constructor>
                |   <annotation constructor>
                |   <grid constructor>
```

The style should match the type of information:

Attribute Type	Style Type
NODE GEOMETRY	symbol
LINEAR GEOMETRY	stroke
SURFACE GEOMETRY (unclosed)	stroke
SURFACE GEOMETRY (closed)	fill
SOLID GEOMETRY	fill
RASTER GEOMETRY	NULL
All Others	annotation

Default styles can be defined on a base class like *Simple* that will be used if there is no style defined for the data being display.

9.6 Colour Constructor

The <colour constructor> specifies a colour definition.

```
<colour constructor> ::=
                COLOUR ( <colour body> )
        |       FOREGROUND ( <colour body> )
        |       BACKGROUND ( <colour body> )
        |       <colour item>
<colour body> ::=
                <colour name>
        |       <colour item>
<colour item> ::=
                RGB ( <value expression> , <value expression> , <value expression> )
        |       CMY ( <value expression> , <value expression> , <value expression> )
        |       HLS ( <hls degree> , <value expression> , <value expression> )
<hls degree> ::=
                <value expression>
<colour name> ::=
                <style name>
        |       <value expression>
```

The <style name> is used to reference an existing colour. No other items can be specified if this directive is used. <hls degree> is a number in degrees between 0 and 360. The <value expression> in RGB, CMY or HLS directives are real numbers between 0 and 1.

9.7 Pattern Constructor

The <pattern constructor> specifies a fill pattern for graphic fills or cursors.

```
<pattern constructor> ::=
                PATTERN ( <pattern item> [{, <pattern item> }...] )
<pattern item> ::=
                <style name>
        |       <colour constructor>
        |       <pattern colour>
```

| PATTERN_SIZE (<value expression> [, <value expression>])
| PATTERN_ORIGIN (<value expression> [, <value expression>])
| PATTERN_BITMAP (<value expression list>)
| PATTERN_EXTENT <coordinate list>
| DRAW_TEXT (<value expression>, <coordinate> , <annotation constructor>)
| DRAW_GRAPHIC (<geometry constructor> , <style constructor>)
| <default style>

<pattern colour> ::=
PATTERN_COLOUR (<value expression> , <colour constructor>)

<default style> ::=
DEFAULT_STYLE (<style name>)

The <style name> is used to reference an existing pattern. No other items can be specified if this directive is used. The <value expression> in <pattern colour> must specify a string of length 1. The <pattern colour> defines one colour for the pattern. The PATTERN_SIZE directive defines the size of the pattern. The PATTERN_ORIGIN directive defines the hot spot for the pattern if used as a cursor.

Each <value expression> in the PATTERN_BITMAP directive must evaluate to a string with the same length. If the PATTERN_SIZE is specified then the number of characters must equal the first <value expression> (x) and the number of <value expression>'s in the PATTERN_BITMAP directive must equal the second <value expression> (y).

The PATTERN_EXTENT directive has two coordinates in the <coordinate list>. This directive defines the coordinate space for the pattern's drawing area used by the DRAW_TEXT and DRAW_GRAPHIC directives. This space will be mapped to the size of the pattern. The DRAW_TEXT directive specifies a string to draw in the pattern. The DRAW_GRAPHIC directive specifies a geometry to draw in the pattern. The <style constructor> must be consistent with the <geometry constructor>.

In the PATTERN_BITMAP directive, a blank character in the string specifies the background colour and a "1" flags the foreground colour. The default action can be changed by a <pattern colour> specification.

9.8 Unit Format Constructor

The <unit format constructor> defines an output format to display.

<unit format constructor> ::=
UNIT_FORMAT (<format item> [{, <format item> }...])

<format item> ::=
<style name>
UNIT_OUTPUT (<value expression>)
UNIT_EXPRESSION (<value expression>)

The <style name> is used to reference an existing unit format. The <value expression> in the UNIT_EXPRESSION defines the expressions to calculate the components of the unit. The <value expression> in the UNIT_FORMAT defines the output format of the unit.

9.9 Fill Constructor

The <fill constructor> specifies a graphic fill style.

```
<fill constructor> ::=
                        FILL ( <fill item> [{, <fill item> }...] )
<fill item> ::=
                        <style name>
                  |     <colour constructor>
                  |     SCALE ( <value expression> [ , <value expression> ] )
                  |     BLANK
                  |     SOLID
                  |     TILE ( <pattern constructor> )
                  |     STIPPLE ( <pattern constructor> )
                  |     OPAQUE ( <pattern constructor> )
                  |     HATCH ( <stroke constructor> )
                  |     BORDER ( <stroke constructor> )
                  |     <fill hatch>
                  |     <default style>
<fill hatch> ::=
                        HATCH_NORMAL [ ( <value expression> ) ]
                  |     HATCH_SHORTEST [ ( <value expression> ) ]
                  |     HATCH_DIRECTION [ ( <value expression> ) ]
```

The <style name> is used to reference an existing fill. The BLANK directive specifies a hollow fill. The SOLID directive specifies a solid fill using the foreground colour. The TILE, STIPPLE and OPAQUE directives define the ways to render the pattern in a fill operation. The HATCH directive specifies a line in a cross hatch fill. The angle and offset are defined in the referenced <stroke constructor>. BLANK, SOLID, TILE, STIPPLE and OPAQUE are mutually exclusive.

<fill hatch> defines the fill angle with regard to the specific fill operation:

NORMAL	reference axis is not rotated. This is the default.
SHORTEST	reference axis is rotated to the angle of the shortest line in the polygon border.
DIRECTION	reference axis is rotated to the angle of the line closest to the X axis.

The <value expression> of the <fill hatch> defines an additional angle of rotation. The BORDER directive specifies a border line for the fill.

9.10 Stroke Constructor

The <stroke constructor> specifies a graphic stroke style.

