Developing a DBMS for Geographic Information:
A Review

Gerald Charlwood
George Moon
John Tulip

WILD HEERBRUGG LTD
513 McNicoll Avenue, Willowdale
Ontario, Canada M2H 2C9

A summary of the development of a database for a geographic information system. The commonly described disadvantages of the relational model (fixed length fields and an excess of tables) were overcome in a variety of ways, allowing the retention of the advantages of the model. The Binary Data Model (BDM) was used to define the system specifications. A software tool was developed to convert the BDM specification into tables in a relational model and into an object oriented interface to the relational database. A small, dedicated development team followed a strict development cycle, resulting in all major milestones being met. One of the main themes in this paper is the handling of complex (spatial) data that does not obviously suit the relational model.

Introduction

Your mission, should you be so bold, is to construct a database to handle highly structured, multi-purpose geographic information and associated textual data, with display capability, and hooks to independent databases, for huge quantities of data to be randomly updated, with some real-time insertion.

This paper presents a review of the authors' experiences while developing a relational database with support tools to handle both spatial and non-spatial data. [3] describes the concept of the system we have developed. It is hoped that this paper provides a useful narration of events in a moderately large geographic information system undertaking. We review the database history from the definition of the original goals, through design, and development, to optimisation. At the time of writing, the database and tools are still being tuned, with initial feedback from first customers. One of the main themes in this paper is the handling of complex (spatial) data that does not obviously suit the relational model.

How did we decide to proceed? What worked? What didn't? This paper follows the actual development cycle.

A number of key decisions may be identified, which, with the chosen hardware and software, defined our working environment.

The product uses a commercial relational database management system (dbms) on a network of Sun-3 (tm) workstations running UNIX (tm) 4.2. Components of the system are linked with a proprietary inter-process communication protocol.

UNIX, while exceedingly popular, has disadvantages. The caveats on the choice of UNIX are that it is not a real-time operating system, and that it is not optimised for the needs of our application (e.g. job scheduling algorithm). In particular, there is a prejudice against processes requiring large amounts of memory and remaining active for long periods of time. UNIX does provide an available, portable, proven environment with excellent system development and support tools.

As mentioned above, we opted for a relational dbms to run under UNIX. In fact, we use the relational dbms not just to produce a particular database but to produce a custom database management system for our customers to develop their own databases. We provide definitions and support structures for the types of data anticipated in our target applications. It is then up to the end user to define the classes of things desired in their application (houses, roads, utility networks, waterways,...).

The relational dbms gives us the flexibility required to support a host of diverse applications. The relational algebra governing the query language is simple and powerful for end users. The reader is referred to [1] for more information on the relational dbms. As with any approach to any non-trivial task, our path was not without its difficulties. Two standard criticisms of the relational model for spatial data are that relations are implemented as tables with only fixed length fields, thereby wasting a lot of space, and that the large number of tables required in a normalised database (db) is hard on performance. These two problems come together in the fact that each 1:many or many:many relation must be implemented as a separate (binary) table. The relational dbms we use supports both variable length text fields and variable length fields of undifferentiated content ("bulk"). The former allow us and end users to store variable length text without wasting space. More structured information, such as lists of coordinate triples or pairs, can be put into bulk fields, with no wasted space. This addresses the first criticism, in that there is no restriction to fixed length fields. The second criticism is partially addressed also, since information that would otherwise require new tables can be put into bulk fields, so long as there is no need to use the relational algebra. Further handling of this performance question will be described below.

A number of alternatives to our approach exist in the market place. These include the use of a proprietary file structure with no dbms, some proprietary files with a dbms, and use of a dbms without variable length fields. We find that the costs of abandoning the dbms: losing the report writer, transaction logging, security, recovery, and rollback are too great. These same drawbacks arise, to a lesser extent, if a dbms is used with some proprietary files. The use of a dbms without variable length fields was felt to be too wasteful, as noted above.

