Relational, Tree/Plex, and Object Oriented Databases

by Arthur B. Smith

This paper presents an informal comparison of three different foundations for database construction (models). It focuses especially on the relative strengths and weaknesses of each technique and the characteristics of database applications that best fit each of the three models. It should be noted at the outset that any of these techniques can be used to model any data system; it is a fairly mechanical process to convert between the different data models. The advantages lie in convenience and simplicity for the database developers, maintainers, and users. Advantages come in two forms: theoretical advantages of one technique over another and pragmatic advantages, which often deal more with marketing realities than computer and information sciences. Both of these factors will be examined.

The relational (or tablebased) database model is by far the most frequently used today and is well represented by such large commercial packages as Oracle, Sybase, Informix, Ingres, and Gupta as well as smaller packages such as DBaseIV, Access, FoxPro, Alpha4 and Paradox. All of these trace back to the pioneering theoretical work by Codd and others in 1970. Of the three systems, the relational model is the most well grounded mathematically, supporting both a relational algebra and a relational calculus. The latter is the basis of the query language SQL and its superset ODBC, which are widely used for user interface communications in two- or three-tiered applications.

The tree/plex model is represented by many of the earliest database languages and systems including IBM's DL/1, M, and the Data Descriptor Language CODASYL. All of these systems directly represent treestructured data and can handle "plex" data (which may have many-to-many relationships) to varying degrees. The mathematics governing these databases is the mathematics of graph theory, as these databases have structures which are directed graphs. In many implementations of these systems it is possible to construct a (possibly limited) mapping to a relational model which allows the use of SQL/ODBC as a query-level interface.

The object oriented databases are relatively new and still rather primitive in some respects. Most of the current implementations such as Objectivity, Poet and Versant, are somewhat lacking in programmer and user conveniences, and the database theory itself is not as mathematically well-grounded as relational and tree/plex databases. These should not necessarily be taken as signs of inherent weakness in this modeling technique, however. Rather, they are indications of the youth of this technique. Serious effort is under way to remedy both of these shortcomings, and both relational and tree/plex database systems are rushing to adopt object oriented models, usually on top of their own inherent structure.

This paper will attempt to show the relative merits of each of these techniques and indicate the characteristics of applications that will benefit from one technique over the others. Each of these techniques has its place, and careful selection of the database model is an important early step in the design of any large database application.

Relational Database Management Systems

Theoretical issues

All relational databases are based on two-dimensional tables as the model for storing data. This model is chosen because it is generally familiar to all users and is seen as a "natural" way of representing the data. Any data system, no matter how complex, can be reduced to a collection of tables (or "relations" in the terminology of RDBMSs) with some redundancy. The redundancy is controlled by forcing the relations into canonical "Normal" forms which minimize unnecessary redundancy without sacrificing associations between data elements.

Each relation (table) can be represented as a rectangular array with the following properties: 1. Each entry in a table represents exactly one data item; there are no repeating groups, 2. Each table is column homogeneous; all items in any column are of the same kind, 3. Each column is assigned a distinct name, 4. All rows are distinct; duplicate rows are not allowed, 5. Both the rows and columns are sequence independent; viewing either in a different sequence cannot change the information content of the relation.

Each row represents a single item which is being described. The columns represent the distinct pieces of information (data elements) which are known about the item. Rows are commonly called "records" and columns are called "fields."

In addition, the operations permitted on these relations are limited to Inserts and Deletes of records (Edits are implicitly permitted as a concatenation of an Insert and a Delete.), Joins (in which a temporary relation is constructed by combining the information in two relations using common fields) and Selects (in which a subset of the records in a relation are selected based on specific values or ranges of value in selected fields).

Other manipulations of the data are not generally supported by relational databases. Addition of ad hoc data, which does not conform to any fields in data definition, for example, is prohibited. Adding a field to allow ad hoc data to be entered would require restructuring the database, often a lengthy process which can only be done when the database is not in use.