```
<stroke constructor> ::=
                        STROKE ( <stroke item>
                            [{, <stroke item> }...] )
<stroke item> ::=
                        <style name>
                    |   <colour constructor>
                    |   BLANK
                    |   SOLID
                    |   OFFSET ( <value expression> )
                    |   ANGLE ( <value expression> )
                    |   SCALE ( <value expression> [ , <value expression> ] )
                    |   DASHED ( <stroke dashed type> , <stroke dashed start> ,
                            <value expression>      [{, <value expression> }...] )
                    |   PARALLEL ( <stroke constructor> )
                    |   RELATIVE ( <symbol constructor> , <stroke relative type> )
                    |   REPEAT ( <symbol constructor> )
                    |   INTERMEDIATE ( <symbol constructor> )
                    |   WIDTH ( <value expression> )
                    |   CAP ( <stroke cap type> )
                    |   JOIN ( <stroke join type> )
                    |   <default style>
<stroke dashed type> ::=
                        <value expression>
<stroke dashed start> ::=
                        <value expression>
<stroke cap type> ::=
                        <value expression>
<stroke join type> ::=
                        <value expression>
<stroke relative type> ::=
                        <value expression>
```

The <style name> is used to reference an existing stroke. The BLANK directive specifies that the base line is not rendered. The SOLID directive specifies a solid line using the foreground colour. The DASHED directive is used to define a dashed line. The <stroke dashed type> defines the kind of dashed line. The <stroke dashed start> defines if the dash starts on an up or down stroke. The <value expression>'s define the alternating lengths of the up and down strokes of the dashed line. BLANK, SOLID and DASHED are mutually exclusive.

The <value expression> of <stroke dashed type> is a string with the following values: CONTINUOUS, ABSOLUTE, DOUBLE. CONTINUOUS uses the length defined in physical distances. ABSOLUTE and DOUBLE use the lengths in screen pixels. All three types draw the down stroke in the foreground colour. In CONTINUOUS and ABSOLUTE the up stroke is blank. In DOUBLE, the up stroke is drawn in the background colour. The <value expression> of <stroke dashed start> is a string with the following values: UP and DOWN.

The PARALLEL directive specifies a parallel line. The <offset> specified in the referenced <stroke constructor> is the offset of the parallel line from the base line. The RELATIVE directive specifies a relative symbol. The <value expression> of <stroke relative type> is a string with the following values: LEFT, MIDDLE, RIGHT. LEFT draws the symbol on the right end point of the line. MIDDLE draws the symbol at the midpoint between the left and right ends of the line. RIGHT draws the symbol on the left end point of the line. The REPEAT directive specifies a repeat symbol. The <offset> specified in the referenced <node constructor> is the distance between repeating symbols. The INTERMEDIATE directive specifies a symbol to draw at the halfway point on the line.

The WIDTH directive specifies the line width. The width is specified in pixels. The CAP directive specifies the line cap style: BUTT, NOTLAST, PROJECTING or ROUND. The JOIN directive specifies the line join style. The JOIN directive only applies to lines that are wider than one pixel and has the following values: BEVEL, MITER, ROUND.

9.11 Symbol Constructor

The <symbol constructor> specifies a graphic symbol style.

```
<symbol constructor> ::=
                        SYMBOL ( <symbol item> [{, <symbol item> }...] )
<symbol item> ::=
                        <style name>
                    |   OFFSET ( <value expression> )
                    |   ANGLE ( <value expression> )
                    |   SCALE ( <value expression>  [ , <value expression> ] )
                    |   ORIGIN <coordinate>
                    |   DIMENSION <coordinate list>
                    |   MASK ( <linear constructor> )
                    |   DRAW_TEXT ( <value expression>, <coordinate> , <annotation
constructor> )
                    |   DRAW_GRAPHIC ( <geometry constructor> , <style constructor> )
                    |   <default style>
```

The <style name> is used to reference an existing symbol. The ORIGIN directive defines the point where the symbol will be rendered. The DIMENSION directive is the dimension of the symbol. The MASK directive defines the masking footprint.

The DRAW_TEXT directive specifies a string to draw in the symbol. The DRAW_GRAPHIC directive specifies a geometry to draw in the symbol. The <style constructor> must be consistent with the <geometry constructor>.

9.12 Annotation Constructor

The <annotation constructor> specifies a graphic annotation style.

```
<annotation constructor> ::=
                        ANNOTATION ( <annotation item> [{, <annotation item> }...] )
```

```
<annotation item> ::=
                    <style name>
                |   <colour constructor>
                |   FONT ( <value expression> )
                |   <unit format constructor>
                |   FONT_TYPE ( <value expression> )
                |   PLACEMENT ( <value expression> [ , <value expression> ] )
                |   PATH ( <value expression> )
                |   SIZE ( <value expression> [ , <value expression> ] )
                |   SPACING ( <value expression> [ , <value expression> ] )
                |   ANGLE ( <value expression> )
                |   POSITION <coordinate>
                |   COURSE <coordinate list>
                |   <default style>
```

The <style name> is used to reference an existing annotation. The FONT directive references a stroke font if the precision is STROKE or a bit map font if the precision is BITMAP. The <unit format constructor> defines the unit conversion for the attribute display.

The FONT_TYPE directive defines the type of font to use. The <value expression> must result in a string with the following values: STROKE and BITMAP.

The PLACEMENT directive defines where the text is placed relative to the origin. The <value expression>'s must result in a string with the following values: LEFT, CENTER, RIGHT, BOTTOM, MIDDLE and TOP and RIGHT. TOP RIGHT is the default.

The PATH directive defines the direction to print the text. The <value expression> must result in a string with the following values: LEFT, RIGHT (default), UP, DOWN.

The SIZE directive defines the x and y size of the text for stroke precision. If the second <value expression> is not specified, y size will be the same as the x size. The SPACING directive defines the spacing between letters and between lines for stroke precision. If the second <value expression> is not specified, the spacing between lines will be the same as the horizontal spacing. The ANGLE directive defines the angle to draw the text.

The COURSE directive specifies a coordinate list to draw the text along.

9.13 Grid Constructor

The <grid constructor> specifies a graphic grid style.

```
<grid constructor> ::=
                    GRID ( <grid item> [{, <grid item> }...] )
<grid item> ::=
                    <style name>
                |   SCALE ( <value expression> [ , <value expression> ] )
                |   SIZE ( <value expression> [ , <value expression> ] )
                |   GRID_MARGIN <grid margin body>
```

