Object–Oriented Database Systems: Promises, Reality, and Future

Won Kim

UniSQL, Inc.
9390 Research Blvd.
Austin, Texas 78759

Abstract

During the past decade, object–oriented technology has found its way into programming languages, user interfaces, databases, operating systems, expert systems, etc. Products labeled as object–oriented database systems have been in the market for several years, and vendors of relational database systems are now declaring that they will extend their products with object–oriented capabilities. A few vendors are now offering database systems that combine relational and object–oriented capabilities in one database system. Despite these activities, there are still many myths and much confusion about object–oriented database systems, relational systems extended with object–oriented capabilities, and even the necessities of such systems among users, trade journals, and even vendors. The objective of this paper is to review the promises of object–oriented database systems, examine the reality, and how their promises may be fulfilled through unification with the relational technology.

1. Definitions

Object–oriented technologies in use today include object–oriented programming languages (e.g., C++ and Smalltalk), object–oriented database systems, object–oriented user interfaces (e.g., Macintosh and Microsoft window systems, Frame and Interleaf desktop publishing systems), etc. An object–oriented technology is a technology that makes available to the users facilities that are based on "object–oriented concepts". To define "object–oriented concepts", we must first understand what an "object" is.

The term "object" means a combination of "data" and "program" that represent some real–world entity. For example, consider an employee named Tom; Tom is 25 years old, and his salary is $25,000. Then Tom may be represented in a computer program as an object. The "data" part of this object would be (name: Tom, age: 25, salary: $25,000). The "program" part of the object may be a collection of programs (hire, retrieve the data, change age, change salary, fire). The data part consists of data of any type. For the "Tom" object, string is used for the name, integer for age, and monetary for salary; but in general, even any user–defined type, such as Employee, may be used. In the "Tom" object, the name, age, and salary are called attributes of the object.

Often, an object is said to "encapsulate" data and program. This means that the users cannot see the inside of the object "capsule", but can use the object by calling the program part of the object. This is not much different from procedure calls in conventional programming; the users call a procedure by supplying values for input parameters and receive results in output parameters.

The term "object–oriented" roughly means a combination of object encapsulation and inheritance. The term "inheritance" is sometimes called "reuse". Inheritance means roughly that a new object may be created by extending an existing object. Now let us understand the term "inheritance" more precisely. An object has a data part and a program part. All objects that have the same attributes for the data part and same program part are collectively called a class (or type). The classes are arranged such that some class may inherit the attributes and program part from some other classes.

Tom, Dick, and Harry are each an Employee object. The data part of each of these objects consists of the attributes Name, Age, and Salary. Each of these Employee objects has the same program part (hire, retrieve the data, change age, change salary, fire). Each program in the program part is called a "method". The term "class" refers to the collection of all objects that have the same attributes and methods. In our example, the Tom, Dick, and Harry objects belong to the class Employee, since they all have the same attributes and methods. This class may be used as the type of an attribute of any object. At this time, there is only one class in the system, namely, the class Employee; and three objects that belong to the class, namely, Tom, Dick, and Harry objects.

Now suppose that a user wishes to create two sales employees, John and Paul. But sales employees have an additional attribute, namely, Commission. The sales employees cannot belong to the class Employee. However, the user can create a new class, Sales_Employee, such that all attributes and methods associated with the class Employee may be reused and the attribute Commission may be added to Sales_Employee. The user does this by declaring the class Sales_Employee to be a "subclass" of the class Employee. The user can now proceed to create the two sales employees as objects belonging to the class SalesEmployee. The users can

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

create new classes as subclasses of existing classes. In general, a class may inherit from one or more existing classes, and the inheritance structure of classes becomes a directed acyclic graph (DAG); but for simplicity, the inheritance structure is called an "inheritance hierarchy" or "class hierarchy".

The power of object-oriented concepts is delivered when encapsulation and inheritance work together.

- Since inheritance makes it possible for different classes to share the same set of attributes and methods, the same program can be run against objects that belong to different classes. This is the basis of the object-oriented user interface that desktop publishing systems and windows management systems provide today. The same set of programs (e.g., open, close, drop, create, move, etc.) apply to different types of data (image, text file, audio, directory, etc.).

- If the users define many classes, and each class has many attributes and methods, the benefit of sharing not only the attributes but also the programs can be dramatic. The attributes and programs need not be defined and written from scratch. New classes can be created by adding attributes and methods to existing classes, rather than by modifying the attributes and methods of existing classes, thereby reducing the opportunity to introduce new errors to existing classes.

2. Promises of OODBs

An object-oriented programming language (OOPL) provides facilities to create classes for organizing objects, to create objects, to structure an inheritance hierarchy to organize classes so that subclasses may inherit attributes and methods from superclasses, and to call methods to access specific objects. Similarly, an object-oriented database system (OODB) should provide facilities to create classes for organizing objects, to create objects, to structure an inheritance hierarchy to organize classes so that subclasses may inherit attributes and methods from superclasses, and to call methods to access specific objects. Beyond these, an OODB, because it is a database system, must provide standard database facilities found in today's relational database systems (RDBs), including nonprocedural query facility for retrieving objects, automatic query optimization and processing, dynamic schema changes (changing the class definitions and inheritance structure), automatic management of access methods (e.g., B+-tree index, extensible hashing, sorting, etc.) to improve query processing performance, automatic transaction management, concurrency control, recovery from system crashes, security and authorization. Programming languages, including OOPLs, are designed with one user and a relatively small database in mind. Database systems are designed with many users and very large databases in mind; hence performance, security and authorization, concurrency control, dynamic schema changes become important issues. Further, database systems are used to maintain critical data accurately; hence, transaction management, concurrency control, and recovery are important facilities.

