The Integrated Data Model: A Database Perspective

David Beech and J. Samuel Feldman

Computer Research Center, Hewlett-Packard Company
Palo Alto, California 94304, USA

Abstract

The Integrated Data Model integrates concepts from three information disciplines: database systems, artificial intelligence, and programming languages. Key concepts are those of abstract type (allowing multiple implementations), object (possibly having multiple types), and relation (definable by logical formulas). This paper provides a brief overview of the model from a database perspective.

1. INTRODUCTION

The Integrated Data Model (IDM) provides a more general and flexible foundation for the manipulation of information than models underlying traditional database management systems. The facilities provided by the model can be employed not only for database queries, updates, and report generation, but also for managing the arbitrary data structures used by systems programs and application programs. Moreover, because this occurs in a database setting, the ability to share this information concurrently with other users and to distribute information among different sites is provided in the same way as for the more conventional kinds of data routinely stored in databases.

IDM takes data abstraction as the fundamental underlying idea, and draws together some concepts previously employed for database systems, artificial intelligence applications, and programming languages. A data model is fundamental to each discipline. Database systems are explicit about this, but in fact every programming language and every artificial intelligence system also embodies a data model. We propose that a single data model can combine the best aspects of all three disciplines and remove many of the shortcomings of each.

1.1 Storing New and Different Types of Data

Current database systems each support their own static set of built-in datatypes. Although work is continuing on extending the implementations of these systems so that "unformatted" data may be stored, such measures are essentially patches.

One important consequence of our application of recent programming language ideas to databases is that new datatypes can be defined as needed, and values of these types may be stored in the database in the same way as other kinds of data. This is accomplished in our design by our treatment of datatypes as abstract objects, which provides a formal framework for defining and using new types dynamically. Another way of looking at this capability is that programs may use the same kinds of data structures for persistent data as for temporary data.

1.2 Natural User Interfaces

The intent of the design is to provide a model that corresponds naturally to the way we perceive information, while still providing sufficient power and allowing for efficient access. We accomplish this partially with a form of entity-relationship model [Childs 68; Abrial 74; Chen 76]. Models of this genre are increasingly being employed for user interfaces and database design tools [Dahl & Bubenko 82; Olle *et al.* 82; Wong & Kuo 82], because of their suitability for representing the user's conceptual model. Attempts to extend the relational model to introduce an entity concept seem awkward [Codd 79]. (We have in fact designed an English-like database manipulation language for our model as one example of a natural user interface.)

2. OBJECTS AND TYPES

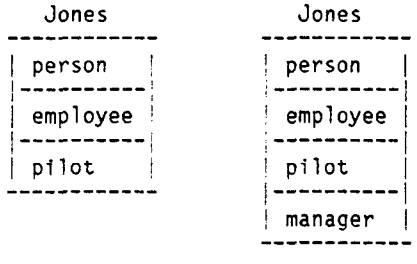
The central, pervasive concept in IDM is that of the *object*. Everything in an IDM database is an object; even the database is itself an object. The simple data abstraction idea is extended in two significant ways:

- An object may have more than one type.
- An object may gain and lose types dynamically.

We must now clarify the meaning of "type". Each type groups together objects that behave similarly. One might define types such as Employee, Account, Document, or Image, for example. Every object is said to be an *instance* of one or more such *types*.

As in a programming language with data abstraction, the type provides a set of *operations* that define the way instances of that type may be manipulated. Since an object may have more than one type, corresponding to the different roles it may be viewed as playing, it may be manipulated via the operations of several types.

As an example, suppose that Jones is a person, an employee and a pilot. If Jones later becomes a manager, the type "manager" can be added as shown.



Each type provides a separate set of operations that can be invoked on Jones. For example, among the operations defined by the Person type might be GetAddress and SetAddress, the Employee type GetManager and SetManager, and the Pilot type GetLicenseNumber. Each type added to an object supplies specific additional capabilities, by providing additional operations.

Types are full-fledged objects. Certain types are predefined by the model; other types can be defined by users. The latter is accomplished by creating a Type object and binding operations to it; the operations are written (in one of several programming languages) as functions which take parameters and return a single result. Operations are treated as objects also.

To summarize our concept of an object:

- Everything in an IDM database is an object.
- Each object is an instance of one or more types.
- Each type defines the operations that are available for its instances.
- The only action available in IDM is to invoke an operation on an object.

3. RELATIONS

Relations, like databases, types, and operations, are objects—each relation is an instance of the predefined type Relation. In many object-oriented models, information is carried in the *attributes*, *properties*, or *components* of objects. In our model all such information is regarded as associating one object to another, and is expressed by relations. For example, one might define a relation between employees and their addresses; or between employees and their departments; or between documents, their creators, and their creation dates (these types are termed the *domains* of the relation).

