On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives

Won Kim
UniSQL, Inc.
8911 Capital of Texas Highway
Austin, Texas 78759

Today there are unmistakable signs that database technology is moving from pure relational technology and pure object-oriented technology toward a unified object-oriented and relational (object-relational, for short) database technology. The basis of unifying relations and objects is the view that the relational model is merely a special case of an object-oriented model. The objective of this article is to expound on this, and along the way address the differences between the relation-centric and object-centric views that exist on this view.

1. Introduction

Today database technology is in the throes of transition from relational technology to object-relational technology. UniSQL (which originated the technology by coming to market in late 1991), Hewlett Packard, and most recently Illustra (previously called Montage and Miro) currently offer commercial object-relational database systems. Vendors of so-called "pure" object-oriented database systems, which in effect is a misnomer for persistent storage managers for object-oriented programming languages, are attempting to add SQL-based ad hoc query facilities and traditional database facilities that mainstream database users have become accustomed to. At the same time, vendors of relational database systems are attempting to extend their products with object-oriented modeling facilities and/or support for object-oriented programming languages.

The ANSI X3H2 (SQL-3) standards committee has been working to formulate object-oriented extensions to the SQL-92 relational database standard, and the ODMG (Object Data Management Group) committee has proposed a snapshot proposal for interfacing object-oriented programs to a database. However, the scope of the work of the ANSI X3H2 committee does not include complete unification of relations and objects; in particular, it does not include consideration of direct support for object-oriented programming languages (i.e., support for seamless storage and retrieval of C++ or Smalltalk objects). Further, a stable snapshot of an object SQL is not expected for another year. Further, the primary focus of the ODMG committee is on "seamless" binding between C++ or Smalltalk programs and a database, rather than on an SQL-compatible ad hoc query language.

I will address a few basic questions here to set the tone of this article. The first question is just what is meant by "marrying (or unifying) objects and relations", or "adding objects to relations". In my opinion, an object-oriented data model subsumes the relational data model; in other words, an object-oriented data model contains all the modeling concepts found in the relational model, and has additional modeling concepts. Therefore, "marrying (or unifying) objects and relations", or "adding objects to relations" is simply extending the relational model to an object-oriented model [KIM93].

The second question is why would one marry objects and relations? In my opinion, there are two reasons. One is to eliminate a few major deficiencies in the relational model with respect to database design and database performance. These include the limited data types supported in relational database systems (I note that this is a problem of the implementation of the relational model, rather than a problem of the relational model itself), the join based retrieval of a collection of related data (even when the need to retrieve the collection of data is known at database design time), and duplication of records to work around the first-normal form restriction. Another reason is to bring the productivity benefits in object-oriented programming to database design and application...
development. These boil down to inheritance (or reusability) of a database schema and programs stored in the database, and support of application programs written in object-oriented programming languages (i.e., C++ and Smalltalk).

The third question is what modeling facilities must be added to the relational model for it to become an object-oriented model. I will address this question in the remainder of this article. Chris Date (and Hugh Darwen) has recently observed that there are two perspectives to adding objects to relations: what they call the “relational bigot’s perspective” and the “object bigot’s perspective” [Database Newsletter 941]. (Since some readers may find these terms offensive, I will use the terms “relation-centric” and “object-centric” instead for the remainder of this article.) They conclude that the relation-centric view (their view) is right, and the object-centric view (most current implementations of object-oriented databases and object-relational databases) is wrong.

In my view, the relation-centric view is simply an incomplete object-centric view; in other words, the relation-centric view has important additions to the original relational model, but is still missing some important modeling facilities before it can satisfy the two reasons for marrying objects and relations. The relation-centric view does not eliminate some of the key deficiencies in the relational model that fueled a decade-long database research in the 1980s, and does not allow support of object-oriented programming.

In this article, I will build the answers to the third question above in two stages. In Section 2, I will briefly summarize the relation-centric view on extending relations with objects, and discuss some of the requirements that it cannot satisfy that gave rise to object-oriented database systems in the first place. In Section 3, I will show how the relation-centric view may be extended to the object-centric view, and discuss how the object-centric view addresses deficiencies of the relation-centric view.

2. Relation-Centric Perspective

The relation-centric perspective is as follows.