Insofar as a database system is a system software whose functions are called from application programs written in some host programming languages, we may distinguish two different approaches to designing an OODB. One is to store and manage objects created by programs written in an OOPL. Some of the current OODBs are designed to store and manage objects generated in C++ or Smalltalk programs. Of course, an RDB can be used to store and manage such objects. However, RDBs do not understand objects, in particular, methods and inheritance. Therefore, what may be called an "object manager" or an "object-oriented layer" software needs to be written to manage methods and inheritance, and to translate objects to tuples (rows) of a relation (table). But, the object manager and RDB combined are in effect an OODB (with poor performance of course)!

Another approach is to make object-oriented facilities available to users of non-OOPLs. The users may create classes, objects, inheritance hierarchy, etc.; and the database system will store and manage those objects and classes. This approach in effect turns non-OOPLs (e.g., C, FORTRAN, COBOL, etc.) into object-oriented languages. In fact, C++ has turned C into an OOPL, and CLOS has added object-oriented programming facilities to CommonLISP. An OODB designed using this approach can of course be used to store and manage objects created by programs written in an OOPL. Although a translation layer would need to be written to map the OOPL objects to objects of the database system, the layer should be much less complicated than the object manager layer that an RDB would require.

In view of the fact that C++, despite its growing popularity, is not the only programming language that database application programmers are using or will ever use, and there is a significant gulf between a programming language and a database system, the second approach is a more practical basis of a database system that will deliver the power of object-oriented concepts to database application programmers. Regardless of the approach, OODBs, if done right, can bring about a quantum jump in the productivity of database application programmers, and even in the performance of the application programs.

One source of the technological quantum jump is the reuse of a database design and program that object-oriented concepts make possible for the first time in the evolving history of database technologies. Object-oriented concepts are fundamentally designed to reduce the difficulty of developing and evolving complex software systems or designs. Encapsulation and inheritance allow attributes (i.e., database design) and programs to be reused as the basis for building complex databases and programs. This is precisely the goal that has driven the data management technology from file systems to relational database systems during the past three decades. An OODB has the potential to satisfy the objective of reducing the difficulty of designing and evolving very large and complex databases.

Another source of the technological jump is the powerful data type facilities implicit in the object-oriented concepts of encapsulation and inheritance. The data type facilities in fact are the keys to eliminating three of the important deficiencies of RDBs. These are summarized below. I will discuss these points in greater detail later.

- RDBs force the users to represent hierarchical data (or complex nested data, or compound data) such as bill of materials in terms of tuples in multiple relations. This is awkward to start with. Further, to retrieve data thus spread out in multiple relations, RDBs must resort to joins, a generally expensive operation. The data type of an attribute of an object in OOPLs may be a primitive type or an arbitrary user-defined type (class). The fact that an object may have an attribute whose value may be another object naturally leads to nested

object representation, which in turn allows hierarchical data to be naturally (i.e., hierarchically) represented.

– RDBs offer a set of primitive, built–in data types for use as domains of columns of relations, but do not offer any means of adding user–defined data types. The built–in data types are basically all numbers and short symbols. RDBs are not designed to allow new data types to be added, and therefore often require a major surgery to the system architecture and code to add any new data type. Adding a new data type to a database system means allowing its use as the data type of an attribute, that is, storage of data of that type, querying and updating of such data. Object encapsulation in OOPLs does not impose any restriction on the types of data that the data part of an object may hold, that is, the types of data may be primitive types or user–defined types. Further, new data types may be created as new classes, possibly even as subclasses of existing classes, inheriting their attributes and methods.

– Object encapsulation is the basis for the storage and management of programs as well as data in the database. RDBs now support "stored procedures", that is, they allow programs to be written in some procedural language and stored in the database for later loading and execution. However, the stored procedures in RDBs are not encapsulated with data; that is, they are not associated with any relation or any tuple of a relation. Further, since RDBs do not have the inheritance mechanism, the stored procedures cannot automatically be reused.

3. Reality of OODBs

There are a number of commercial OODBs. These include GemStone from Servio Corporation, ONTOS from ONTOS, ObjectStore from Object Design, Inc., Objectivity/DB from Objectivity, Inc., Versant from Versant Object Technology, Inc., Matisse from Intellitic International (France), Itasca (commercial version of MCC's ORION prototype) from Itasca Systems, Inc., O2 from O2 Technology (France). These products all support an object–oriented data model. Specifically, they allow the user to create a new class with attributes and methods, have the class inherit attributes and methods from superclasses, create instances of the class each with a unique object identifier, retrieve the instances either individually or collectively, and load and run methods.

These products have been in the market since as early as 1987. However, most of them have been in evaluation, and preliminary prototype application development; that is, they have not been seriously used for many mission–critical applications. Further, a fairly large number of copies of the products have been given away for free trial, artificially boosting the total count of product installations. The worldwide market size for all of the current OODBs combined is estimated to be $20–30 million — a tiny fraction of the $3 billion worldwide market size for all database products. To be sure, the past several years have been a gestation period for object–oriented technology in general, and object–oriented database technology in particular. Further, the technical market and OOPL market which the current OODBs have targeted are new markets that have not been previously relied on database systems. However, the lack of maturity of the initial (and to a good extent, the current) OODB offerings has also contributed significantly to their slow acceptance in mission–critical applications.

3.1 Limitations

limitations as persistent storage systems

One key objective, and therefore, selling point, of most of the current OODBs is the support of a unified programming and database language, that is, one language (e.g., C++ or Smalltalk) in which to do both general–purpose programming and database management. This objective was the result of the current situation where application programs are written in a combination of a general–purpose programming language (mostly, COBOL, FORTRAN, PL/I, or C), and database management functions are embedded within the application programs in a database language (e.g., the SQL relational database language). A general–purpose programming language and a database language are very different in syntax and data model (data structures and data types), and the necessity of having to learn and use two very different languages to write database application programs has been frequently regarded as a major nuisance. Since C++ and Smalltalk already include facilities for defining classes and a class hierarchy (i.e., for data definition), in effect, these languages are a good basis for a unified programming and database language. The first step that most of the vendors of the early OODBs took was to make the classes and instances of the classes persistent, that is, to store them on secondary storage and make them accessible even after the programs which defined and created them have terminated.