We found two development paradigms to choose between: requirements driven, top-down, structured design and development or rapid prototyping with a quick turn-around time between prototypes. We opted for the former, although our requirements were incomplete, controversial, mutable, and inconsistent (i.e. normal). As we were not developing the db software in a vacuum - other members of our development team needed tools to work with - we made prototypes available for internal use as quickly as possible. One impact of this necessity was that developing the range of functions was more important than performance for our internal product. We evaluated performance along the way however, with an eye to future improvements. The contents of the early prototypes were the data components we believed to be necessary for our product. These components mainly involved the storing and retrieving of large amounts of topographic data. Graphics support data was added later.

We decided to have a small, tight group build the database and tools, as opposed to a large, shifting or distributed group. The rationale was to create a team atmosphere where intimate working relationships would foster a smooth flow of ideas and mutual assistance.

Contents and Queries

Each geographic database contains a mixture of spatial and non-spatial (mostly textual) data including definitions of the spatial and attribute data to be captured and manipulated, on which a wide variety of queries need to be supported.

The basis of the spatial data is spatial primitives of various topologic types: node, line, and surface. On these are built simple features and triangles. Complex features are built on simple ones. Interactive assistance is provided for defining the structures of simple and complex feature classes customers require.

That is, the spatial data are organised:

- primitives
  - [1] nodes
  - [2] simple lines
  - [3] arcs
  - [4] smooth curves
  - [5] circles
  - [6] surfaces
- features
  - [1] simples
  - [2] triangles
  - [3] complexes

Non-spatial data include:

[1] attributes - text, character, (long) integer, or floating point - of the various primitives and features,

[2] references tying the primitives and features together, sometimes taking the form of distinct tables, and sometimes variable length fields of either text or bulk,

[3] references to attribute data in other databases, which may be external to our system,

[4] definitions of feature classes,

[5] the apparatus to support graphic display, and

[6] support for database management.

Database management depends on the organisation of data into databases called projects, with working subsets, also databases, called partitions which must be checked in and out of projects. This provides a central repository of data (the project) with the capability of multi-user access and update via the various partitions.

In general, each captured piece of spatial data is stored once and may be displayed in a variety of ways, with user selection of which other data is to be displayed. Thus data content is distinct from data display. Selection of data to be displayed is done when the partition is defined. This selection is done by choosing a number of "themes". Each theme specifies classes of data to be displayed, a scale, and graphic attributes for each class. Thus each theme provides a way of displaying a subset (possibly all) of the spatial and attribute data in the partition. Distinct themes may display different data or the same data in different ways.

An issue arising from the complexity of the data structures involved is the management of shared primitives and features. Sharing of primitives and features arises when the flexibility of the data structure allows two or more spatial entities to build on the same primitive or feature (e.g. a road and cadastral parcel might share a boundary linear primitive). If a shared primitive or feature is moved or deleted, all the features referencing it must be identified and updated. Advantages to allowing sharing are that there is a saving of space, and that when a shared primitive is edited, all features referencing it are, in effect, edited. Thus, if a river is a national boundary and the river moves, it is not necessary to also update the national boundary. In cases where two features are desired to be contiguous, but only accidentally, it is easy for the user to create them using no shared primitives. The possibility of shared primitives showed up clearly in the data model and was approved by marketing and users.

The query language sql (tm) is supported by the relational dbms, taking advantage of explicit database structure. There is here a balance to be maintained between the pull of performance which tends to hide structure and the pull of the query language which uses it. For example, coordinate lists for lines may be stored in bulk fields, reducing the number of tables required. On the other hand, standard sql is only of limited use for these lists. We found it useful to extend sql by adding grammar and vocabulary to handle referencing between spatial entities, to handle queries based on the values in bulk fields, and to handle spatial relationships such as overlap, connectivity, and containment. For example suppose we want to select the names of all hospitals in Dover in the Kent county partition in project England database. Note the method of identifying the project and partition in the queries. Assume that "hospital" and "town" are (user-defined) feature classes. Classes town and hospital have defined attribute "name". That is, each town and hospital may have a name. (The user specified, during the definition of the project, whether the name is mandatory and its maximum length.) The first query assumes that each hospital is stored with an attribute "town_name".

```
Select hospital.name from England(Kent)
        where   hospital.town_name = "Dover"
```

If the town name is not available, we can retrieve the hospital names by looking for hospitals spatially contained within Dover. This uses the fact that, in the system, every spatial object has a stored minimum enclosing rectangle ("mer"). This uses an embedded select: first get the mer of Dover, and then compare it with hospital mer´s.