```
                        | GRID_TICK <grid tick body>
                        | GRID_MARK <grid mark body>
                        | GRID_ANNOTATION <grid annotation body>
                        | <default style>
<grid margin body> ::=
                          ( <grid margin item> [{, <grid margin item> }...] )
<grid margin item> ::=
                          <stroke constructor>
                        | GRID_LEFT ( <value expression> )
                        | GRID_RIGHT ( <value expression> )
                        | GRID_TOP ( <value expression> )
                        | GRID_BOTTOM ( <value expression> )
                        | GRID_THICKNESS ( <value expression> )
<grid tick body> ::=
                          ( <grid tick item> [{, <grid tick item> }...] )
<grid tick item> ::=
                          <stroke constructor>
                        | GRID_SIZE ( <value expression> )
                        | GRID_OFFSET ( <value expression> )
                        | GRID_LOCATION <grid mask>
<grid mark body> ::=
                          ( <grid mark item> [{, <grid mark item> }...] )
<grid mark item> ::=
                          <symbol constructor>
                        | GRID_SCALE ( <value expression> [ , <value expression> ] )
                        | GRID_ANGLE ( <value expression> )
                        | GRID_SPACING ( <value expression> [ , <value expression> ] )
<grid annotation body> ::=
                          ( <grid annotation item> [{, <grid annotation item> }...] )
<grid annotation item> ::=
                          <annotation constructor>
                        | GRID_AXIS_FORMAT ( <value expression> [ , <value expression> ] )
                        | GRID_OFFSET ( <value expression> )
                        | GRID_LOCATION <grid mask>
<grid mask> ::=
                          ( <grid mask bit> [{, <grid mask bit> }...] )
<grid mask bit> ::=
                          <value expression>
```

The <style name> is used to reference an existing grid. The GRID_AXIS_FORMAT directive defines the format statement for the labels X and Y axis of the grid. The SIZE directive defines the sheet height and width. The GRID_MARGIN directive defines the appearance of the border. The GRID_TICK directive defines the appearance of the tick marks. The GRID_MARK directive defines the appearance of the grid marks. The GRID_ANNOTATION directive defines the appearance of the margin text.

10 Concluding Remarks

To summarize, these are the following GIS extensions to SQL:

- Object oriented extensions include data modeling facilities and ADT support.

- Spatial Extensions to enable the user to store (with a GEOMETRY ADT), query and update the spatial data.
- Extended Transaction model that includes a Long (possibly remote) transaction.
- Data Dictionary Extensions to maintain the information required for projects, long transactions, fragmentation information, thematic display, transformations and accuracy.
- Input/Output Facilities including destination clause and STYLE ADT.
- Language extensions to add flexibility to the current SQL standard.

GIS/SQL will benefit the users by providing a common language to manage the GIS information and control data transfers through SQL gateways. External systems that do not have support for GIS information can still take advantage of information by defining a view on a spatial constraint. The object oriented modeling facilities provide a way to present the user with a view of their data and relationships that is familiar, Using the long transaction model, the problem of large data loading or update that is typically performed off site is handled in a simple manner. The temporal extensions enable the user to deal with information from the past, present and a possible future in the same environment. The spatial extensions enable the user to perform very sophisticated spatial analysis in an ad hoc fashion. This is done by building on the same language foundation contained in most relational database systems today.

11 Glossary

Term	Definition
ADT	Abstract Data Type: A data type that is defined from the set of system defined data types. Operations on ADTs are performed with methods and operators.
Angle	An angle is measured in degrees counter-clockwise where 0 is East.
ATB	Analytical Tool Box: SYSTEM 9 Analysis Package.
Attribute	A descriptor for a given characteristic that derives its value from a domain. This is represented as a column in a table.
Azimuth	An azimuth is measured in degrees clockwise where 0 is North.
Centroid	The center of gravity for a entity projected so that it is usually contained within the geometry.
Class Definition	A classification of a real world phenomenon. This is represented as a table.
Composite Feature	An entity that is formed from two or more simpler entities.
DAG	Directed Acyclic Graph (or Acyclic Directed Graph).
Entity	The abstraction of a real world phenomenon defined in terms of its geometry, topology or attribute data.
Feature	The abstraction of a real world phenomenon defined in terms of its geometry and topology data.
GIS	Geographic Information System
Geometry	The coordinate representation for an entity.
LHS	Left Hand Side.
Masking	An operation in plotting to determine how the information is rendered on the output device.
Persistent Object	Objects that are stored on a persistent storage media such as a disk.
RHS	Right Hand Side.
Simple Feature	An entity that models a single real world phenomenon.
Transaction	A program module that accesses a database and has the following properties [18]: *Atomicity* Either all the transaction's actions are completed, or none of them are. *Consistency* A transaction changes from one consistent database state into another. *Isolation* The results of a transaction cannot be seen by other concurrent transactions until it commits. *Durability* Once a transaction commits, its results on the database are permanent.
Topology	The properties of a geometrical figure that are unaffected when it is subjected to any continuous transformation or deformation.

References

[1] ANSI X3.135-1989, *Database Language SQL with Integrity Enhancement*, FIPS PUB 127-1.

[2] ANSI X3.135-199X, *Database Language SQL*, Draft Proposed American National Standard.

[3] ANSI X3.217-199X, *Remote Database Access*, Draft Proposed American National Standard.

[4] Ashworth, M., *Performing Speculative Analysis in a GIS Environment*, GIS/LIS'90 Proceedings, American Society for Photogrammetry and Remote Sensing, p. 598-597, November 1990.

[5] Ashworth, M., *GIS/SQL Language Definition for the Transaction-Based Multi-User SYSTEM 9 Database System*, Available on request, July 1992.

[6] *SYSTEM 9 ATB Specialist's Manual*, SYSTEM 9 Technical Documentation, June 1990.

[7] Beckmann, N., H. Kriegel, R. Schneider, B. Seeger, *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*, Proc ACM SIGMOD Int. Conf. on Management of Data, p. 322-331, 1990.

[8] Claire, R.W., S.C. Guptill, *Spatial Operators for Selected Data Structures*, Auto-Carto V/Commission IV, International Society for Photogrammetry and Remote Sensing Symposium, p. 189-200, August 1982.

[9] Date, C.J., A Guide to the SQL Standard, Addison-Wesley Publishing Company Inc., 1987.

[10] Egenhofer, M., *Extended SQL for Graphical Display*, Cartography and Geographic Information Systems, Vol. 18, No. 4, p.230-245, 1991.

[11] Egenhofer M., J. Sharma, *Topological Consistency*, Proc. 5th Int. Symp. on Spatial Data Handling, V1 p. 305-343, August 1992.

[12] Foley, J.D., A. van Dam, S. K. Feiner, J. F. Hughes, Computer Graphics: Principle and Practice 2nd Ed., Addison-Wesley Publishing Company, 1990.

[13] Guo, A., V. B. Robinson, *RT-Tree: Spatiotemporal Indexing for Dynamic Land Databases*, Proc of the Canadian Conference on GIS, p.143-155, March 1992

[14] Guttman, A., *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proc ACM SIGMOD Int. Conf. on Management of Data, p. 47-57, 1984.

[15] Mackay, D.S., V. B. Robinson, *Towards a Heterogeneous Information Systems Approach to Geographic Data Interchange*, **Institute for Land Information Management Discussion Paper 92/1**, University of Toronto, 37p. 1992.

[16] Moon, G., M. Ashworth, *Capabilities Needed in Spatial Decision Support Systems*, **GIS/LIS'90 Proceedings**, American Society for Photogrammetry and Remote Sensing, p.594-600, November 1992.

[17] **NOAA/USGS General Map Projection Package**, Dr. A. A. Elassal, GCTP/II, Version 1.0.2, September 1986.

[18] Ozsu, M.T., P. Valduriez, **Principles of Distributed Database Systems**, Prentice Hall Inc., 1991

[19] Pigot, S., *A Topological Model for a 3D Spatial Information System*, **Proc. 5th Int. Symp. on Spatial Data Handling**, V1 p. 344-360, August 1992.

[20] Tuori, M., G. Moon, *A Topographic Map Conceptual Model*, **Proc. of the Int. Symp. on Spatial Data Handling**, Zurich, 1984

[21] *SYSTEM 9 Query Language*, **SYSTEM 9 Technical Documentation**, June 1990.

[22] *Spatial Data Transfer Standard*, **FIPS-173**. 1992

[23] Stonebraker, M., et al., *On Rules, Procedures, Caching and Views in Database Systems*, **Proc. 1990 ACM-SIGMOD Conference on Management of Data**, Atlantic City, N.J., June 1990

[24] Tomlin, C.D., **Geographic Information System: Cartographic Applications**, Prentice-Hall, 1990.

Mark Ashworth is a Principal Software Engineer in the GIS Business Unit of Computervision. He is responsible for the development of the object oriented, spatial database management system at the core of SYSTEM 9. Mark also lead the team that designed and implemented the Analytical Tool Box, an innovative GIS software package which provides extensible tools to retrieve, display, report and analyze data from the SYSTEM 9 database. Mark represents Computervision on GIS related database standards in industry and, for the last two years, he has been working on GIS extensions to SQL. In 1980, Mark Ashworth is received an Honors B.Sc. in Computer and Information Science from University of Guelph of Ontario, Canada. He has been with the SYSTEM 9 project for the past 7 years.

Mr. Ashworth can be reached by telephone at (416) 508-8020 x884, by facsimile at (416) 508-8077, or by electronic mail at Internet address mark@cvgis.prime.com .

A. Appendix - Examples

A.1 Create New Simple Classes

This example creates a class called parcel that is a subclass of Simple. The attribute s9_geometry has been modified from the Simple definition to specify surface topology. The attribute pid holds that parcel id.