Current OODBs that are designed to support OOPLs place various restrictions on the definition and use of objects. In particular, most systems treat persistent data differently from nonpersistent data (e.g., they make it illegal for a persistent object to contain the OID of a nonpersistent object), and therefore require the users to explicitly declare whether an object is persistent or not. Further, they cannot make certain types of data persistent, and therefore prohibit their use.

limitations as database systems

The second, much more severe, source of immaturity of most of the current OODBs products is the lack of basic features that users of database systems have become accustomed to and therefore have come to expect. The features include a full nonprocedural query language (along with automatic query optimization and processing), views, authorization, dynamic schema changes, and parameterized performance tuning. Besides these basic features, RDBs offer support for triggers, meta data management, constraints such as UNIQUE and NULL — features that most OODBs do not support.

– Most of the OODBs suffer from the lack of query facilities; and those few systems that do provide significant query facilities, the query language is not ANSI SQL–compatible. Typically, the query facilities do not include nested subqueries, set queries (union, intersection, difference), aggregation functions and group by, and even joins of multiple classes, etc. — facilities fully supported in RDBs. In other words, these products allow the users to create a flexible database schema and populate the database with many instances, but they do not provide a powerful enough means of retrieving objects from the database.

– RDBs support views as dynamic windows into the stored database. The view definition includes a query statement to specify the data that will be fetched to constitute the view. A

view is used as a unit of authorization. No OODB today supports views.

- RDBs support authorization — that is, they allow the users to grant and revoke privileges to read or change the tuples in the tables or views they created to other users, or to change the definition of the relations they created to other users. Most OODBs do not support authorization.

- RDBs allow the users to dynamically change the database schema using the ALTER command; a new column may be added to a relation, a relation may be dropped, and a column can sometimes be dropped from a relation. However, most of the current OODBs do not allow dynamic changes to the database schema, such as adding a new attribute or method to a class, adding a new superclass to a class, dropping a superclass from a class, adding a new class, and dropping a class.

- RDBs automatically set and release locks in processing query and update statements the users issue. However, some of the current OODBs require the users to explicitly set and release locks.

- RDBs allow the installation to tune system performance by providing a large number of parameters that can be set by the system administrator. The parameters include the number of memory buffers, the amount of free space reserved per data page for future insertions of data, and so forth. Most of the OODBs offer a limited capability for parameterized performance tuning.

Because of the deficiencies outlined above, most of these products will require major enhancements. It is safe to assume that the vendors of these products will make the required changes to their current software, rather than rewriting the products from scratch. The extent of the changes that will be required to bring these products to full–fledged database systems that can at least match the level of database functionality expected of today's database systems is so great that it is not expected that the enhanced products will attain the robustness and performance required for mission–critical applications within the next three or four years.

Upgrading most of the current OODBs to true database systems poses not only major technical difficulties as outlined above, but also a serious philosophical difficulty. As we have seen already, most of the current OODBs are closer to being merely persistent storage systems for some OOPL than database systems. The term OODB was not deliberately designed to be misleading and confusing, since the OODBs were designed to manage a database of objects generated by programs written in OOPLs. However, the database users have been trained during the past two decades to think of a database system as a software that allows a large database to be queried to retrieve a small portion of it, that does not require any hint from the user about how to process any given query, that allows a large number of users to simultaneously read and update the same database, that automatically enforces database integrity in the presence of multiple concurrent users and system failures, that allows the creator of a portion of a database to grant and revoke access privileges to his data to other users, that allows the installation to tune the performance of a database system by adjusting various system parameters, and so forth. For this reason, the term OODB has become a misnomer for most of the current OODBs.

Most of the current OODBs have essentially extended the OOPLs with a run–time library of database functions. These functions must be called from the application programs, with appropriate specifications of the input and output parameters. The syntax of the calling functions is made consistent with the application programming language. As the current OODBs are upgraded to true database systems, a major extension to the current library of database functions will be necessitated to support query facilities. Today's programming languages, including object–oriented languages, simply are not designed with database queries in mind. A database query may return an indeterminate number of records or objects that satisfy user–specified search conditions. Therefore, the application program must be designed to step through the entire set of records or objects that are turned until there is no more left. This is what led to the introduction of the cursor mechanism in database systems. The result of a database query must therefore be assigned to some data structure and accompanying algorithm that can store and step through an indefinite number of objects. Further, there will arise the need to provide facilities to specify nested subqueries, postprocessing on the result of a query (corresponding to GROUP BY, aggregation functions, correlation queries, etc.), and set queries (union, intersection, difference). In the name of a unified programming and database language, presumably, all these facilities will be made available to the programmers in a syntax that is consistent with the programming languages. In other words, the unified language approach does not eliminate the need for any of the database facilities; rather, it merely makes the facilities available to the users in a different syntax. Further, the syntax, to be consistent with the host programming languages, is at a low, procedural level. A procedural syntax is always more difficult for non–technical users to learn and use. Therefore, it is not clear if ultimately the unified language approach offers any advantages over that of embedding a database language in host programming languages.

3.2 Myths

There are many myths about OODBs. Many of these myths are totally without merit, and are the result of the unfortunate label "database system" that has been attached to most of the current OODBs that are not full–fledged database systems comparable to the current RDBs. Some of the myths are the result of the evolving nature of the technology. Yet others represent concerns from purists that in my view are not practically useful.

OODBs are 10 to 100 times faster than RDBs

Vendors of OODBs often make the claim that OODBs are between 10 to 100 times faster than RDBs, and back up the claim with performance numbers. This claim can be misleading unless it is carefully qualified. OODBs have two sources of performance gain over RDBs. In an OODB the value of an attribute of an object X whose domain is another object Y is the object identifier (OID) of the object Y. Therefore, if an application has already retrieved object X, and now would like to retrieve object Y, the database system may retrieve object Y by looking up its OID. Figure 1.a illustrates two instances of the class Person, and two instances of the class Company, such that the class Company is the domain of the attribute Worksfor in the class Person. The value stored in the Worksfor attribute is the OID of an object of the class

Company. If the OID is a physical address of an object, the object may be directly fetched from the database; if the OID is a logical address, the object may be fetched by looking up a hash table entry (assuming that the system maintains a hash table that maps an OID to its physical address).