1. The domain of a column of a relation may be not only the alphanumeric data types (e.g., integer, float, string) that relational database systems today allow, but also any arbitrary user-defined data types (e.g., point, polygon, document, image).

A user-defined data type consists of “all possible values for the type”, and “functions that perform operations on each value of the type”.

A user-defined data type, not the relation, is equivalent to a class in object-oriented programming or object-oriented database systems.

2. User-defined data types form an inheritance hierarchy to provide inheritance paths for only the functions (methods) associated with them. Specifically, the structure of a user-defined data type that provides a template for the values of the data type is not inherited.

3. All queries (i.e., projection, selection, and join) and data manipulation statements (i.e., insert, update, and delete) are directed to relations, and never to user-defined data types.

I note that this extended relational model is not really a full object-oriented data model. Some of the key considerations that are missing are the following:

1. In an object-oriented model, a class includes instances. The relation-centric “object model” does not really include the notion of a class. Although the model equates an arbitrary user-defined data type to a class, there is a problem. A data type is simply a domain that includes “all possible values” for the type. A class, on the other hand, only has instances that are of interest to users (i.e., instances that users have created and inserted into the extent of the class).

2. In an object-oriented model, each instance of a class has an associated unique object identifier. The relation-centric “object model” does not include any notion of object identity, except for the unique handle that is the value itself of a data type. A tuple of a relation does not have a unique identifier that can be made visible to the user as a handle for fetching a specific tuple using the identifier. (I note that relational database systems do assign tuple identifiers to tuples in relations. However, the user interface does not include a mechanism for retrieving individual tuples using the tuple identifiers as the key.)

3. In an object-oriented model, a subclass (or subtype) inherits not only methods from superclasses (supertypes), but also attributes (i.e., structure). The relation-centric model expressly excludes the structure of a data type from inheritance. It is not said why this restriction is imposed.

4. In an object-oriented model, a class has methods associated with it. Although the above description of the relation-centric object model seems to suggest
that a data type has functions (i.e., methods) associated with it, proponents of the relation-centric view take a confusing stance that methods should not be associated with a class (data type), but should be free-floating (as with stored procedures in relational database systems). I am assuming that the description above is the intended model.

5. In an object-oriented model, if a data type is specified as the domain of an attribute, the domain is often actually meant to be not only the specified data type, but also all its subtypes. The relation-centric model is not capable of supporting this, since the structure of a data type cannot be inherited.

Although the relation-centric model is certainly more powerful than the conventional relational model, it still falls well short of satisfying the following important requirements that has led to object-oriented database systems (not just object-relational database systems).

1. The extended relational model does not provide a natural means of modeling an object configuration (often called a complex object, composite object, or a nested object). An object configuration is a cluster of related objects in which the root object recursively refers to other objects. An object configuration may be a physical configuration or a logical configuration. An example of a physical configuration is a parts assembly of a physical device (e.g., an automobile engine), in which an object is a constituent of only one other object (i.e., a particular cylinder belongs to only one particular engine). In a logical configuration (e.g., a compound document or a design of a building), an object may be a constituent of more than one other object (i.e., a particular drawing may be included in two or more different documents; a particular design of a room may be included in two or more building designs).

Unless this particular shortcoming of the relational data model is addressed, relational database technology will continue to leave unfulfilled the requirements of a range of application domains (e.g., computer-aided design, engineering, analysis) that require rapid traversal of an object configuration.

Database application developers have long recognized the awkwardness of representing an object configuration in terms of multiple relations logically connected via the primary key/foreign key constraint. Further, they have long been baffled by the intolerable performance in fetching an object configuration (in its entirety or in part) thus awkwardly represented. The only way to fetch an object configuration in a relational database is by performing joins across the relations that contain tuples that represent all constituent objects.

A natural way to model an object configuration is by having an object recursively refer to (i.e., point to) its constituent objects. (In fact, the primary key/foreign key constraint is merely a long-winded way of expressing this simple referencing from one tuple to related tuples.) A natural (and efficient) way to fetch an object configuration is to first find the tuple that represents the root object of the configuration, and then recursively fetch all tuples that represent constituent objects by chasing down the pointers from the root object.