```
CREATE TABLE parcel (
        ISA             Simple,
        s9_geometry     GEOMETRY (SURFACE),
        pid             CHARACTER(8) NOT NULL UNIQUE);
```

This example creates a class called road that is a subclass of Simple. The attribute s9_geometry has been modified from the Simple definition to specify linear topology. The attribute name holds the name of the road. The attribute road_length holds the calculated length of the road.

```
CREATE TABLE road(
        ISA         Simple,
        s9_geometry     GEOMETRY (LINEAR),
        name        CHAR(64),
        road_length     LENGTH(s9_geometry));
```

A.2 Create new Composite Class

This example creates a composite feature class called block. This composite includes two simple feature classes called parcel and road. The attribute plan holds the plan number for the block.

```
CREATE TABLE block (
        ISA             Composite,
        COMPOSED OF (parcel,road),
        plan        INTEGER);
```

A.3 Create an Entity

This example inserts an entity into the class parcel.

```
INSERT INTO parcel (pid , s9_geometry )
        VALUES ( "1079647A", SURFACE
            (   ( 501596.4 , 6276913.3 ),
                ( 501613.9 , 6276917.4 ),
                ( 501614.2 , 6276957.4 ),
                ( 501695.3 , 6276957.3 ),
                ( 501596.4 , 6276913.3 )));
```

A.4 Selecting Entities

This example selects the parcel ids of all the parcels that overlap any soil of type KNZ2.

```
SELECT pid FROM parcel WHERE s9_geometry OVERLAPS (
                    SELECT s9_geometry FROM soil
                        WHERE type = "KNZ2");
```

This example selects all the gravel pits within 100 kilometers:

```
SELECT FROM  gravel_pit WHERE s9_geometry  OVERLAPS (
                SELECT BUFFER(s9_geometry, 100km) FROM PARCEL
                    WHERE owner = "Jones");
```

This example selects the parcels that have an area greater than 100 acres and have some frontage on a lake in the district of Parry Sound.

```
SELECT FROM parcel WHERE AREA(s9_geometry) > 100ac AND
                    s9_geometry IS ADJACENT TO (
                        SELECT lake.s9_geometry FROM lake,district
                            WHERE lake.s9_geometry OVERLAPS
                                district.s9_geometry AND
                                district.name = "Parry Sound");
```

This example determines the type of soil on all parcels greater than or equal to 20 acres. The area of each soil type in the parcel is also calculated.

```
SELECT parcel.pid,soil.type,
                AREA(OVERLAP(parcel.s9_geometry,soil.s9_geometry))
                FROM parcel, soil
                    WHERE AREA(parcel.s9_geometry) >= 20ac;
```

A.5 Create a Project Schema Example

This is a detailed example of creating a project schema from scratch with GIS/SQL.

A.5.1 create the project.

```
CREATE PROJECT 'Northern_Ontario' (
                SURFACE ( ( 501596.4 , 6276913.3 ),
                        ( 501613.9 , 6276917.4 ),
                        ( 501614.2 , 6276957.4 ),
                        ( 501695.3 , 6276957.3 ),
                        ( 501596.4 , 6276913.3 ) ),
                UNIT METRIC,
                HOST 'ThunderBay',
                PORT 1040,
                LOCATE IN '/dbs',
```

```
NOT COMPRESSED);
```

A.5.2 create some styles.

```
CREATE STYLE 'solid_blue_fill'  FILL ( FOREGROUND ('blue'));
CREATE STYLE 'solid_blue_line'       STROKE ( FOREGROUND ('blue'), SOLID );

CREATE STYLE 'dam'      SYMBOL (
                        ORIGIN (.5,.5), DIMENSION ((0,0),(1,1)), MASK
                                                  (RECTANGLE ((0, 0),(1,1))),
                        DRAW_GRAPHIC (
                              LINE((0, .55), (1, .55), (1, .45),
                                    (0, .45), (0, .55)),
                              STROKE ( COLOR ("blue"), SOLID) ) );
CREATE STYLE 'black_line'     STROKE ( FOREGROUND ('black'), SOLID );
CREATE STYLE 'black_border_fill'     FILL ( BLANK, BORDER ( 'black_line' ) );
CREATE STYLE 'black_text'
                  ANNOTATION ( FOREGROUND ('black'), FONT ("simprom"),
                        FONT_TYPE ("STROKE"), SIZE (.3, .4), SPACING (.4), POSITION (0,
                        .5),
                        PLACEMENT ("TOP", "RIGHT"), PATH ("RIGHT"));
```

A.5.3 create some classes.

```
CREATE TABLE lake (
                  ISA                     Simple,
                  name                    CHARACTER (24) NOT NULL,
                  max_depth               DOUBLE PRECISION,
                  s9_geometry             GEOMETRY (surface)
                        DEFAULT FILL ('solid_blue_fill' ) );

CREATE TABLE river (
                  ISA               Simple,
                  name              CHARACTER (24) NOT NULL,
                  len               LENGTH(s9_geometry),
                  s9_geometry       GEOMETRY (LINEAR)
                        DEFAULT FILL 'solid_blue_line' ) );

CREATE TABLE dam (
                  ISA                     Simple,
                  name                    CHARACTER (12) NOT NULL,
                  owner                   STRING (12, 24, 64),
                  s9_geometry             GEOMETRY (node)
                        DEFAULT SYMBOL ('dam' ) );
```

```
CREATE TABLE parcel_boundary (
                    ISA                 Simple,
                    legal_length        DOUBLE PRECISION,
                    calc_length         LENGTH(s9_geometry),
                    legal_direction     DOUBLE PRECISION,
                    calc_direction      AZIMUTH(s9_geometry),
                    s9_geometry         GEOMETRY (linear)
                        DEFAULT STROKE ( "black_line" ));

CREATE TABLE soil (
                    ISA                 Simple,
                    type                SMALLINT
                        DEFAULT ANNOTATION ('black_text' )
                        NOT NULL
                        CHECK ( type > 0 and type < 4 ),
                    s9_geometry         GEOMETRY (surface));

CREATE TABLE water_way (
                    ISA                 Composite,
                    COMPOSED OF         (lake, river, dam));

CREATE TABLE parcel (
                    ISA                 Composite,
                    COMPOSED OF         parcel_boundary,
                    s9_geometry         GEOMETRY (surface)
                            DEFAULT FILL ( 'black_border_fill' ));
```

A.5.4 create some themes.

```
CREATE THEME water ( MAPSCALE ( 5000 ), lake, river, dam );
CREATE THEME soil_map ( MAPSCALE ( 10000 ), lake, river, dam, soil, parcel );
```

A.5.5 create thematic display.