The current RDBs allow only a primitive data type as the domain of an attribute of a relation. As such, the value of an attribute of a tuple can only be primitive data (such as a number or string), and never be another tuple. If a tuple Y of a relation R2 is logically the value of an attribute A of a tuple X of a relation R1, the actual value stored in attribute A of tuple X is a value of attribute B of tuple Y of relation R2. If an application has retrieved tuple X, and would now like to retrieve tuple Y, the system must in effect execute a query that scans the relation R2 using the value of attribute A of tuple X. Figure 1.b is an equivalent representation in an RDB of the object-oriented database in Figure 1.a. The domain of the attribute Worksfor in the relation Person is the primitive data type String. If an application has retrieved the Person tuple for "John", and would like to retrieve the Company tuple for "UniSQL", it needs to issue a query that will scan the Company relation. Imagine that the Company relation has thousands or tens of thousands of tuples. If no index is maintained on attribute B (Name) of relation R2 (Company), the entire relation R2 must be sequentially searched to find tuple Y (for "UniSQL"). If an index is maintained on attribute B, tuple Y may be retrieved about as fast as in OODBs that resort to a hash table lookup, but less efficiently than in OODBs that implement OIDs as physical addresses (and therefore do not require any hash table lookup).

A second source of performance gain in OODBs over RDBs is that most OODBs convert the OIDs stored in an object to memory pointers when the object is loaded into memory. Suppose that both objects X and Y have been loaded into memory, and the OID stored as the value of attribute A of object X is converted to virtual memory pointer that points to object Y in memory. Then navigating from object X to object Y, that is, accessing object Y as the value of attribute A of object X, becomes essentially a memory pointer lookup. Figure 2.a illustrates the database representation of the objects of the classes Person and Company. Figure 2.b illustrates the memory representation of the same objects. The OIDs stored in the Worksfor attribute of the Person objects have been converted to memory addresses. Imagine that hundreds or thousands of objects have been loaded into memory, and that each object contains memory pointers to one or more other objects in memory. Further, imagine that navigation from one object to other objects is to be performed repeatedly. Since RDBs do not store OIDs, they cannot store in one tuple memory pointers to other tuples. The facility to navigate through memory-resident objects is a fundamentally absent feature in RDBs, and the performance drawback that results from it cannot be neutralized by simply having a large buffer space in memory. Therefore, for applications that require repeated navigation through linked objects loaded in memory, OODBs can dramatically outperform RDBs.

If all database applications require only OID lookups with database objects or memory-pointer chasing among objects in memory, the 2 to 3 orders of magnitude performance advantage for OODBs over RDBs is very much valid. However, most applications that require OID lookups also have database access and update requirements which RDBs have been designed to meet. These requirements include bulk database loading; creation, update, and delete of individual objects (one at a time); retrieval of one or more objects from a class that satisfy certain search conditions; joins of more than one classes (as we will see shortly); transaction commit; and so forth. For such applications, OODBs do not have any performance advantage to offer. In fact, even for the example database of Figure 1, if the objective of the application is to fetch Person objects, along with the related Company objects, that satisfy certain conditions (e.g., all Persons whose Age is greater than 25 and whose Salary is less than 40000 — i.e., a general query), rather than fetching a specific Company object for a given Person object (i.e., a simple navigation), OODBs may not enjoy any performance advantage at all, depending on how the OIDs are implemented and whether the query

Person

oid	name	age	salary	worksfor
115	John	25	25000	002
267	Chen	30	25000	001

Company

oid	name	age	president	location
001	Acme	15	Cohen	NY
002	UniSQL	3	Kim	Austin

Figure 1.a Object representation in an OODB

Person

name	age	salary	worksfor
John	25	25000	UniSQL
Chen	30	25000	Acme

Company

name	age	president	location
Acme	15	Cohen	NY
UniSQL	3	Kim	Austin

Figure 1.b Tuple representation in an RDB

optimizer is designed to exploit the OIDs in processing queries.

OODBs eliminate the need for joins

OODBs significantly reduce the need for joins of classes (comparable to joins of relations in RDBs); however, they do not eliminate the need altogether. In OODBs the domain of an attribute of a class C may be another class D. However, in RDBs the domain of an attribute of a relation R1 cannot be another relation R2. Therefore, to correlate a tuple of one relation with a tuple of some other relation, RDBs always require the users to explicitly join the two relations. OODBs replace this explicit join with an implicit join, namely the fetching of the OIDs of objects in a class that are stored as the values of an attribute in another class. The examples in Figure 1 illustrated this point. The specification of a class D as the domain of an attribute of another class C in an OODB is in essence a static specification of a join between the classes C and D.

The relational join is a general mechanism that correlates two relations on the basis of the values of a corresponding pair of attributes in the relations. Since two classes in an OODB may in general have corresponding pairs of attributes, the relational join is still useful and, therefore, necessary in OODBs. For example, in Figure 1, the classes Person and Company both have attributes Name and Age. Although the Name and Age attributes of the class Company are not the domains of the Name and Age attributes of the class Person, and vice versa, the user may wish to correlate the two classes on the basis of the values of these attributes (e.g., find all Person objects whose Age is less than the Age of the Company the Person Worksfor).

object identity eliminates the need for keys

Object identity has received more attention that it merits. Object identity is merely a means of representing an object, and also guaranteeing uniqueness of each individual object. An OID does not carry any additional semantics. Even if the OID lends uniqueness to each object, the OID is generated automatically by the system and usually not even made visible to the users. Therefore, it does not offer a convenient means of fetching specific desired objects from a large database (i.e., when the user does not know the OIDs of the objects). It is more convenient for the user to be able to fetch one or more objects using user-defined keys. For example, in the example database of Figure 1, if the Name attribute is a primary key, the user may fetch one Person object by issuing a query that searches for a specific Name.