In general, few real-world databases can be represented using a single relation table. Most applications use multiple relations which contain columns (fields) with the same name. These common data allow the joins of two or more relations to form meaningful associations. This is best shown by an example. Consider two relations, "EMPLOYEE" and "DEPARTMENT" shown in the following diagram:

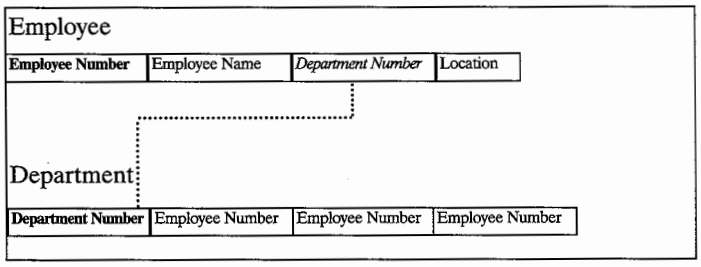

Fig. 1

In this example, the Employee Number and Department Number fields are bolded, indicating that these are the primary key fields. This means that data elements in these fields uniquely identify the row (i.e., no two rows have the same data element in the key field). Furthermore, the Department Number field is found in both relations. This allows a "join" of the two relations so that, for instance, the Department Name for any given Employee could be determined.

In many cases, the join is not so simple. Suppose we need to find a way to determine the manager for any employee. We could construct the following data structure:

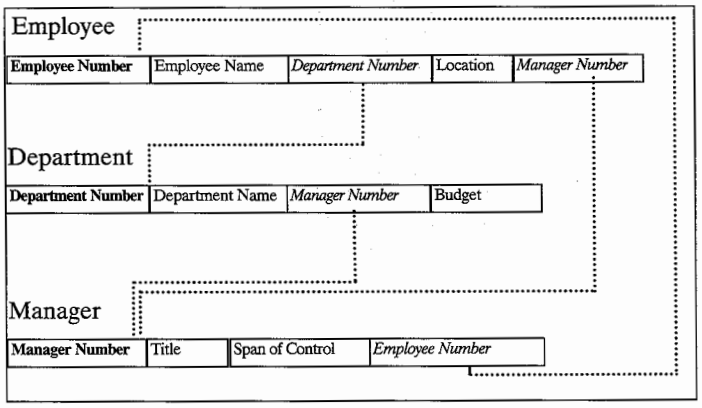

Fig. 2

This seemingly intuitive form may cause problems because of the redundancy of the Manager Number link from Employee directly and through Department. This redundancy allows an employee's manager to be different from his department's manager. The above structure would be inappropriate if that is not allowed. Instead, a more appropriate structure would be:

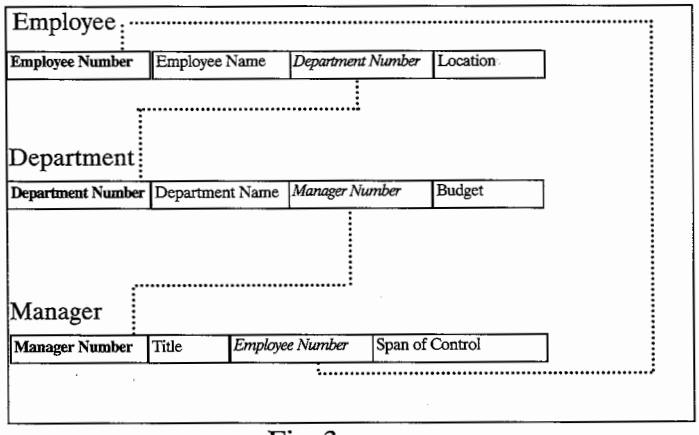

Fig. 3

This structure has removed the redundancy which allowed an Employee to have a separate Manager from the Manager of his Department, but in doing so has

eliminated the direct link which may be desirable for performance reasons in a large database. These trade-offs between performance and data integrity are common in virtually all database models.

A simple, real-world example can require an even more complex structure. Consider an Employee who is a member of more than one Department. The rules of joins in a relational database do not allow many-to-many links (which might be designated with an arrow which is doubleheaded at each end). To represent a many-to-many relationship (e.g., each Department has multiple Employees, and each Employee can be a member of Multiple Departments), we need to construct a separate relation which is the crossproduct of the two columns:

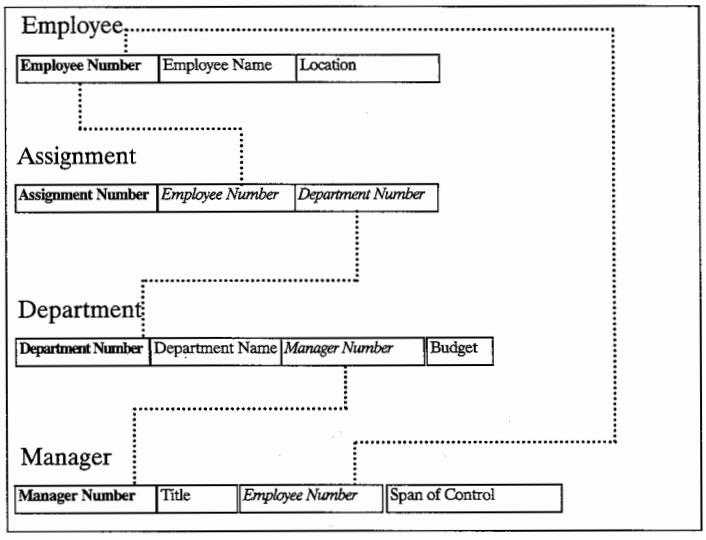

Fig. 4

Here the Assignment relation contains a row for every employee department assignment. That is, if an Employee is part of a Department, then the respective Employee Number and Department Number are found in exactly one row of the Assignment relation. The Assignment Number field is actually unnecessary because the Employee Number and Department Number can jointly serve as a key. Most (but not all) RDBMSs allow relations which have a multiple field key. For those which do require a single key field in each relation, the structure would be as shown above.

The "natural approach" of using tables may become even more strained when the data is sparse. Sparse data means that not every field in every row contains data. In some applications, data is very sparse—only a few of the many defined columns for a relation may contain data in any given row. When sparse data is the norm in a real world application, the user typically does not think of the data as tabular. For example, consider a relation for a patient encounter (visit). It could contain fields which give key values for radiographs, ultrasounds, MRIs, CTs, endoscopies, CBCs, Electrolytes, Creatinine/BUN, Pulse, Respiration, Temperature, Weight, and so on. Most of these fields would be blank for any given encounter. Thinking of these as fields in a table is at best somewhat counterintuitive. It is more natural to think of these data as a list of entries rather than fields in a table. This "userview" is not easily supported in a relational model; these different entries cannot be stored in a single table as they violate column-homogeneity—the data stored in the field are not all of the same kind as required for relational tables.

Relational databases would be very inefficient if this sparse data were actually stored in a rectangular array internally, since space would be allocated for these empty data elements. In actual practice, most modern RDBMSs use tables as a "logical" structure, but use an internal "physical" structure which can more efficiently store sparse data such as a BTree. The choice of physical structure is sometimes left to the database designer as a performance tuning issue.

As a second example, consider a simplified version of a computer-based patient record (CPR). In this database

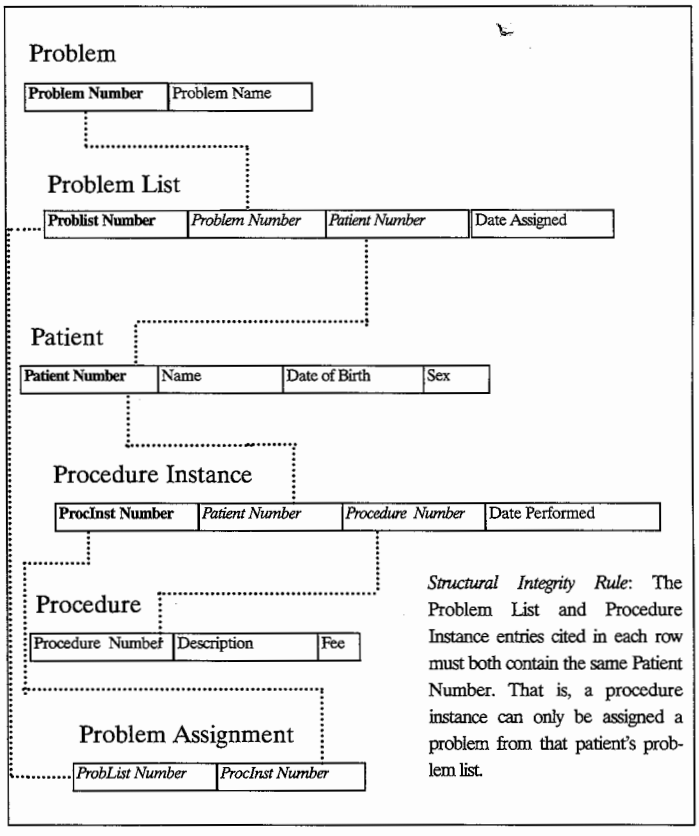

Fig. 5

we have information about patients and procedures which are done on these patients (these procedures might be examinations, laboratory tests, etc.) Patients also have a set of problems which have been identified, and any of the procedures may be identified with one or more of these problems. A relational structure for this database is shown in figure 5.