```
CREATE THEMATIC DISPLAY 'sandy'(
                    FOR soil.s9_geometry, FILL ( COLOUR ('brown'), SOLID),
                    IN soil_map,          WHERE type = 1 );

CREATE THEMATIC DISPLAY 'rock'(
                    FOR soil.s9_geometry      FILL ( COLOUR ('violet'), SOLID),
                    IN soil_map,              WHERE type = 2 );

CREATE THEMATIC DISPLAY 'loam'(
                    FOR soil.s9_geometry      FILL ( COLOUR ('tan'), SOLID),
                    IN soil_map,              WHERE type = 3 );
```

Conceptual Folding and Unfolding of Spatial Data for Spatial Queries

Jan W. van Roessel
ESRI
380 New York Street
Redlands, CA 92373
USA

1 Introduction

This paper is an exploratory paper, written as a vehicle for developing ideas for a spatial query language. As such it is incomplete, may contain inconsistencies, inadequate references and incomplete ideas. We hope that it will get a few main points across, so that these can be rejected or be further explored.

Many proposals for SQL extensions are based on the object in a field concept, where the objects can be lines, points or polygons, or other things. Relational database managers increasingly support a binary large object or "blob" as a data type. ADT Ingress supports abstract data types such as line, point or polygon. The SQL 3 proposal has user defined abstract data types. Various operations on these objects are proposed, for example, those based on distance, direction and neighbourhood relationships. The general trend seems to be make SQL the "kitchen-sink: query language, supporting not only a relational but also an object oriented philosophy.

We would like to go back to the basics, and try to see if relational concepts can be applied to spatial data, and whether existing relational operations can be used when a piece of two dimensional space is conceptually considered a relation. If this can be done then the large body of relational knowledge can be used and existing SQL constructs can be applied in a new light.

2 Cover as Relation

Let us define a cover as a subset of two dimensional space. It can be said to consist of an infinite set of points, but with a finite extent. In practice, it is either represented by vector data or by raster data. The vector representation copes with the infinite number of points by describing the fence around the points, while raster data solves this problem by using a sampled representation. Whatever the practical representation methods may be, they should not restrict the types of questions that can be asked. Queries are made on a conceptual level and they can manipulate abstract concepts. We can therefore conceive of a cover as a relation with an infinite number of tuples, and express the queries in terms of the conceptual relations.

This may sound like a worthwhile goal, but unfortunately we are restricted in the practical representation of the data. It does not make sense to perceive of queries on a conceptual level

that cannot be carried at the practical level, because the conceptual information is physically absent.

The fences in vector data represent the interior as one blob, and any heterogeneous attributes that occur inside such a fence cannot be attached to individual points. Similarly, for raster data, heterogeneous attributes associated with a raster point cannot be attached to its surroundings.

The solution is a compromise, based on the concept of a practical spatial object, its "unfolding" to the conceptual level of an infinite relation, followed by a query, then followed by a "folding" into practical spatial objects.

3 The coverage meta schema

To define more precisely how a coverage can be viewed as a spatial relation we must define prototype schema for the coverage. A relational schema is defined as a set of attributes with respective domains. The set union of a number of relational schemas is also a relational schema. We define a spatial object meta-schema as a class of relational schemas where each schema of that class has a certain functionality with respect to a spatial object. The set union of a number of meta-schemas is also a meta-schema. The following meta schemas are proposed:

S is the meta schema for the spatial address of the point. Its attributes represent coordinates or spatial address components.

A is the meta schema for the homogeneous attributes of a spatial object.

B is the meta schema for the heterogeneous attributes of a spatial object.

K is the meta schema for an attribute called the object key. The value of this key is a unique representation for the object.

Let C = S[K][A][B], where C is the set union of the meta-schema S and any combination of K, A, or B, with the brackets indicating optional use. We would require the minimum meta schema of SKA for a spatial object cover.

We would then say that a schema C is an instance of the meta schema and a cover c(C) is an instance of a relation for this schema.

We may consider the relational primary key for various combinations of the component meta schemas. SKAB has primary key S, because each point is uniquely defined by its spatial address. KAB has primary key KB, because by definition A is functionally dependent on K. KA has primary key K.

Unique values for A represent disconnected spatial objects which may be called zones, themes or regions, and are equivalent to the "value" concept for a grid cell value. The schema for a raster representation might therefore well be SA.

Two coverages are said to be spatial join compatible when each coverage has the same subschema S. Attribute names for a subschema of type K should never be identical for coverages that are to be joined.

4 Spatial Objects

For our definition of a spatial object there are two important concepts. First, the object consists of points that have the same defining attributes, and hence the term homogeneous attributes. Let t^i and t^j represent two tuples corresponding to separate points. They have the same homogeneous attributes if the subtuples obtained by mapping ontothe sub-schemaA are identical, or $t^i_A = t^j_A$ (actually, it suffices to consider the attributes in the relational primary key of **A**, since the other homogeneous attributes are functionally dependent on this key).

Secondly, the points in the object should be contiguous or nearly contiguous. If we define a distance function $d(ti_S, t^j_S)$, and a neighbourhood ε then the tuples in c(C) can be grouped into objects based on the distance function as well as homogeneity. Each object is an equivalence class of an equivalence partition defined by the equivalence of the two arbitrary tuples, $t^i \varrho^+ t^j$, where ϱ is the transitive closure of the relation ϱ, which is:

$$\rho: \left(d(t_S^i, t_S^j) < \varepsilon\right) \wedge t_A^i = t_A^j$$

Each object will have an assigned unique object key t_K which embodies both attribute homogeneity and contiguity.

Note that the size of ε defines the granularity and fuzziness of the boundary between neighbouring objects.

5 Folding and Unfolding

The folding and unfolding concepts are borrowed from Lorentzos and Johnson (1987) who used the same idea to extend the relational model to cope with generic intervals, although they used a discrete interval model. Their operators produce practical results, while ours are conceptual.

Conceptual unfolding for a practical vector representation merely means to think of the coverage data as a relation. The vector boundary may be stored in one file, while the homogeneous attributes are stored in primary feature table which has the meta schema **KA**, with an object key and a set of homogeneous attributes. Heterogeneous attributes may be stored in a second table with defined join attributes, such as the spatial object key. To think of all these components as in a single relation is not too difficult. The concept is that of a "universal relation." Date (1990) makes the following remark about the universal relation: "...The second, and more pragmatically significant, manifestation of the universal relation concept is as a user interface ... Users should frame their database requests, not in terms of relations and joins among those relations, but rather in terms of attributes alone."