OODBs eliminate the need for a (non-procedural) database language

This myth came about because most of the current OODBs offer only limited query capabilities. Vendors of the OODBs elected to focus their development efforts on the performance of database navigation, and making objects persistent. The commands necessary to invoke the limited database facilities have been presented to the users as calls to a library of database functions, that is, a procedural language. Upgrading most of the current OODBs to true database systems, in particular adding full query facilities comparable to those supported in RDBs, will necessitate a nonprocedural query language, which will be very difficult to hide. OODB vendors are now attempting to provide nonprocedural query languages, generally labeled as Object SQL.

query processing will violate encapsulation

One objective of encapsulating data and program into an object in OOPLs is to force the programmers to access objects only by invoking the program part of the objects, and keep the programmers from making use of knowledge of the data structures used to store the objects or the implementation of the program part. In the course of processing a query, the database system must read the contents of objects, extract OIDs that may be stored in some attributes of the objects, and retrieve objects that correspond to those OIDs. Object purists regard this as violating object encapsulation, since the database system examines the contents of objects. This view is not practical or useful. First, it is the database system that examines the contents of objects, not any ordinary user. Second, the act of examining the values stored in attributes of objects may be regarded as invoking the "get (or read)" method implicitly associated with every attribute of every class. If purity of objects must be preserved at all cost, then every single numeric and string constant used must be

Person

oid	name	age	salary	worksfor
115	John	25	25000	002
267	Chen	30	25000	001

Company

oid	name	age	president	location
001	Acme	15	Cohen	NY
002	UniSQL	3	Kim	Austin

Figure 2.a Object representation in database

Person

addr	name	age	salary	worksfor
040	John	25	25000	020
080	Chen	30	25000	004

Company

addr	name	age	president	location
004	Acme	15	Cohen	NY
020	UniSQL	3	Kim	Austin

Figure 2.b Object representation in memory

explicitly assigned an OID! But no known OOPL or OO application system does it.

OODBs can support versioning and long-duration transactions

There is a general misunderstanding that somehow OODBs can support versioning and long-duration transactions, and, by implication, versioning and long-duration transactions cannot be supported in RDBs. Although the paradigm shift from relations to objects does eliminate key deficiencies in RDBs, it does not address the issues of versioning and long-duration transactions. The object-oriented paradigm does not include versioning and long-duration transactions, just as the relational model of data does not include them. Simply put, C++ or Smalltalk does not include any versioning facilities or long-duration transaction facilities.

The reason versioning and long-duration transactions have become associated with OODBs is simply that they are database facilities that have been missing in RDBs and that have been identified as requirements for those applications that OODBs, with their more powerful data modeling facilities and object navigation facilities, can satisfy much better than RDBs (e.g., computer-aided engineering system, computer-aided authoring system, etc.). In fact, most OODBs do not even support versioning and long-duration transactions. The few OODBs that do offer what are labeled as versioning and long-duration transactions provide only primitive facilities.

Versioning and long-duration transactions can be supported in both OODBs and RDBs with equal ease or difficulty. Let us consider a few aspects of versioning. If an object is to be versioned, often a timestamp and/or version identity may need to be maintained. This can be implemented by creating system-defined attributes for the timestamp and/or version identity. Clearly, this can be done both for each versioned object in a class in OODBs and each versioned tuple in a relation in RDBs. Similarly, version-derivation history may be maintained in the database. Further, such versioning facilities as version derivation, version deletion, version retrieval, etc., may be expressed by extending the database language of OODBs and RDBs.

Next, let us consider long-duration transactions. A transaction is simply a collection of database reads and updates that are treated as a single unit. RDBs have implemented transactions with the assumption that they will interact with the database only for a few seconds or less. This assumption becomes invalid and long-duration transactions become necessary in environments where human users interactively access the database over much longer durations (hours or days). Regardless of the duration of a transaction, a transaction is merely a mechanism for ensuring database consistency in the presence of simultaneous accesses to the database by multiple users and in the presence of system crashes. What differentiates an OODB from an RDB is the data model, that is, how data is represented (i.e., attributes and methods, and classes and class hierarchy in an OODB vs. attributes and relations in an RDB). It should be clear that the paradigm difference between RDBs and OODBs does not solve the problems that transactions are designed to solve.

OODBs can support multimedia data

OODBs are a much more natural basis, than RDBs, for implementing functions necessary for managing multimedia data. Multimedia data is broadly defined as data of arbitrary type (number, short string, Employee, Company, image, audio, text, graphics, movie, a document that contains images and text, etc.) and arbitrary size (one byte, 10K bytes, 1 gigabyte, etc.). The reason is that OODBs allow arbitrary data types to be created and used, the first requirements for managing multimedia data.

However, object-oriented paradigm (i.e., encapsulation, inheritance, methods, arbitrary data types — collectively or individually) does not solve the problems of storing, retrieving, and updating very large multimedia objects (e.g., an image, an audio passage, a textual document, a movie, etc.). OODBs must solve exactly the same engineering problems that RDBs have had to solve to allow the BLOB (binary large object) as the domain of a column in a relation, including incremental retrieval of a very large object from the database (the page buffer in general cannot hold the entire object), incremental update (a small change in an object should not result in a copying of the entire object), concurrency control (more than one user should be able to access the same large object simultaneously), and recovery (logging should not lead to copying of an entire object).

4. Fulfilling the Promises of OODBs

Today, both the deficiencies of RDBs and the promises of OODBs are fairly well-understood. However, OODBs have not had significant impact in the database market. Two of the reasons are that most of the current OODBs lack maturity as database systems (i.e., they lack many of the key database facilities found in RDBs) and that they are not sufficiently compatible with RDBs (i.e., they do not support a superset of ANSI SQL).

The emerging industry and market consensus is that object-oriented technology can indeed bring about a quantum jump in database technology, but there are at least three major conditions that must be met before it can deliver on its promises.

First, new database systems that incorporate an object-oriented data model must be full-fledged database systems that are compatible with RDBs (i.e., whose database language must be a superset of SQL).