Figure 5 clearly shows how the complexity of a data model can grow when the relational constraints are applied to data which are not inherently tabular. The conceptual description of the database invoked only patients, procedures, and problems. Because of the many-to-many relationships between these items, it was necessary to introduce second-level relations (Problem List and Procedure Instance) and one third-level relation (Problem Assignment). It also required the introduction of a structural integrity rule for the third-level relation in order to prevent introduction of nonsensical data (a procedure for one patient assigned the problem of another patient).

Other relational structures can be used to represent this data, but they cannot be simpler without either limiting the flexibility of the database (e.g., by making it impossible to assign a problem to a patient without a procedure being assigned that problem) or violating the accepted rules for a normalized relational database (Third Normal Form). The argument for the "intuitiveness" of a relational database that has been advanced by Codd and others seems somewhat strained in situations like this. This is not to say that the argument is specious, merely that it does not apply equally well to all sets of data.

Pragmatic issues

The relational database model is currently the preeminent database model in the computer industry and has held this position for approximately 20 years. This position of strength, ironically, may end up becoming a position of weakness. While relational databases have come to dominate the database industry over the last twenty years, many experts feel that they are reaching the end of their lifetime as a dominant player, to be superseded by more modern and flexible models, notably the object oriented database model. There is some question as to whether the giants (Oracle, Informix, Sybase, Gupta, etc.) will be able to make the change in a timely enough fashion, given their huge installed customer base. The need to maintain backward compatibility with this customer base can severely hamper their abilities to keep up with advances in database technology. On the other hand, the comfortable financing of this large customer base makes development easier. All of the large relational database vendors are adding object oriented tools layered on top of an underlying relational structure. These tools are useful during database design, but do not appear to offer any advantages at run time, since the underlying relational model is preserved.

If you are looking for a mature, well-documented, well-understood, well-maintained product for today, the RDBMSs are clearly the leaders. If you are looking for those properties ten years down the road, they may or may not be. Many experts expect there to be a major paradigm shift in database technology. Of course, predicting that some other company will be the next Oracle is easy; predicting which one will be the next Oracle is worthy of an oracle of a different sort....

Tree/Plex Database Management Systems (e.g., M)

Theoretical Issues

The structure for these DBMSs is based on trees and directed graphs rather than tables. In the tree-structured databases, the data is structured as a set of data nodes. Each node of the tree can have data stored with it and can have any number of (directed) links to child nodes. There is generally no requirement that the child and/or sibling nodes be of like type. It usually is possible for a node to contain an open-ended list of links to multiple nodes of like type, allowing direct representation of many-to-many relationships.

In plex structured databases, the tree requirement of each node having exactly one parent (except for the root which has zero parents) is relaxed. Links may be restricted to forming a directed acyclic graph (DAG) (no sequence of directed links from a node can lead back to the node), or may allow any directed graph (a set of nodes and directed links between pairs of nodes with no other restrictions).

M implements the tree-structured database directly (child nodes are conceptually components of the parent node) and allows external links (single or in arrays) to any other nodes.

The example we have used for relational databases might be implemented in M with the following structure (assuming employees are part of just one department).

In this diagram, the parent node is the Department. Each department has a single Manager child and an array of Employee children. Note that the Manager is simply an Employee, identified by the Manager subnode which points to this special employee with the Employee Ptr. No reverse direction is needed on this pointer—an employee's manager can be determined by looking at the Manager child of the employee's parent node (Department). Note that the structure of the data is largely reflected in the structure of the database without the use of separate linkages.

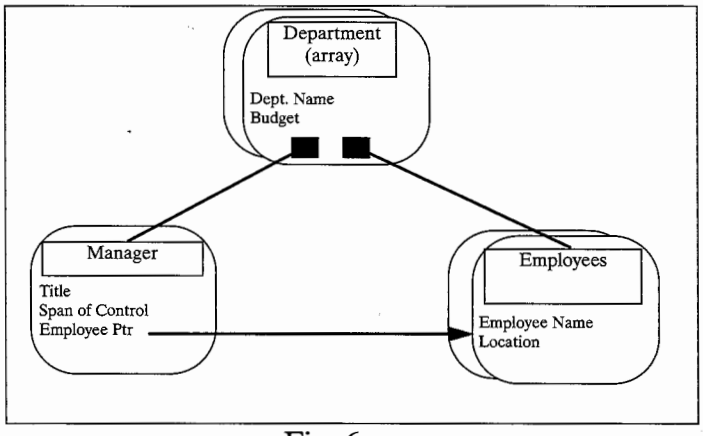

Fig. 6

This structure, however, is inappropriate if the assumption that an employee is in only one department is false. It also deals poorly with the situation when an employee changes departments, even if he is only ever in one at a time. This is because instead of merely changing a pointer (typically a single number) an entire set of data (the employee) must be moved from inside one data element (department) to inside another. While this usually does not require actually moving the data stored on the computer, it is almost always more complex than shifting a pointer.