A database system may easily implement a reference from one object to its constituent object(s) by storing in a tuple representing an object pointers to tuples that represent the object's constituent objects. The pointers need not be physical pointers that the relational model was designed to eliminate; the pointers may be logical pointers, and further, they are encapsulated, that is, the users are not allowed to directly create, read, update, and delete the pointers. A database system may automatically generate unique logical pointers (called object identifiers), and never allow the users to read, update, or delete the pointers. In fact, even today's relational database systems actually use tuple identifiers stored in relations to get to related tuples quickly in managing system catalogs (e.g., in looking up tuples that contain information about columns in a relation, once a tuple describing a relation has been found). So, the use of object identifiers within an object as a means of enhancing performance in getting from one object to related objects is not such an ungodly thing.

The lack of object identity in turn means no pointer swizzling. In other words, when a tuple of a relation R and a tuple of a relation T are fetched, a relational database system does not link the two tuples with a memory pointer. Object-oriented database systems perform pointer swizzling by converting an object identifier of object B that is stored in an attribute of object A to a memory pointer to object B, when both object A and object B have been fetched into memory. Since the extended relational model does not admit object identity, there is no basis for pointer swizzling.
Proponents of the relational model have long maintained that allowing pointer chasing in database applications is a bad idea and is a step back into the days of hierarchical and CODASYL database systems. However, many applications demand the level of performance that can be attained only through pointer chasing through objects (data) in memory, and/or at least the performance gain that is possible through a direct reference (rather than through joins) from one record to another record in the database. This is one of the most important reasons that gave rise to the so-called "pure" object-oriented database systems (i.e., storage managers for C++ or Smalltalk programs), all of which perform pointer swizzling. Of the object-relational database systems, UniSQL is the only product that supports pointer swizzling (over and above ad hoc object SQL queries).

2. The extended relational model also does not provide a basis for efficiently supporting object-oriented programming languages. The reason for this is, again, that the model does not include object identity. There is no object identity for each tuple of a relation; and there is no object identity associated with each value of a user-defined data type (other than the value itself). This means that in order to fetch a particular object created and stored by an object-oriented program, the object request must be mapped to a SQL query that can uniquely identify a tuple of a relation (rather than just submitting a handle to an object identifier of the desired object).

Unless a database system provides a mechanism for the efficient implementation of seamless support for object-oriented programming, the mainstream database technology will continue to leave out an important emerging application domain, namely, applications that are written in C++ or Smalltalk.

3. Although the extended relational model allows a set of values in a row/column entry in a relation, it insists that the set be treated as a single atomic value in the interest of preserving the first-normal form restriction of the original relational model. In other words, a database system supporting the extended relational model should not provide any means of accessing and manipulating individual elements of a set. (I note that the treatment of a set as the value of an attribute is not one of the core object-oriented concepts, and as such this particular point is rather tangential to the main subject of this article. I include this discussion to illustrate my view that it is unreasonable to doggedly hold onto some theoretical restrictions in the face of serious practical problems that result from them. I will further illustrate this view in the next section.)

In closing this section, I would like to observe that a database system is merely a tool that application developers use to develop and run their applications. If it becomes clear that database systems based on particular assumptions (i.e., data model, theory, and engineering) are inadequate in supporting the needs of application developers, then the assumptions (and therefore the database systems) must be changed. The relational model is a simple basis of building one type of database system, and has served its purpose well. But such highly useful and natural means of modeling and managing data as a repeating group (i.e., set) and record retrieval via pointer chasing, both of which were supported in hierarchical and CODASYL database systems, were thrown out in the interest of the first-normal form theory and data independence. But as in most things in life, if one gains something, one tends to lose something else. Application developers have had to work around the loss of repeating group and pointer chasing. If application developers want these, database system developers should simply provide them. After all, a data model is not a religion: if the relational calculus cannot explain everything, then it is time to extend the theory or find another theory that is more encompassing, rather than telling users that they must not demand things that may disrupt the simple elegance of an old mathematical theory.

3. Object-Centric Perspective

The premise of the relation-centric perspective is that the object-centric perspective equates a relation with a class, and that object identity and pointer chasing must at all cost be excluded from any data model considerations. The fact is that no object database systems have ever equated a relation with a class; in this sense, the relation-centric perspective is a criticism against a non-existent perspective.

The object-centric perspective is in general an attempt to go beyond the relation-centric perspective in order to satisfy requirements that the relation-centric perspective can not. The following is a summary of an object-centric perspective (in particular, UniSQL).