Second, application development tools and database access tools must be provided for such database systems, just as they are critical for the use of RDBs. The tools include graphical application (form) generator, graphical browser/editor/designer of the database, graphical report generator, database administration tool, and possibly others.

Third, a migration path (a bridge) is needed to allow co-existence of such systems with currently installed RDBs, so that the installations may use RDBs and new systems for different purposes and also to gradually migrate from their current products to the new products.

In this section, I will provide an outline of how an object-oriented database system may be built that is fully compatible with RDBs, and how a migration path may be provided from RDBs to such a new database system. UniSQL, Inc. has a commercial database system, UniSQL/X, that supports a superset of ANSI SQL with full object-oriented

extensions. UniSQL, Inc. also offers graphical database access tools and application generation tool for use with UniSQL/X. Further, UniSQL, Inc. offers a commercial federated (multi) database system, UniSQL/M, that allows co-existence of UniSQL/X with RDBs, while giving the users a single-database illusion. I will use UniSQL/X and UniSQL/M to illustrate key concepts in this section.

Unification of the relational and object-oriented technologies is most definitely the underpinning for post-relational database technology. ORACLE Corporation recently announced plans to develop an object-oriented extension to SQL. The ANSI SQL3 standards committee is currently designing object-oriented extensions to SQL2. The objective of SQL3 is exactly the same as that guided the development of the UniSQL/X database language. SQL3 is about 3-4 years away. Further, HP's OpenODB supports a database programming language called OSQL that is based on a combination of SQL and functional data model (rather than relational data model).There is also a proposal and initial implementation from Texas Instruments for a database programming language called ZQL[C++] that extends C++ with SQL-like query facility. The vendors of some OODBs are also preparing to develop "SQL-like" languages, generally labeled as Object SQL, that include facilities for defining and querying object-oriented databases, as an add-on to their existing OODBs. This represents a major direction change in their product strategy. Just a few years ago, these vendors merely attempted to provide gateways between their OODBs and some RDBs.

4.1 Unifying RDBs and OODBs

Unification Architectures

Broadly, there are three possible approaches to bringing together OODBs and RDBs: gateway, OO-layer on RDB engine, and a single engine. In the gateway approach, an OODB request is simply translated and routed to a single RDB for processing, and the result returned from the RDB is sent to the user issuing the original request. The gateway appears to the RDB as an ordinary user of the RDB. The current implementations of gateways impose various restrictions on the OODB requests; they either accept only read requests, only one request (rather than a sequence of requests as a single transaction), or only simple requests (i.e., not all types of queries comparable to those RDBs are capable of processing). Although the gateway approach makes it possible for an application program to use data retrieved from both an OODB and an RDB, it is not a serious alternative for unifying relational and object-oriented technologies. Its performance is unacceptable because of the cost of translating requests and returned data, and the communication overhead with the RDB. Further, its usability is unacceptable because the application programmers or users have to be aware of the existence of two different databases.

In the OO-layer approach (exemplified by HP's OpenODB), the user interacts with the system using an OODB database language (in the case of OpenODB, an ObjectSQL), and the OO layer performs all translations of the object-oriented aspects of the database language to their relational equivalents for interaction with the underlying RDB. The translation overhead can be significant, and this architecture inherently compromises performance. For example, the OO layer would map objects to tuples of relations, and generate the OIDs of objects and pass them to the RDB as an attribute of the tuple, using the interface the RDB makes available; it would also map an OID found in an object to its corresponding object stored in the RDB, again using the RDB interface; and so forth. An RDB consists of two layers: data manager layer and storage manager layer. The data manager layer processes the SQL statements, and the storage manager layer maps the data to the database. The OO layer may be interfaced with either the data manager layer (i.e., talk to the RDB via SQL statements) or the storage manager layer (i.e., talk to the RDB via low-level procedure calls). The data manager interface is much slower than the storage level interface. (OpenODB uses the data manager interface between its OO layer and the underlying RDB). Since this approach assumes that the underlying RDB will not be modified to better accommodate the needs of the OO layer, it can incur serious performance and operational problems when sophisticated database facilities need to be supported. For example, if a large number of classes in a class hierarchy must be locked (e.g., to support dynamic schema changes), the OO layer must either acquire locks one at a time (incurring a performance penalty and risking deadlocks), since an RDB has no provision for locking a class hierarchy atomically (roughly, in one command); or lock the entire database with one call to the underlying RDB (potentially preventing any other user from accessing any part of the database). Neither option is desirable. Further, if the OO layer is to support updates to objects in memory and automatically flush updated objects to the database when the application's transaction commits (finishes), the individual objects must be inserted back into the database one at a time, using the RDB interface.

The rationale for the OO-layer approach is to be able to port the OO layer on top of a variety of existing RDBs; this flexibility is obtained at the expense of performance. The OO-layer approach is the basis of a database system that makes a variety of databases appear to be a single database to application programs. Such a database system is known as a "multidatabase system". The OO-layer approach can be used as a basis of a multidatabase system that makes it possible for application programs to work with data retrieved from OODBs and RDBs. I note that OpenODB currently is not a multidatabase system. Its OO layer can connect to only one RDB. I will discuss multidatabase systems in greater detail later.

The unified approach melds the OO layer and the RDB into a single layer, while making all necessary changes in both the storage manager layer and the data manager layer of the RDB. The database system must fully support all the facilities the database language allows, including dynamic schema changes, automatic query optimization, automatic query processing, access methods (including B+-tree index, extensible hashing, external sorting), concurrency control, recovery from both soft and hard crashes, transaction management, and granting and revoking of authorizations. The richness of the unified data model added to implementation difficulties.

Unifying the Data Models

A relational database consists of a set of relations (tables), and a relation in turn consists of rows (tuples) and columns. A row/column entry in a relation may have a single value, and

the value may belong to a set of system–defined data types (e.g., integer, string, float, date, time, money). The user may impose further restrictions, called integrity constraints, on these values (e.g., the integer value of an employee age may be restricted to between 18 and 65). The user may then issue a nonprocedural query against a relation to retrieve only those tuples of the relation the values of whose columns satisfy user–specified conditions. Further, the user may correlate two or more relations by issuing a query that joins the relations on the basis of a comparison of the values in user–specified columns of the relations.