´><´ signifies spatial containment

```
Select hospital.name from England(Kent)
        where   hospital[mer] ><
                [select town[mer] from England(Kent)
                        where   town.name = "Dover"]
```

The other direction of extension of sql is in the handling of data in bulk fields. Selection is supported on values of data elements within structures in a bulk field, and based on the ordinal position of the structure in the list of structures in the field.

For example, it is possible to select lines where the x coordinate of a structure in the coordinate list for the line is greater than (less than, etc.) a given value. That is:

```
select lines from England(Kent)
        where line[coord.x] > 100.0
```

It is also possible to select lines where the first (second, third, etc.) coordinate has a y value satisfying some condition.

Design

We will not deal in this paper with the general question of the architecture of the system, except to say that it is "modular": in working with the system a number of processes must cooperate, communicating with one another. The structure of the database part of the system is presented, along with a description of the methods and tools used in its development.

Five principles governed the design of the interface to the database.

[1] It must be object oriented: presenting objects intelligible to the end user, with components describing the object´s properties and relationships. Objects are described in detail below.

[2] It must shield the users, both programmers and users of the query language, from the underlying tables.

[3] It must use generic low-level update routines to minimise the effort and time involved in development.

[4] It must provide a consistent interface to the data. This interface should use a limited number of routines rather than one routine for each data element. In addition, application programmers should be able to go to a single source to discover the definitions of the objects. These definitions are contained in the TG input (see below). Along with a list of all the objects and all their components, is a description of the data format of each component, with the relevant constraints. These constraints include, but are not limited to, whether the component is mandatory and whether it is read only, write once, or repeatedly writable.

[5] Use of a memory cache of objects would minimise file I/O. This cache should contain the data being actively used by the application. The latter can access data in the db tables (relatively slow) or in the cache (fast).

The db and management tools may be viewed as a layered whole with the relational dbms at the heart. This is surrounded by a layer of utility functions to handle variable length lists, an object cache, data dictionary routines, and generic, db-internal read/write/delete functions. The use of these generic routines is crucial since they rely on generated code and handle almost all cases uniformly. Around this is a layer of application read/write/delete functions, functions to manipulate objects in the cache, and functions to create, delete, move, open, and close databases. This layer provides the consistent, concise application interface. Around this is the world of application programs and the extended query language.

Application layer:	applications, queries
Application interface:	read, write, update routines, cache object functions, db functions
db internal layer:	variable length list functions, object cache, data dictionary functions, generic read, write, update routines
kernel:	relational dbms

From the point of view of an application accessing a database or of an end user, the database contains spatial objects such as houses, roads, nodes, lines and non-spatial objects such as database definitions, graphic transform definitions (for defining display characteristics), and themes.

In general terms, an object is characterised by its properties and its relations with other objects. Its properties include things like its identification number, its class identifier, its name, its minimum enclosing rectangle, or its description. Possible relations include that fact that simple features reference primitives, surface primitives reference lines (and perhaps other surfaces), complex features reference simpler ones, partitions are owned by projects, themes are used by partitions, and graphic transforms are associated with simple features and primitives, given a theme. Note that the latter is a ternary relation (theme + class = graphic transform) which is easily handled in our data model, while causing difficulties for the entity-relationship model. These properties and relations are realised in an object´s "components", which may be of fixed or variable length. Objects give the application programmer, and the query language user a view of the data which is independent of the particular tables involved, and therefore of changes to the underlying implementation. This becomes essential during performance tuning and when there are changes to the BDM specification.

Two tools are instrumental in the design, development, and evolution of this multi-level object: the Binary Data Model ("BDM") [2,4,5] and the table generator ("TG").

In brief, the BDM is a way of handling metadata: a method of analysing, organising, and presenting information handling requirements of a database. It enables system designers to work with end users to agree on a mutually comprehensible specification of the database contents. It is accepted by ANSI/SPARC as the standard for abstract data modelling. From this specification it is a simple algorithm to arrive at tables for a relational database in at least third normal form. The BDM rivals the entity-relationship model, but is more expressive and more readily yields a database implementation of the specified structures.