1. The domain of a column of a relation may be not only the alphanumeric data types (e.g., integer,
A user-defined data type consists of "all possible values for the data type", and "functions that perform operations on each value of the type".

A user-defined data type is equivalent to a class in object-oriented programming or object-oriented database systems.

2. User-defined data types form an inheritance hierarchy to provide inheritance paths for not only the functions associated with them, but also the structure of the values of the type.

3. Queries (i.e., projection, selection, and join) and data manipulation statements (i.e., insert, update, and delete) are directed to relations.

4. A relation is augmented to a class by including a system-defined column for tuple identity (i.e., object identifier) and, optionally, functions (i.e., methods) that operate on each tuple of the relation. A relation is then the extent of this type of class. Queries (i.e., projection, selection, and join) and data manipulation statements (i.e., insert, update, and delete) may be directed to the relations that are the extents of the classes.

A class is regarded as a data type that belongs in the type hierarchy.

5. A class augmented from a relation may contain an attribute that "references" another class augmented from a relation. The reference is to the extent of the class.

I note that the above summary of an object-centric model fully subsumes the extended relational model of the relation-centric view. The first 3 points above are identical to those of the extended relational model (except that in point 2, inheritance includes not only the functions associated with a type, but also the structure of the values of the type). Points 4 and 5 are additions to the extended relational model. Further, the extensions 4 and 5 above, contrary to the implied concern in the relation-centric perspective that they may bring about chaos, are well-founded on research and engineering work that has gone on for over 10 years around the world.

Point 4 is a simple way to bring relational query facilities to the extents of user-defined classes. Proponents of the relation-centric view are troubled by the thought of issuing a query against a class, primarily because, they say, the result of a query against a class is not a class. A class has methods, but the result of a SELECT * query against a class will lose the methods. In my view, this concern is not well-founded. A query against a class is really against the extent of a class, just as a query against a relation in a relational database is also against the extent of a relation. Indeed, all types of relational queries apply to the extents of the classes. If the query result needs to be saved as the extent of a new class, the user may add to the new class any useful methods defined in the original class. The database system generates object identifiers for all tuples of the extent of the new class.

A relation in a relational database has two aspects: intensional and extensional. The intensional aspect of a relation is the relation definition, which includes all properties of the relation, such as integrity constraints (UNIQUE, primary key, foreign key), and domain, length, and integrity (NULL allowed) specification for each column. The extensional aspect of a relation is simply the set of tuples that store user-created values. A query in a relational database is evaluated against the extent of a relation, and the result is a set of tuples that form an extent of a new relation. But the result of a query does not necessarily have the full intensional specification! For example, the result of a join of two relations is a relation, but the resulting relation does not necessarily have all the integrity constraints automatically specified.

I need to make a few remarks about encapsulation at this point. There are at least two aspects to encapsulation in the context of a database: attributes in a class, and methods in a class. If one carries a 'rigorous' view of encapsulation to attributes in a class, one needs to write two methods for each attribute in order to make them public (i.e., accessible to users); namely, a method for reading the value stored in the attribute, and a method for updating the value. For a database, this is an unproductive exercise. I believe that one should take the simple view that the attributes in a database class are all public attributes.

Another aspect of encapsulation is methods in a class. The relation-centric perspective seems to be that methods should not be associated with a single class, because sometimes one needs to write methods that operate on values of more than one data types. I believe there is a very simple solution to this problem. Just as in object-oriented programming languages, one should have methods associated with a single class. However, one should
also have 'free-floating' methods that belong to the entire database, rather than any particular class.

Point 5 is a simple way to allow the natural nesting of objects, and provides a basis for pointer swizzling, to address the data modeling and performance requirements that relational database systems have proven to be very unsatisfactory. Point 5 in essence represents the sum total of the theoretical research and engineering work that has gone on for over 10 years on overcoming the serious limitations of the relational model. It appears that proponents of the relation-centric view have three areas of concern with respect to point 5: querying nested objects, breaking of encapsulation, and pointer swizzling.