UniSQL/X generalizes and extends this simple data model in three ways, each reflecting a key object–oriented concept. A basic tenet of an object–oriented system or programming language is that the value of an object is also an object. The first UniSQL/X extension reflects this by allowing the value of a column of a relation to be a tuple of any arbitrary user–defined relation, rather than just an element of a system–defined data type (number, string, etc.). This means that the user may specify an arbitrary user–defined relation as the domain of a column of a relation. The first CREATE TABLE statement in Figure 3 shows the specification of an Employee relation under the relational model. The values of the Hobby and Manager columns are restricted to character strings. The second CREATE TABLE in Figure 3 reflects data–type extension for the columns of a relation. The value for the Hobby column no longer needs to be restricted to a character string; it may now be a tuple of a user–defined relation Activity. Similarly, the data type for the Manager attribute of the table Employee can even be the Employee relation itself.

Allowing a column of a relation to hold a tuple of another relation (i.e., data of arbitrary type) directly leads to nested relations; that is, the value of a row/column entry of a relation can now be a tuple of another relation, and the value can in turn be a tuple of another relation, and so forth, recursively. In Figure 1 we have seen how this conceptually simple extension may result in significant performance gain when retrieving data. This also gives a database system the potential to support such applications as multimedia systems (which manage image, audio, graphic, text data, and compound documents that comprise of such data), scientific data processing systems (which manipulate vectors, matrices, etc.), engineering and design systems (which deal with complex nested objects), and so forth. This is the basis for bridging the large gulf in data types supported in today's programming languages and database systems.

The second UniSQL/X extension is the object–oriented concept of encapsulation, that is, combining of data and program (procedure) to operate on the data. This is incorporated by allowing the users to attach procedures to a relation and have the procedures operate on the column values in each tuple. The third CREATE TABLE statement in Figure 3 shows the PROCEDURE clause for specifying a procedure, RetirementBenefits, which computes the retirement benefit for any given employee and returns a floating–point numeric value. Procedures for reading and updating the value of each column are impliciitly available in each relation.

A relation now encapsulates the state and behavior of its tuples; the state is the set of column values, and the behavior is the set of procedures that operate on the column values. The user may write any procedure and attach it to a relation to operate on the values of any tuple or tuples of the relation. There is virtually unlimited application of procedures.

The third UniSQL/X extension is the object–oriented concept of inheritance hierarchy. UniSQL/X allows the users to organize all relations in the database into a hierarchy, such that between a pair of relations P and C, P is made the parent of C, if C is to take (*inherit*) all columns and procedures defined in P, besides those defined in C. Further, it allows a table to have more than one parent relation from which it may take columns and procedures. The child relation is said to inherit columns and procedures from the parent relations (this is called *multiple inheritance*). The hierarchy of relations is a directed acyclic graph (rather than a tree) with a single

```
1. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT, Hobby CHAR(20), Manager CHAR(20));

2. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT, HOBBY Activity, Manager Employee);

        CREATE TABLE Activity
            (Name CHAR(20), NumPlayers INTEGER, Origin CHAR(20));

3. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT, HOBBY Activity, Manager Employee)
     PROCEDURE RetirementBenefits FLOAT ;

4. CREATE TABLE Employee
     (Job CHAR(20), Salary FLOAT, HOBBY Activity, Manager Employee)
     PROCEDURE RetirementBenefits FLOAT
     AS CHILD OF Person ;

        CREATE TABLE Person
            (Name CHAR(20), SSN CHAR(9), Age INTEGER);
```

Figure 3. Successive Extensions to the Relational Model

system–defined root. Further, an IS–A (generalization and specialization) relationship holds between a child relation and its parent relation. In the fourth CREATE TABLE in Figure 3, the Employee relation is defined as a CHILD OF another user–defined relation Person. The Employee relation automatically inherits the three columns of the Person relation; that is, the Employee relation will have the Name, SSN, and Age columns, even if they are not specified in its definition.

The relation hierarchy offers two advantages over the conventional relational model of a simple collection of largely independent (unrelated) relations. First, it makes it possible for a user to create a new relation as a child relation of one or more existing relations; the new relation inherits (reuses) all columns and procedures specified in the existing relations and their ancestor relations. Further, it makes it possible for the system to enforce the IS–A relationship between a pair of relations. RDBs require the users to manage and enforce this relationship.

Now, let us change the relational terms as follows. Change "relation" to "class", "tuple of a relation" to "instance of a class", "column" to "attribute", "procedure" to "method", "relation hierarchy" to "class hierarchy", "child relation" to "subclass", and "parent class" to "superclass". The UniSQL/X data model described above is an object–oriented data model! An object–oriented data model can be obtained by extending the relational model. The terms "object–oriented data model", "extended relational data model", and "unified relational and object–oriented data model (unified, for brevity)" become synonymous if the data model is obtained by augmenting the conventional relational data model with the first three extensions described above. However, an extended relational model (system) is not an object–oriented model (system), if it does not include all three extensions. Further, it is important to note that a database system based on such a model, because of its relational foundation, may be built by adapting all the theoretical underpinnings of the relational database technology that have been developed during the past two decades.

Although each of the three extensions individually may appear to be minor, the consequences of the extensions, individually and collectively, with respect to ease of application data modeling and/or subsequent increase in query performance can be significant. The nested relation extension eliminates the need for cumbersome workarounds that users of RDBs have had to resort to. The procedure and relation hierarchy extensions open up significant new possibilities in application data modeling and application programming. Further, the nested relation and relation hierarchy extensions reflect the powerful data type facilities of OOPLs.