Results of analysis of the database requirements are expressed in a language which may then be used to produce graphical portrayal of the analysis and to produce input to TG.

Given this input, TG produces a specification of database tables, objects, and mappings between these two views. The generic read/write/delete functions rely on these mappings. Thus, we have an automated environment which goes from a "user friendly" specification of the database contents to database tables, object definitions, and functions mapping between tables and objects.

Advantages, to the end users and developers, of this approach include:

[1] The initial specifications are intelligible to end users and function as computer input.

[2] TG eliminates human error in generating tables, objects, and functions from the BDM specifications.

[3] It is easy to re-run TG whenever the initial BDM specifications change.

[4] TG guarantees that the same algorithm will be consistently applied to generate tables and objects. (People do move on.)

[5] Guaranteed consistency in data representations: if one element of the initial specification occurs as fields in several tables, or as multiple fields in one table,

we are guaranteed that each occurrence of it has the same data format.

[6] The generic read, write, and update routines greatly reduce the amount of code to be produced, thereby reducing costs and shortening the schedule.

[7] The insulation of the applications from the underlying tables makes possible various performance enhancements, without having to rewrite all the applications.

TG and BDM together are an invaluable time-saver, in addition to contributing to the internal consistency of the product and ensuring that what the user saw is what the user will get.

Performance Considerations

Having produced an initial version of the product, having shown the objects and functions described above to be feasible, we turned to performance issues.

There are four areas to look at: profiling of code execution to determine critical modules, attention to inter-process communication, minimising disk I/O, and minimising file I/O.

Rather than spending a lot of time during development trying to optimise all the code and algorithms, profiling of in-house test code and applications was used to determine the bottlenecks. Having found the slow points in execution, there are various remedies. Sometimes it is found that code is superfluous, perhaps because an integrity check is being done twice. Sometimes it is found that an algorithm can be improved upon: perhaps it was originally too general or simply not the best available for the task. The slow points discovered in code execution included:

- interrogation of internal data structures used to convert objects to db tables. The solution was to change TG to generate different mapping structures which support faster access to the database.

- the functions for handling variable length lists. Mechanisms were implemented to force more of the lists to remain in memory.

- updating indices when adding significant amounts of data. It is much faster to drop the indices during update and recreate them afterwards. This assumes that the data has enough integrity to guarantee that there will be no violations when re-creating unique indices. Facilities were provided to allow indices associated with objects to be dropped and recreated.

- queries on the object cache. Cache queries were accelerated by implementing an internal indexing scheme and by modifying the cache organisation.
- Spatial retrievals from the dbms. These are now performed by accessing an internally developed spatial indexing scheme. The indexing method is based on two dimensional extendible hashing. Initially, the indexing software made calls to the variable length list handling functions. This was found to be too slow and was replaced by a layer of software which manages the index directly. Pages from the extendible hash are now cached directly in a memory area of fixed size, and swapped on an lru (least recently used) basis.

Performance increases due to code optimisation ranged up to thousands of percent in some parts of the system. Overall performance has increased by a factor of ten as compared to the initial prototype.

Note that the extended query language is not affected by these changes since the query language software gets data from the db using the application interface layer of the db and is immune to changes to the underlying structures.

For the future, a number of possible paths exist. Two of these involve further reductions of file and disk I/O. The first of these is that cached objects may be stored in a new database, using the bulk fields, with many fewer files than the original. On this approach, we could reduce the number of tables to one, or to the number of object types supported. The basic table layout would consist of a primary key section followed by a data area: the objects would be linearised and stored in bulk fields. One issue here is handling of updates: objects store duplicates of information, unlike normalised tables. Another route would be to develop a table management and caching scheme to reside on top of the commercial dbms. In this scheme, we would map many records into a single relation managed by the dbms vendor. We would be responsible for getting the correct data out of the single relation. The mapping could be based on pages of records. This is not a trivial amount of work. In either case, the object cache manager would be changed to use a cache of fixed size, instead of the present, virtually infinite cache. The cache manager would be responsible for swapping objects or table pages in or out of the cache. A prediction algorithm could be used to ensure that desired pages are in memory as often as possible.