A more appropriate structure is shown in Fig 7. It contains two types of top level nodes, Department and Employee, both of which are arrays (i.e., there may be multiple distinct instances of each). Each Department has an identified Manager, which includes a pointer to an Employee instance (the Employee who is the manager of the department). Each Department node also has an array of pointers to Employee instances, representing all the employees in the department including the manager (so that appointing a new manager doesn't automatically remove the outgoing manager from the department).

Similarly, each Employee has an array of Department

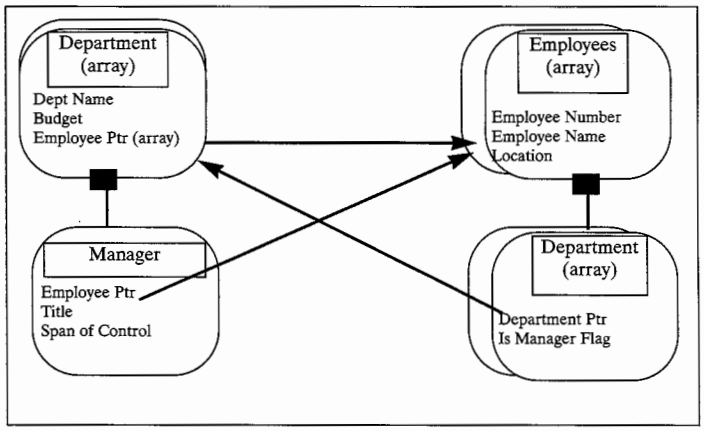

Fig. 7

subnodes identified, each of which consists of a pointer to the Department node and a flag to indicate if this employee is the manager of that department. Employees can be members (or even managers) of multiple departments, simply by having multiple Department instances defined within the Employee. This data model contains all the same relationships as the relational example that was given above. While the structure of individual components is more complex than the individual tables of the relational model, the data model as a whole may be more intuitive in this presentation.

Let us turn our attention now to the second example—the simplified patient record. Using a tree/plex structured database, we can construct the following data model for this database (see Fig 8).

Problem List Number Date Assigned

This data model has all of the same structural information as the relational model we looked at earlier (including the constraint that a procedure instance can only be assigned a problem from that patient's problem list, which required a structural integrity rule in the

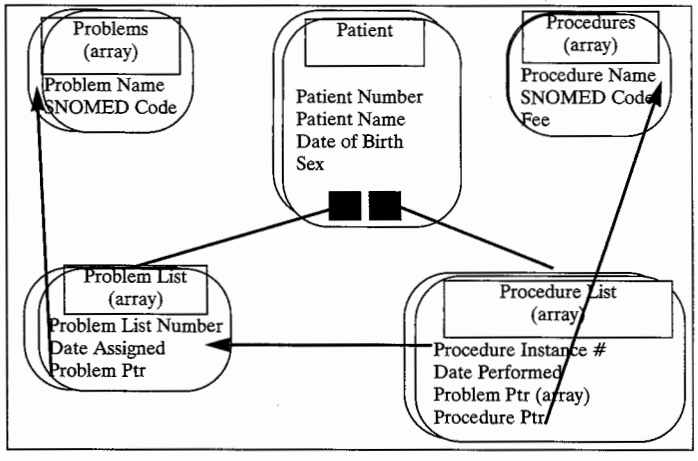

Fig. 8

relational model). Again, the structure of the individual components is more complex than the tables used in the relational model, but the overall structure is much more intuitive, particularly since the structural integrity is guaranteed without external rules.

Pragmatic Issues

The structures defined above are based on the M language/database system. This is an ANSI Standard programming language that has been around for over twenty years and has undergone a number of substantial revisions, most recently in 1995. In terms of market share, M has never been large and has been shrinking in relative quantity (but increasing in absolute quantity) for several years. It is used primarily in health care and banking, but is also used for a number of niche applications such as yellow pages production. M is declared dead at frequent intervals, but nevertheless continues on.