Conceptual folding, after the relations have been manipulated, consists of folding the points back into new objects. The process consists of four main steps: (1) specification of new defining (homogeneous) attributes, (2) determination of the object keys, (3) a change in the spatial relation to reflect the new homogeneous attributes, and (4) separation of spatial and attribute information.

The homogeneous attributes are defined as a part of the query. If there is no specification, the set union of the input schemas is used.

The new defining attributes determine the new object keys through the object relation specified earlier. Old object key attributes never can be join items, because they are required to have different names by definition. If they are present, for instance as the result of a natural join, they are eliminated in favour of a new object key attribute or they might be renamed and become type A or B attributes.

The change in the spatial relation that occurs because of a change in defining attributes results from the relationship between the object key and other attributes. For homogeneous attributes it is a single valued dependency, for heterogeneous attributes a multi-valued dependency.

We demonstrate the concept with a coverage relation consisting of two points defining two spatial objects that are merged into a single object by folding:

TABLE 1

S		K	A	
X	Y	K	T1	T2
1	1	1	a	b1
2	1	2	a	b2

In Table 1 the homogeneous attributes are T1 and T2. Suppose that the folding process redefines T1 as the sole homogeneous attribute, while T2 becomes heterogeneous. The result is that every single point then acquires the properties b1 and b2.

TABLE 2

S		K	A	B
X	Y	K	T1	T2
1	1	1	a	b1
1	1	1	a	b2
2	1	1	a	b1
2	1	1	a	b2

After separation of spatial and other attributes a feature attribute table for a vector object corresponding to Table 2 may be:

TABLE 3

K	A	B
K	T1	T2
1	a	b1
1	a	b2

Or one might separate these tables into a primary and secondary feature attribute table:

TABLE 4

K	A
K	Tl
1	a

TABLE 5

K	B
K	T2
1	b1
1	b2

where Table 3 is a primary feature attribute table with a one-to-one relationship between object and rows and Table 4 is a related table with a one to many relationship. Unfolding of the boundary representation and reattaching the feature attribute tables (Tables 3 and 4) is equivalent to performing joins on object key as the join item, producing Table 2.

6 Operations on spatial relations

In the conceptually unfolded state either relational algebra or relational calculus can be applied to express the desired query results. Corresponding SQL constructs can be used to formulate SQL queries. We will first consider the relational algebra as a method to convey our ideas, because it allows their expression in the simplest and most direct way. A number of examples are presented. This then followed by a corresponding form of spatial SQL, and a restatement of some of the examples.

6.1 Relational Algebra

The relational algebra operators are: selection, projection, product, union, intersection, difference, join, divide and rename. We will use the syntax provided by Date (1990) and specify the required extensions for the approach proposed here.

A brief summary of Date's relational algebra operator syntax is:

Operator	Syntax	Comments
Union	A UNION B	relations A and B must be schema compatible
Intersection	A INTERSECT B	relations A and B must be schema compatible
Product	A TIMES B	Cartesian product
Difference	A MINUS B	relations A and B must be schema compatible
Selection	A WHERE X theta Y	instead of X theta Y can use restriction condition
Projection	A[X,Y,....,Z]	X,Y,...Z must be in schema of A
Natural join	A JOIN B	if A and B have no attributes in common A TIMES B
Theta-join	(A TIMES B) where X theta Y	Join on other than equality
Rename	A RENAME X AS Y	
Divide	A DIVIDEBY B	A schema X1,X1,Xm,Y1,Y1,Ym; B schema Y1,Y2,Ym

Any expression is a valid relation, and parenthesized expressions can be used as input to an operator. A precise BNF is found in Date (1990:300).

6.2 Natural Joins and Natural Outerjoins

The natural join of two spatial relations is equivalent to a "spatial overlay." The natural join takes the intersection of both input attribute schemas to define the attributes for which tuple values must match to generate a join tuple. It takes the set union of the attribute schemas to determine the output schema.

Assuming for the moment that both input relations have the same spatial address schema (necessary if the dimensionality of the output should be the same as the input), then the natural join guarantees that this subschema is also part of the output schema, and that points in the output relation are common to both input relations, thus performing a spatial intersection.

The set intersection of the input schemas may include other attributes as well. They are evaluated along with the spatial address. If common attributes for the same spatial address do not agree in value, a hole will appear in the output relation for that point.

A natural outer join between two relations does not discard tuples that do not match on the set intersection of the attribute schemas, but instead outputs the tuple and supplies null values for the attributes in the output schema that are not a part of the input relation schema. A left natural outer join does this only for the first relation, and a right natural outer join does it for the second relation. The spatial results of the various joins are shown schematically in Figure 1.

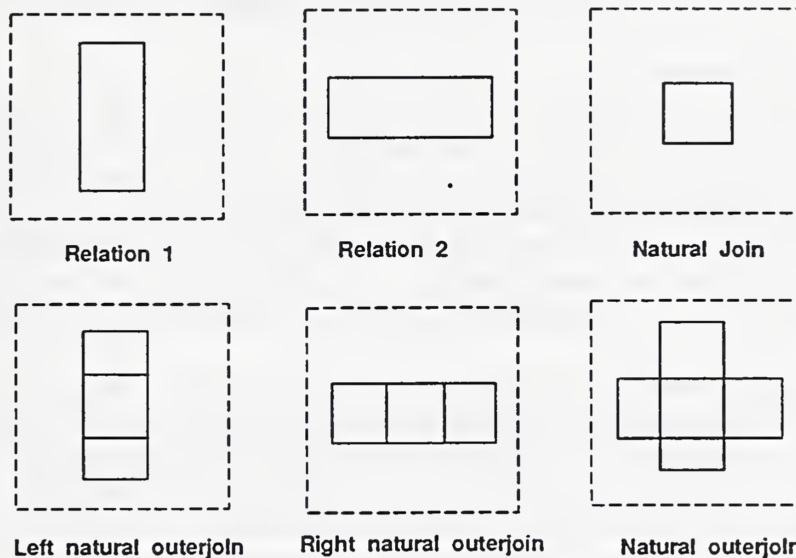


Figure 1. Spatial effects of natural join and natural outerjoin.

We will use the following notation for the various outer join operators:

Operator	Syntax
Natural Outerjoin	**A JOIN+ B**
Left natural outerjoin	**A JOIN+L B**
Right natural outerjoin	**A JOIN+R B**

It is interesting that the different types of join have a direct correspondence to the overlay functions in Arc/Info (ESRI, 1991):

Join	Overlay function
Join	Intersect
Natural outer join	Union
Left natural outer join	A Identity B
Right natural outer join	B Identity A

So far we have discussed operations on two layers. For more than two layers we must perform multiple joins, for instance (A JOIN B) JOIN C. As Date (1990) points out the JOIN is both associative and commutative so that we may simplify to: A JOIN B JOIN C. The same holds for the outer natural join, but not for either the left or right outer natural join.

6.3 Relational Set Operators

The join operators as described above can be used for a number of spatial operations, but they do not form a complete set. For instance, they cannot be used to achieve the data management functions in Arc/Info (ESRI, 1991) such as "erase" and "clip." The reason is that the joins themselves are composite operations that operate both on spatial and other attributes simultaneously. Separating the spatial operations, and then combining the result with non-spatial attributes leads to a basic system that can be used to accomplish any spatial operation.

Separating the spatial address from a spatial relation requires a project, for instance A[LOC] and B[LOC] for coverages A and B. Note that the projection removes duplicate spatial addresses, so that each point is represented by a single row. The projected A and B coverages are then schema compatible by definition, so that the UNION, INTERSECTION, and DIFFERENCE operators can be applied to produce spatial results. The spatial results must then be joined back to the original relation to attach the attributes.

The result of the natural join in the previous sections is then obtained with the following statement:

```
(A[LOC] INTERSECT B[LOC]) JOIN A JOIN B
```

where the term in parentheses represents the spatial operation, and the JOIN A JOIN B term represents the reattachment of non-spatial attributes.

The natural outer join is then:

```
(A[LOC] UNION B[LOC]) JOIN+L A JOIN+L B
```

The left outer joins are required to pick up null attributes in the A and B coverages where the spatial result has spatial addresses that do not occur in the A or B coverages.

The left and right natural outer joins are expressed as:

```
((A[LOC] INTERSECT B[LOC]) UNION A[LOC]) JOIN A JOIN+L B
((A[LOC] INTERSECT B[LOC]) UNION B[LOC]) JOIN+L A JOIN B
```

An "erase" of a part of cover A that corresponds to the area occupied by cover B (where A and B need not be schema compatible) may now be expressed as

```
(A[LOC] DIFFERENCE B[LOC]) JOIN A
```

And a clip of cover A using the area of cover B is

```
(A[LOC] INTERSECT B[LOC]) JOIN A.
```

Thus operations can be expressed in a "standard" manner in which spatial address operations are first done, and are followed by attribute joins. The coverages involved need not be schema compatible.

Finally, we note that the joins in the attribute joins could all be replaced with left natural outer joins. This would help standardize the attribute join operation.

7 The FOLD Operator

The conceptual folding and unfolding approach does not require any special algebra operator for unfolding the cover. However, folding requires the specification of the defining attributes. These attributes (and any other attributes that are functionally dependent) become the new homogeneous attributes.

Following the Date (1990) type syntax, the fold operator will be as follows:

```
R FOLD A1 A2..Am
```

where R is the conceptual spatial relation to be folded and A1, A2, ..Am are the defining attributes. This operator is very similar to the traditional GIS "merge and dissolve," and equivalent to the Arc/Info (ESRI, 1991) dissolve operation on a single or redefined (compound) item. In the algebra it is not necessary to use the FOLD expression if the sub-schema for homogeneous attributes is the set union of the homogeneous attribute sub-schemas of the input layers.

7.1 The Distance Join

To cope with GIS functions such as the buffer-zone operator, we need one more extension to the traditional relational algebra.

It is a special form of the theta join, which in Date's syntax is written as:

```
(A TIMES B ) WHERE X theta Y
```

where X and Y are attributes (or constants) and theta may represent <, >, <=, etc.

Assuming that A and B are spatial relations where each tuple has a spatial address, we can define a distance function DIST on the spatial addresses of two arbitrary tuples from A and B. A distance join is then defined as:

```
(A TIMES B) WHERE DIST theta Y.
```

To see how this construct generates a buffer zone, consider that for each tuple in A all tuples from B within the distance are joined, generating a circle. The next point also generates a circle, slightly offset from the first, all duplicate tuples are removed because the output is a set, making a union of the two circles. Then the third circle is joined with the union of the first two, and so on.

8 Examples

We will now present some examples for traditional GIS operations as expressed in the relational algebra. For these examples we will use the following data dictionary for the feature attribute table. The unfolded relation will have an object key and a spatial address as attributes as well (KEY and LOC).

8.1. Select and Intersect. A map needs to be developed showing sites meeting the criteria of *land use is brushland* and *soiltypes are suitable for development*. The following statements express this goal:

```
(SOILS WHERE SUITABILITY = 3)
JOIN
(LANDUSE WHERE LUTYPE = 300)
```

An alternate form is

```
(SOILS JOIN LANDUSE) WHERE SUITABILITY = 3
AND LUTYPE = 300
```

8.2. Select and Union. A variation on the theme in the first example is to select those sites where either one of the two criteria holds. For this we use the natural outer join:

```
(SOILS WHERE SUITABILITY = 3)
JOIN+
(LANDUSE WHERE LUTYPE = 300)
```

As in the first example, an alternate form is

```
(SOILS JOIN+ LANDUSE) WHERE SUITABILITY = 3 OR LUTYPE = 300
```

8.3. Selection with multiple layers. In the above example the input consists of two layers. For more than two layers we must perform multiple joins, for instance (A JOIN B) JOIN C. As explained, this my be simplified to A JOIN B JOIN C. The same holds for the

outer natural join, but not for either the left or right outer natural join. To locate the semi-improved roads that pass through unsuitable soils with a landuse of water:

```
ROADS JOIN+ LANDUSE JOIN+ SOILS
WHERE ROADCLASS = 2  AND  (LUTYPE = 500 OR SUITABILITY = 0)
```

The join condition is the algebraic form of the *GIS sandwich*. One proposal for a spatial query language is to let this condition exists as a default for all registered layers of interest.

8.4. Dissolve. A cover may also be created in which the spatial objects for the selected area show only the difference in land use type (thus "dissolving" on the other attributes). This is accomplished with the FOLD operator:

```
FOLD(
        (SOILS WHERE SUITABILITY = 3)
        JOIN
        (LANDUSE WHERE LUTYPE = 300)
) LUTYPE
```

In the resultant schema for the output cover, LUTYPE is the homogeneous attribute, while SOILTYPE and COST/HA now are heterogeneous attributes. Thus a vector output map would show only the maximal size polygons with a single LUTYPE.

8.5. Bufferzone. Suppose that the site selection requires that the selected sites are within 300 meters of existing sewer lines. This is accomplished with the distance join:

```
(       (SOILS WHERE SUITABILITY = 3)
        JOIN
        (LANDUSE WHERE LUTYPE = 300)
) TIMES SEWER WHERE DIST < 300
```

At the same time, the selected sites must be more than 20 meters removed from any existing streams.

```
(       (       (SOILS WHERE SUITABILITY = 3)
                JOIN
                (LANDUSE WHERE LUTYPE = 300)
        ) TIMES SEWER WHERE DIST < 300
) TIMES STREAMS WHERE DIST > 20
```

8.6. Erase. An erase can be defined as an operation where one cover is "erased" by the overlapping part of a second cover. If the two covers have compatible schemas, one can simply use the difference operator, but this is usually not the case. The solution is to make compatible schemas by projecting only the spatial addresses, to take the difference, and then to perform a join of the resulting relation with the first input relation to recover the original input schema. In the following example a part of the soil's coverage is erased where there is overlap with landuse:

```
(SOILS [LOC] MINUS LANDUSE[LOC])
JOIN
SOILS
```

DATA DICTIONARY

Coverage	Feature Type	Attributes	Value	Description
SOILS	poly	SOILTYPE	string	soiltype
		SUITABILITY	0	unsuitable
			1	poor suitability
			2	moderate suitability
			3	good suitability
LANDUSE	poly	LUTYPE	100	urban
			200	agriculture
			300	brushland
			400	forest
			500	water
			600	wetlands
			700	barren
		UNITCOST	integer	cost per area unit
STREAMS	line	STREAMCLASS	1	major stream
			2	minor stream
SEWERS	line	DIAMETER	real number	pipe diameter
		SYMBOL	1	60 cm pipe
			77	45 cm pipe
ROADS	line	ROADCLASS	1	improved
			2	semi-improved

8.7. Update. An update is an operation where one part of a cover is replaced with another part, that is "pasted" onto the old part. We mentioned in the discussion of the natural join for spatial relations, that points will not appear in the output where locations agree, but similar attributes do not. Holes are left instead. We can make use of this fact to do an update. The new cover will have attributes that are the same or different from the old. For those that are different, a hole is created. Then we can use the union operator between the joined cover with the hole and the new cover to fill the hole. Any overlap will be eliminated due to the fact that relations are sets without duplicate tuples. Assuming for the example that we update SOILS with SOILSNEW, the expression becomes:

```
(SOILS JOIN SOILSNEW)
UNION SOILSNEW
```

8.8. Identity. In Arc/Info (ESRI, 1991) there is an operation called "identity" where one map is intersected with another map, but the outline of the first map determines the outline of the output map. This is exactly what is accomplished with a left outer natural join. An

identity between soils and landuse is therefore written:

```
SOILS JOIN+L LANDUSE
```

8.9. Clip. And a clip of cover A using the area of cover B is

```
(A[LOC] INTERSECT B[LOC]) JOIN A.
```

9 Extended SQL

The SQL extensions needed are the same as the extensions to the relational algebra, namely a FOLD clause, and a distance function. These are very minimal extensions compared to some other attempts. The FOLD clause is explicit, and unlike the algebra, is present in every query, similar to the select cause. The query is begun with a "create cover" clause. The reader should keep in mind that the purpose of the SQL is to do spatial overlays, we do not deal with frequently encountered additions dealing with object relationships such as "to the North of" or "adjacent to." These belong to a different class of spatial query.

9.1. Select and Intersect. SQL requires that the join conditions for a join are specified explicitly. In the algebra

```
(SOILS WHERE SUITABILITY = 3)
JOIN
(LANDUSE WHERE LUTYPE = 300)
```

Where JOIN means natural join. This operator would automatically join on common attributes. In SQL one has to be specific, and have prior knowledge of corresponding columns. It is easier therefore to pick the spatial location columns as the attributes that correspond by definition. As the LOC attribute is a guaranteed join item, we have specified it in the following syntax:

```
CREATE COVER goodsites AS
FOLD soils.*, landuse.*
SELECT soils.*, landuse.*
FROM soils, landuse
WHERE soils.loc = landuse.loc
AND soils.suitability= 3
AND landuse.lutype = 300;
```

In this case the results are the same as those for the algebra because there are no other common attributes. If this was not the case, and the values for the other attributes did not agree, while spatial locations did, a hole would appear with the use of the algebra, but not with the SQL.

9.2. Select and Union. The syntax is the same as for the previous example, with the exception of the outer join notation in SQL:

```
CREATE COVER goodsites AS
FOLD soils.*, landuse.*
SELECT soils.*, landuse.*
FROM soils, landuse
WHERE soils.loc = landuse.loc(+)
AND soils.suitability= 3
AND landuse.lutype = 300;
```

9.3. Selection with multiple layers

```
CREATE COVER goodroads AS
FOLD roads.*
SELECT roads.*
FROM roads, soils, landuse
WHERE roads.loc = landuse.loc(+)
AND roads.loc = soils.loc(+)
AND roads.roadclass = 2
AND (landuse.lutype = 300
OR soils.suitability = 0);
```

9.4. Dissolve

```
CREATE COVER goodsites AS
FOLD landuse.lutype
SELECT soils.*, landuse.*
FROM soils, landuse
WHERE soils.loc = landuse.loc
AND soils.suitability= 3
AND landuse.lutype = 300
```

9.5. Bufferzone

```
CREATE COVER goodsites AS
FOLD soils.*, landuse.*
SELECT soils.*, landuse.*
FROM soils, landuse
WHERE soils.loc = landuse.loc
AND soils.suitability= 3
AND landuse.lutype = 300
AND soils .loc IN
        (SELECT soils.loc
                FROM soils, sewer
                WHERE DIST(soils.loc, sewer.loc) < 300);

AND soils.loc IN
        (SELECT soils.loc
                FROM soils, streams
                WHERE DIST(soils.loc, streams.loc) > 20);
```

10 Comparison of Algebra and SQL

The SQL examples show that the algebra is more succinct and elegant than the SQL. Perhaps a different set of extensions will provide better results, but the SQL seems unwieldy and indirect. The use of subqueries for bufferzones is not very satisfactory. The algebra on the other hand is lacking summary operators such as GROUP BY, SUM, AVE and COUNT, which are available in SQL. A spatial equivalent that is needed is AREA. Without these operators meaningful queries can not be made.

Date (1990) suggests a SUMMARIZE A GROUPBY X ADD SUM Y AS Z type of statement, where SUM may be replaced by COUNT, AVG, MAX, and MIN. We suggest dropping "SUMMARIZE A" and adding the capability to GROUPBY contiguous non-universe spatial locations (those that are in the spatial relation) as GROUPBY CONTIGUOUS.

The SUM function needs clarification. The spatial relation is conceptual and has an infinite number of tuples. We must therefore define SUM X as a total figure for the area of the group weighted by the areas within the group for which X is constant. AREA is defined as SUM 1.

Finding those sites then with an area > 2 ha and also computing the construction costs for those potential sites is expressed as:

```
FOLD(
        (GROUBY CONTIGUOUS(
                (       (       (SOILS WHERE SUITABILITY = 3)
                                JOIN
                                (LANDUSE WHERE LUTYPE = 300)
                )TIMES SEWER WHERE DIST < 300
                        )TIMES STREAMS WHERE DIST > 20
                                )ADD SUM 1 AS AREA,
                                ADD SUM COST/HA AS COST)
                                WHERE AREA > 2
        )LUTYPE, COST
```

11 Concluding Remarks

This paper outlines an approach to spatial query language that uses the mechanics and semantics from the relational model. Spatial data can be thought of as conceptual relations through "folding and unfolding." It seems that the advantages of the approach are:

- Minimal algebra and minimal SQL extensions required
- Relational algebra applies
- Approach has significant expressive power for overlay type queries.
- Can be applied to both raster and vector data.

On the other hand we may say that:

- The average user may not be able to think in relational terms for spatial data.
- The approach does not make very much use of topology and therefore does not lend itself to queries related to neighbourhoods
- It does not distinguish between different feature types, such as point, line and polygon.
- It is not clear how such an approach should be implemented.

Since we have presented the approach in terms of an algebra, we can conclude with Date (1990) in saying that *an algebra is often used as a yardstick against which the expressive power of a relational query language can be measured.* This implies that SQL should have the same potential as the algebra, but on the surface it would seem that the equivalent implementation in SQL is more cumbersome and unwieldy than the algebra.

References

Aaronson, P., 1987. Attribute Handling for Geographic Information Systems., **Proceedings** Auto-Carto 8, Baltimore Maryland, pp. 346-355.

Date, C.J., 1990. **An Introduction to Database Systems, Vol. 1**, Addison Wesley, Reading Mass.

ESRI. 1991. **ARC/INFO User's Guide**, Environmental Systems Research Institute, Redlands, CA.

Lorentzos, N.K. and R.G. Johnson. 1987. *A Model for a temporal Relational Algebra,* **Proceedings** of Conference on Temporal Aspects in Information Systems, Sophia-Antipolis, France, North Holland Publishing Company.

Jan van Roessel is a programmer in the Software Development Group of the Environmental Systems Research Institute (ESRI). He develops and maintains software for vector-based spatial data construction and analysis. He is currently the project leader for the development of *geographic regions* as a composite spatial feature type in ESRI's widely sold geographic information system. He has been a member of the Spatial Data Transfer Standard Technical Review Board. Dr. van Roessel did his undergraduate work in the Netherlands and received a MF in Forestry from the University of Washington, a Masters in Engineering Science and a Ph.D. in wildland resource science from the University of California at Berkeley. During the past 25 years he has also been employed by the U.S. Forest Service, The Earth Satellite Corp and TGS Technology at the USGS EROS Data Center.

Dr. van Roessel can be reached by telephone at (909) 793-2853, and by electronic mail using the Internet address jvanroessel@esri.com.