An ad hoc query may be easily formulated against the "path" of referenced classes. A "path" is formed starting with a class, followed by the attribute that references another class. A path is nothing more than a generalized form of an attribute. For example, if a class Automobile has an attribute Manufacturer that "references" a class Company, a path that results is of the form "Automobile.Manufacturer". A path name may be used in place of an attribute name in a query predicate. For example, Automobile.Manufacturer.Name = 'General Motors' is a query predicate in which a path name "Automobile.Manufacturer.Name" is used in place of a single attribute name. The notion of a path and the use of a path in a query expression in an extended relational database or an object-oriented database has been formalized over the past 10 years. Perhaps the earliest research that dealt with a path query is in the design of an extended relational database language, named GEM, at Bell Laboratories [Zaniolo 83]. A recent research paper on the subject is [Kifer, Kim, and Sagiv 92].

One common complaint about a path query is that encapsulation may be violated in evaluating a query. I take the view that there is no harm done as long as it is the database system, rather than a random user, that violates encapsulation. There is no good reason that we should prevent a database system from seeing some user data. A database system sees many other things that ordinary users cannot see; such as the authorization catalog, lock table, log file, etc. A database system is a privileged user, and a robust database system does not do any damage to itself or the database.

An obvious way to implement the "reference" attribute is to store the identifiers of the tuples in the referenced class as the values of the attribute. For example, if a class Automobile has an attribute Manufacturer that "references" a class Company, the value stored in an instance of the class is the identifier of a particular instance of the class Company. (I note that, for simplicity, in the UniSQL database system, the notion of "reference to a class" is implicit; that is, no special syntax is provided to distinguish between a class and a reference to a class, when the class is specified as the domain of an attribute.)

The identifier stored in a reference attribute can be used (by the database system) to directly fetch the referenced instance. Further, the identifiers of instances may be mapped (by the database system) to memory pointers, when the instances are fetched into memory. This is the pointer swizzling technique that commercial "pure" object-oriented database systems and the UniSQL object–relational database system use. I emphasize that pointer traversal should not be the only means of accessing data in a database system. A database system for managing a very large database must in general provide ad hoc query facilities. However, I also recognize that some important applications absolutely require pointer traversal. Further, it is entirely possible, as the UniSQL object–relational database system has proven, ad hoc query facilities and pointer traversals can be combined in a single database system, and users can use either or both of the data access mechanisms. The user may first issue an ad hoc query to retrieve a small number of objects of interest, and then, for each object thus retrieved, access constituent objects via object identifiers (or memory pointers, if the objects are already in memory).

The pointer swizzling technique has been in use for over 10 years. It was first formalized in building the LOOM (large object–oriented memory) language at Xerox Palo Alto Research Center [Kaelter 81], and was further developed in implementing the persistent Algol language [Atkinson 83]. A further refinement of the technique was done in the implementation of the ORION object–oriented database system at MCC [Kim et al. 88]. A recent research paper has analyzed a few different pointer swizzling techniques [White and Dewitt 92].

Before closing this section, I note that the use of a class in a type hierarchy is somewhat troublesome, even though even the relation-centric perspective allows it. The domain is all possible values in a data type. But a class consists of only user-created instances. So, from this perspective,
a class is not a data type. However, there is much to be gained and little to sacrifice by viewing a class as a type and providing a single type hierarchy for defining inheritance.

Concluding Remarks

Today, database technology is perceptibly moving toward a unified object-oriented and relational database technology as a means of overcoming some key shortcomings in data modeling and performance capabilities in existing relational database systems, and also as a means of leveraging some key benefits that object-oriented technology offers. A key issue is how relations and objects can be united. The most alluring view is to extend the relational model with key object-oriented concepts, namely arbitrary data types, inheritance, encapsulation and nesting of objects.

Recently, Chris Date (and Hugh Darwen) proposed a simple scheme to extend the relational model with arbitrary data types and inheritance. This proposal, which Date calls the relational bigot's perspective, is a significant improvement over the conventional relational model. However, it falls well short of overcoming the key shortcomings in relational database systems that have given rise to the so-called "pure" object-oriented database systems, namely seamless integration of object-oriented programming and database, and support for natural modeling and efficient retrieval of complex configuration of objects. In this article, I presented the rationale for uniting objects and relations, and the major elements that are needed to make it happen properly. In so doing, I tried to point out that the relation-centric perspective is merely a way-station to getting to the full object-centric perspective on uniting objects and relations. I also emphasized that the extensions to the relation-centric perspective are well-founded on the research and engineering work done for over a decade. The UniSQL database system has largely implemented the object-centric perspective that I outlined in this article.

References