A weak spot in most M applications is the user interface which until recently had to use the "roll-and-scroll" presentation model of the seventies, designed around dumb or moderately smart terminals. Within the last two years, most major M vendors have added facilities to allow development of sophisticated user interfaces using third party development tools. Full-featured interface development tools, such as those in many relational databases, have yet to be developed for M. The relatively small market share commanded by M slows development by M vendors (due to lack of funds) and makes the market unattractive to most large third party developers. Current trends in third party development tools towards adoption of open standards has made them much more accessible to M.

Most M systems now allow M to communicate freely with relational databases using SQL or ODBC. M can serve as either the "front end" or the "back end" in these joint ventures. To use this bridge, however, the database administrator must define a mapping of the M data structure to a relational model, since SQL and ODBC both assume this model.

There is a substantial effort in the standards body for M to extend the language to fully support object oriented programming and object oriented database design and management. The flexibility of M data structures and the late binding of M data types allows a "true" object oriented approach (with the possible exception of Encapsulation) as opposed to the "layered" approach of object oriented tools implemented on top of a relational database.

If M becomes an object oriented database system and vendors supply interface development tools, it could become a major player in the industry. Without this, it is probably destined to remain a relatively minor player in the database industry with the exception of the niches in which it has a loyal following

Object Oriented Database Management Systems (OODBMS)

Theoretical Issues

The object oriented database is a relatively new concept and, as such, is somewhat less well-defined. The requirements for a database being "object oriented" differ depending on who one talks to. The features that seem to be common in most implementations are:

1. ***Abstraction:*** Each real world "thing" that is kept track of in the database is a member of some "class." This class defines all properties, methods, public and private data structures, and routines which can be applied to "objects" (instances) of this class. A class defines an abstract data type. A "method" is a procedure invoked to do something to or with an object (e.g., to print itself, or copy itself). A "property" is a data value associated with each object of a class that identifies the state of that object in some way (e.g., color, age). Not all implementations support properties—they are short-hand for a method with no arguments (e.g., report your color, report your age).

2. ***Encapsulation***: The internal representation of the data and the implementation details of public and private methods (routines) is part of the class definition and is known only to this class. Access to objects of a class is only allowed through the properties and methods for that class or its parents (see inheritance, below) and not through knowledge of the internal implementation details.

3. ***Inheritance*** (single or multiple) Classes are defined as part of a class hierarchy. Each lower level class definition inherits the properties and methods of its parent, unless they are explicitly not inherited, or are overridden by a new definition. In single inheritance, a class can have only one parent class (i.e., the class hierarchy is tree structured); in multiple inheritance, a class can be descended from multiple immediate parents (i.e., the class hierarchy is structured as a directed acyclic graph (DAG), but not necessarily a tree). Not all object

oriented databases support multiple inheritance.

4. ***Polymorphism***: Multiple classes may have the same name for their methods and properties even though they are deemed different. This allows access methods to be written which will work appropriately with objects of very different classes, as long as they have the appropriately named methods/properties defined. For example, the print method may be defined in many classes, but perform differently when invoked, based on the target object's class.

5. ***Messaging:*** Interaction with objects is done by sending a message to the object and (optionally) receiving a message in response. This differs from calling a procedure (as is used in most other models). To invoke a method on an object, one sends a "method yourself" message to that object. The messaging paradigm is not always used in object oriented databases, but is used in the "truer" object oriented implementations.

In an object oriented database, each type of thing being kept track of has a class associated with it, and the links between the classes are handled by properties and methods of the classes. To get a feel for how this is done, we will again look at our two simple examples. In the first example, we will define two top level classes: Department and Employee. We will also define a class, Manager, which is a subclass of Employee. The properties and methods which are inherited are shown in italics.

Objects (instances of a class) can be created (instantiated) by invoking the Instantiate method which is common to all classes. An employee object, once instantiated, is assigned to the department by invoking the department's Hire method (with the employee as an argument).

To make an employee a manager, one invokes the Promote method on the employee (with the department as the argument). This instantiates a new object of class Manager, duplicating the inherited properties and then destroys the existing Employee instance (using the Destroy method, also common to all classes). It returns the new object reference (handle). Note that this method also needs to update the Employee List property for each department in the employee's Dept List to reflect the new ObjRef. Since Manager is a subclass of Employee, this Employee List will not have any type conflicts (an object reference to an Employee can, in fact be an object reference to a Manager, since that is a subclass of employee—the reverse does not hold true).

Note also that the inheritance of the Promote method is explicitly blocked in the Manager subclass (this is usually done by defining a Promote method at the Manager class level which does nothing). This is appropriate if an employee can only be the manager of a single department. If a particular employee can manage multiple departments, the Promote method must apply to managers as well. In this case the Promote method must also update the Dept Manager property in any departments in the Dept List property of the Manager.