The db bottlenecks we found were, for the most part, very standard, arising from inefficient algorithms and data structures, I/O and the number of tables in the db.

The former problem was dealt with by modifying TG to produce more efficient structures, modifying internal routines to handle these new structures, and by redesigning the object cache to allow fast access to objects in core. It is noteworthy that only the internal routines had to be changed.

The latter problem was dealt with by reducing the number of reads/writes into the relational dbms through better utilisation of the object cache, and by replacing the calls to the variable length list functions with a layer of software to manage a cache of pages of extendible hash indices.

The overall modular architecture made it easy for us to juggle the number of processes and the grouping of functionality into various combinations of processes.

The UNIX 4.2 scheduling algorithm has a bias against large processes. On the other hand, inter-process communication can be a bottleneck, depending on the frequency and size of the information packets being transmitted. A balance must be struck among creating a large number of small processes, creating a smaller number of (large) processes and making efficient use of shared memory for inter-process communication. Initially, our design called for our database software and application software to run as separate processes, with our own inter-process communication software linking them. As there is a huge amount of traffic between such pairs of processes, it was found to be better to combine them.

Conclusions

The standard objections to use of a relational model for spatial data are the performance degradation due to the large number of tables involved and the need to use fixed length fields which waste space. These come together when handling line coordinate lists: either use a fixed length field, of virtually infinite size and waste a lot space, or save space, at the cost of another table and one table access for each coordinate in the list.

The latter objection is simply outdated. Relational db managers are being extended to support tables with variable length fields. Use of a fixed number of fields is not a commitment to fixed length fields. Variable length fields are useful for storing information about an object (e.g. the coordinate lists of lines), for information between objects (referencing information) and for storing whole pages of data.

The problem of the number of tables required is addressed by either linearising objects and placing them in bulk storage - so long as the problem of duplicated data is handled - or by implementing a proprietary table management scheme which would sit on top of the existing dbms.

While we obtain the advantages of a dbms, including transaction logging, security, and rollback, we can use variable length fields of text or bulk to avoid the problems inherent in a strict relational model without variable length fields.

The close-knit database development team met all its major milestones, and adapted well to shifts of direction and the changes required in tuning performance.

The use of the binary data model gives us a precise specification which users can evaluate, so there are no surprises when the system is delivered. Its use with the table generator gave us the ability to respond quickly and easily to changes in requirements: eliminating the need for repeated hand-crafting of huge amounts of crucial code. In addition, TG guarantees, within the limits of its algorithm, that what was specified in the BDM is what is built. The use of the binary data model and TG greatly enhanced the group's ability to supply the needed functions.

A modular, multi-process architecture allows us to optimise our use of the underlying UNIX environment by using a reasonable number of moderately large processes, with a balanced amount of inter-process communication.

Acknowledgements

We would like to acknowledge the support of Wild Heerbrugg, Heerbrugg, Switzerland during the preparation of this paper which is published with their permission. Details expressed herein are subject to change without notice.

"sql" is a registered trademark of IBM.

"Sun-3" is a registered trademark of Sun Microsystems.

"UNIX" is a registered trademark of AT&T.

Bibliography

[1] Date, C.J. 1983 - "An Introduction to Database Systems: Volume II", Addison-Wesley

[2] Mark, L. and Roussopoulos, N. 1986 - Metadata Management: Computer December 1986, volume 19, number 12, pp. 26-36

[3] McLaren, R. and Brunner, W. 1986 - The Next Generation of Manual Data Capture and Editing Techniques - The Wild System 9 Approach: Proceedings 1986 ACSM-ASPRS Annual Convention, volume IV, pp. 50-59

[4] Nijssen, G.M. and Carkeet, M.J. A comparison of Conceptualisation and Normalisation: University of Queensland

[5] van Griethuysen, J.J. (ed) 1982 - Concepts and Terminology for the Conceptual Schema and the Information Base: publication number ISO/TC97/SC5 - N 695.