The association of a particular employee with a particular department is represented by a *relationship*. The relation represents the concept of department membership, by defining the relationships that are valid at any given moment.

3.1 Manipulation of Relations

The type Relation provides operations to find, insert, update, or delete specific relationships. For example, a relation can be queried to find all the employees in a given department, or, since relations in our model are inherently symmetrical, the department of a given employee.

It is sometimes more natural to carry out these manipulations as operations on the instances of the domains. For example, one might want to ask a particular employee object what its department is. This is easily accomplished by providing operations on Employees such as "GetDepartment", "SetDepartment", etc., which simply access the proper relation (compare with the functional data model [Shipman 81]).

3.2 Generality of Relations

Objects such as paragraphs, images, and documents can be arbitrarily complex structures (the operations defined by their respective types define the legal interfaces for manipulating them). Since there is no restriction on which types constitute the domains of a relation, such objects may be related by the same mechanism

described for simpler objects such as employees, departments, and addresses. Thus, one could create relations to express the correspondence between paragraphs and images, between artists and the images they have created, or between employees and the voice messages sent to them.

4. OTHER IMPORTANT FEATURES

Our model supports multiple implementations of the same type. For example, one might define a HashTable type with several alternative implementations; users of a particular hash table object need not know which kind of hash table it is, only that it responds to the same interface.

Relations may be *derived*—instead of storing the actual relationships, they are computed as needed. Because relations are objects with a specific fixed interface, users of the relation do not need to distinguish between derived and stored relations—at least when making retrievals. The view update problem may be addressed within the framework of the model by taking advantage of multiple implementations to provide special update procedures.

As in other similar models, types may be arranged into a hierarchy (actually a lattice, since a type may have more than one parent). If Employee is a subtype of Person, for example, creation of an employee object will automatically include the Person type. The type lattice facilitates compile-time operation name resolution; for example, a reference to "GetName" on an employee can be bound at compile time to the "GetName" operation defined by type Person.

5. CONCLUSION

The Integrated Data Model offers generality, power, and flexibility. At the same time, we believe that it provides a basis for building friendly and natural interfaces.

Subsets of the model and of an English-like interface have been prototyped. Implementation of the model consisted of writing the low-level layer to support abstract objects, and then simply creating each of the predefined types and writing the operations defined by these types.

We believe that a full implementation can perform competitively; similar designs ([Chan *et al.* 82] [Cattell 83]) have shown promise that optimization techniques (such as clustering of relationship tuples, and replication of immutable objects like integers) can be successfully applied.

Work is in progress on the important areas of transaction management, synchronization, and distribution. We are taking a fairly conventional approach in the design of these features—although, of course, they will be expressed in terms of operations provided by predefined types.

REFERENCES

[Abrial 74] Abrial, J.R. Data Semantics. In [Klimbie 74], 1-59.

[Cattell 83] Cattell, R.G.G. Design and Implementation of a Relationship-Entity-Datum Data Model. Xerox Palo Alto Research Center, Computer Science Laboratory CSL-83-4 (1983).

[Chan *et al.* 82] Chan, A.; Danberg, S.; Fox, S.; Lin, W.; Nori, A.; and Ries, D. Storage and Access Structures to Support a Semantic Data Model. *Proc. 8th Int. Conf. on Very Large Data Bases* (1982 September), 122-130.

[Chen 76] Chen, P.P.-S. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Trans. on Database Syst.* 1:1 (1976 March), 9-36.

[Childs 68] Childs, D.L. Description of a Set-Theoretic Data Structure. *Proc. FJCC. 1968*, 557. North Holland.

[Codd 79] Codd, E.F. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Syst.* 4:4 (1979 December), 397-434.

[Dahl & Bubenko 82] Dahl, R.; and Bubenko, J. IDBD: An Interactive Design Tool For Codasyl-DBTG-Type Data Bases. *Proc. 8th Int. Conf. on Very Large Data Bases* (1982 September), 108-121.

[Klimbie 74] Klimbie, J.W. and Koffeman, K.L. (eds.). *Data Base Management*. North Holland (1974).

[Olle *et al.* 82] Olle, T.W.; Sol, H.G.; and Verrijn-Stuart, A.A. Information Systems Design Methodologies: A Comparative Review. North Holland (1982), 648pp.

[Shipman 81] Shipman, D.W. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Syst.* 6:1 (1981 March), 140-173.

[Wong & Kuo 82] Wong, H.; and Kuo, I. GUIDE: Graphical User Interface for Database Exploration. *Proc. 8th Int. Conf. on Very Large Data Bases* (1982 September), 22-32.