EMPLOYEE	
Property	**Type**
Emp Name	Text
Emp Number	Numeric
Location	Text
Dept List	List of ObjRef:Department
Method	**Returns**
Promote(Dept)	ObjRef:Manager

Fig. 9

MANAGER	
(subclass of Employee)	
Property	**Type**
Emp Name	*Text*
Emp Number	*Numeric*
Location	*Text*
Dept List	*List of ObjRef:Department*
Span of Control	Integer
Title	Text
Method	**Returns**
<Promote>	*explicitly not inherited*

Fig. 10

DEPARTMENT	
Property	**Type**
Dept Name	Text
Budget Dollar	
Employee List	List of ObjRef: Employee
Dept Manager	ObjRef: Manager
Method	**Returns**
Hire(Employee)	<nothing>

Fig. 11

The object oriented implementation of the second example (the simplified patient record, Fig. 5) can be constructed in two different ways depending on whether or not properties can have structured values. In either case there will be a class for (abstract) problems and another for (abstract) procedures. The difference occurs in the construction of the patient class and any additional classes. If a property value can consist of separate pieces, the problem list and procedure list for a patient are probably best described as part of the patient object, consisting of lists of structured values as shown in figures 12-17.

ABSTRACT PROBLEM	
Property	**Type**
Prob Name	Text
Code	Text (fixed length)
Method	**Returns**

Fig. 12

ABSTRACT PROCEDURE	
Property	**Type**
Proc Name	Text
Code	Text (fixed length)
Fee	Dollar
Method	**Returns**

Fig. 13

PATIENT		
Property	**Type**	
Pt Name	Text	
Pt Number	Numeric	
Date of Birth	TimeStamp	
Sex	M/F	
Problist	List of:	
	ProbList Number	Numeric
	Date Assigned	TimeStamp
	Abstract Problem	ObjRef:Problem
ProcList	List of:	
	ProcList Number	Numeric
	Date Performed	TimeStamp
	ProbList Number	Numeric
	Abstract Procedure	ObjRef: Procedure
Method	**Returns**	
Add Problem(Prob Name)	ProbList Number	
Add Procedure(Proc Name)	ProcList Number	
Assign (ProcList Number, ProbList Number)	< nothing>	

Fig. 14

Not all object oriented databases allow these structured properties. In this case, Abstract Problem and Abstract Procedure classes are unchanged, but the Problem List property becomes a list of references to Problem Instance objects (a new class), and the Procedure List property becomes a similar list of refer ences to Procedure Instance objects (also a new class). The Instance classes should not be confused with the Abstract classes defined above.

In some object oriented databases, notably those built on top of a relational model, properties cannot contain a list of data. In this case, not surprisingly, the class def-

PROBLEM INSTANCE	
Property	**Type**
Problem	ObjRef: Abstract Problem
Date Assigned	TimeStamp
Method	**Returns**

Fig. 15

PROCEDURE INSTANCE	
Property	**Type**
Procedure	ObjRef: Abstract Proc.
Date Assigned	TimeStamp
Problems	List of ObjRef:Abstract Problem
Method	**Returns**

Fig. 16

PATIENT	
Property	**Type**
Pt Name	Text
Pt Number	Numeric
Date of Birth	TimeStamp
Sex	M/F
ProbList	List of ObjRef:Problem Instance
ProcList	List of ObjRef:Procedure Instance
Method	**Returns**
Add Problem(Prob Name)	Problem Instance obj ref
Add Procedure(Proc Name)	Procedure Instance obj ref
Assign (ProcList ObjRef, ProbList Ob jRef)	<nothing>

Fig. 17

initions will closely resemble the relational structure for this problem examined earlier.

The advantage to the object oriented approach lies in shifting the focus away from the data structure (in particular the form of the links between data types) to the process by which these data are established, modified and destroyed. The actual data structures are an implementation detail best left to the inner workings of each class—and different classes may have very different implementations to handle efficiency tradeoffs. Maintenance of the database is accomplished just by knowing the methods and properties the classes make available.

Pragmatic Issues

The object oriented data base model is at a higher level of abstraction than the relational or tree/plex database—the implementation of a class can be accomplished using either of these models, or some other model. Since the design focus is more concentrated on process than structure, however, it is important that one select an underlying model which has sufficient strength, flexibility, and efficiency for processing to allow appropriate construction of methods.

Relational databases with their strict definition of structure and limited set of allowed operations are arguably inappropriate underlying platforms for an object oriented database. The M language/database system, with its more flexible data structure and more procedural approach, appears at first particularly well-suited to serving as an underlying platform for an object oriented database system. It appears that object oriented approaches in M can outperform similar approaches in relational databases in speed of access and manipulation.

There are a number of commercial offerings that are "pure" object oriented database systems. These include Objectivity, Poet, and Versant. In general, these systems are relatively immature and lack the development and support tools that are needed for large-scale database implementation and maintenance. Furthermore, these object oriented systems seem to be quite slow at accessing large databases, lacking the efficiency of the relational or tree/plex databases. While very large, complex databases exist today in both relational and tree/plex systems, the databases constructed using object oriented databases are relatively small and simple. This is widely viewed as a sign only of the immaturity of the technology and not the limit of its potential.

Summary and Recommendations

The three different techniques for representing and manipulating data in a database each have their strengths and their weaknesses. The relational model is the most well-established and supported today and will likely remain so for at least the next several years. It is also the most strictly structured paradigm, which works both for it and against it. The functional requirements of the database system virtually dictate the structure of the database, creating a more uniform style which can be easier to support. On the other hand, data which does not naturally fit the table metaphor requires particularly complex and counterintuitive structures, which make support more difficult. In addition, a relational database is defined almost entirely by its structure, rather than by procedures or methods. This makes it easy to document the database using data dictionary tools and requires less "back-end" programming. On the other hand, if the functional specifications of the database are unstable or evolving, changes must be made at the data dictionary level, usually involving periods of data inaccessibility and a revision of the structural documentation.

A tree or plex structured database (e.g., M) is much less inherently structured than an equivalent relational database. This allows greater freedom in the construction of the database which, in turn, allows for increased efficiency and/or a more "natural" modeling of the data. In addition, the flexibility and procedural nature of these databases make them more easily adapted to changing functional specifications as the database system evolves. Additional and ad hoc data elements can be added without disabling database access, and new operations can more often be added without structural change to the data.

On the other hand, the increased flexibility also carries a much greater burden in terms of documentation and local standards for structure and process. Failure to adequately document both the procedural and structural natures of the database lead to maintenance problems well beyond those seen in relational databases. The burden of documentation may well be too severe if the data being modeled fits naturally into two-dimensional tables.

The object oriented approach to database systems has been widely heralded as the next major paradigm, replacing the relational model over the next several

years. It does allow an increased flexibility over the relational model, and focuses the development and documentation on the individual types of things being recorded rather than the overall structure of the database. This promises to be particularly useful in constructing complex database systems which have many different kinds of data, particularly if there are frequent many-to-many relationships which do not fit naturally into tables.

Unfortunately, the object oriented approach is largely untested and is not available today in the mature form of the other two approaches. The most accessible approach to an object oriented database is one layered on a relational or tree/plex model. Since the object oriented approach is much more concerned with process than structure, it appears that the tree/plex model is the better fit. Construction of an object oriented database using a relational framework sacrifices the freedoms enjoyed during the design phase when it comes to implementation and maintenance.

The choice of database model then, comes down to a careful examination of the data to be modeled and the resources available. Databases which are stable and well defined and which fit naturally into two dimensional tables, and which require large-scale support are best constructed using a relational model such as Oracle or Sybase. Databases which are evolving and complex with multiple many-to-many relationships and sparse data, and which are defined largely in terms of process are best suited to the tree/plex model of M. Relatively small databases which are defined largely by the objects they describe and the actions those objects can perform are perhaps well-suited to object oriented approaches. Whether the anticipated object oriented database revolution is realized by layered approaches on top of relational models or M, or by native object oriented platforms unfortunately remains to be seen.

While the above considerations are appropriate for new database projects, they neglect an important factor in upgrading existing databases, namely the system and staff resources already present. All of the systems shown are generally able to model the same data. Comparison tests between different systems are rare and often contradictory, but where they seem to agree is that a database designer/implementor familiar with any of these techniques can construct a more efficient database (in terms of both time and space) using the tools with which they are most familiar. A significant existing investment in personnel and/or software may easily override any other factor in determining the best underlying platform for database construction. **M**

Art Smith received his masters degrees in chemistry and computer science and information systems at the University of Delaware in 1985 and has worked in scientific and database applications since that time. He now works for the University of Missouri Veterinary Medical Teaching Hospital and does consulting with Emergent Technologies.