Query and Data Manipulation

Of course, it is not enough just to define a data model that allows the users to represent complex data requirements. Once the database schema has been defined using the data definition facilities, the database may be populated with a large number of user–defined objects. The power of a database system comes into play when the users can retrieve and update tiny fractions of the database efficiently. To allow this, a database system provides query and data manipulation (insert, update, delete) facilities.

The UniSQL/X query language, unlike mere "SQL–like" object query languages, is a superset of ANSI SQL, and as such, if the extensions are removed from the syntax, it degenerates to ANSI SQL. By a "SQL–like" language I mean a database language that is either a subset of SQL or that does not support the same semantics of SQL. A SQL–like language that is a subset of SQL is one, for example, that does not support nested subqueries in the WHERE clause or aggregation functions in the SELECT clause, etc. It is also one that does not include facilities for defining and using views, or facilities for dynamically making changes to the database schema, or facilities for specifying the UNIQUE and NULL constraints on attributes of a class, or facilities for granting and revoking authorizations, and so forth. A SQL–like database language that does not support the same semantics of SQL is one, for example, that treats NULL values differently from SQL, or that refuses to commit a transaction after accepting all read and update requests from the user without any complaints, or that introduces a restriction that does not exist in SQL (e.g., the DROP CLASS command does not allow a class to be dropped if any objects still belong to a class, while the DROP TABLE command in SQL results in the dropping of a table and all its tuples, whether or not there are tuples), and so forth.

If a set of classes are defined just as relations in conventional relational databases, the users of the UniSQL/X query language may issue all queries in ANSI SQL syntax, including joins and nested subqueries, queries that group and order the results, and queries against views. Let us consider two simple examples using Figure 4. In the figure, the class Employee is defined as a subclass of the class Person, and the class Activity is the domain of the attribute Hobby of the class Employee. The first query finds all employees who earn more than 50000 and are over 30 years of age, and outputs the average salary of all such employees by job category. The second query is a join query, which finds the names of all employees who earn more than their managers.

```
SELECT   Job, Avg (Salary)
FROM     Employee
WHERE    Salary < 50000 AND
         Age > 30
 GROUP BY   Job ;
```

```
SELECT   Employee.Name
FROM     Employee
WHERE    Employee.Salary > Employee.Manager.Salary;
```

The UniSQL/X query language also allows the formulation of a number of additional types of queries that become necessary under the unified data model (i.e., queries that are not applicable under the relational model). The unified data model is richer, and thus it gives rise to query expressions that do not arise in RDBs. In particular, it allows *path queries*, that is, queries against nested classes; queries that include methods as part of search conditions; queries that return nested objects; and queries against a set of classes in the class hierarchy.

An example of a query on a class hierarchy is to retrieve instances from a class and all its subclasses. In the following query, the keyword *ALL* causes the query to be evaluated against the class Person and its subclass Employee.

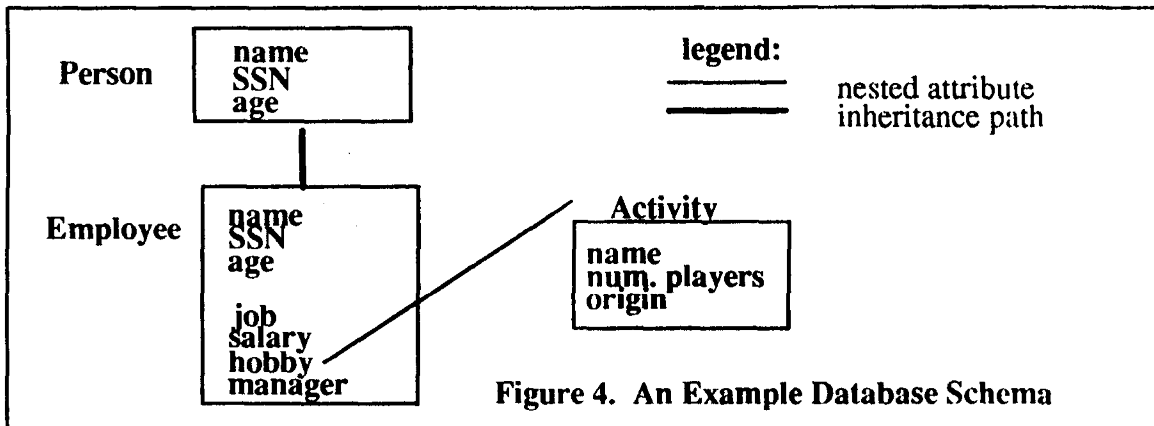

Figure 4. An Example Database Schema

SELECT Name, SSN
FROM ALL Person
WHERE age > 50;

An example of a path query that retrieves nested objects, using Figure 4, is "Find the names of all employees and their employers for those employees who earn more than $50,000 and whose hobby is tennis". This query is evaluated against the nested objects defined by the classes Employee and Activity. The query is formulated by associating the predicate (Name = 'tennis') with the class Activity, and the predicate 'Salary > 50000' with the class Employee. The query returns all attributes of Employee from the nested Employee objects that satisfy the query conditions.

*SELECT **
FROM Employee
WHERE Salary > 50000 AND
Hobby.Name = "Tennis";

The dot notation in the predicate (Hobby.Name = "Tennis") extends the standard predicate expression to account for the nesting of attributes through the use of arbitrary data types.

Support for Object Navigation

Like some OODBs that are designed to make OOPL objects persistent, UniSQL/X provides workspace management facilities to automatically manage a large number of objects in memory (called a workspace or an object buffer pool). In particular, UniSQL/X automatically converts the storage format of objects between the database format and the memory format, automatically converts the OIDs stored in objects to memory pointers when objects are loaded from the database into memory, and automatically flushes (writes) objects updated in memory to the database when a the transaction that updated them finish.

These workspace management facilities in UniSQL/X make it possible for database application programs to navigate memory-resident objects via memory-pointer chasing, and to propagate changes to individual objects collectively to the database. RDB applications must resort to explicit queries that either join two relations or at least search a single relation to emulate the simple navigation from one object to another related object. Further, RDB applications must also propagate updated tuples one at a time to the database, via the RDB interface (either the data manager level or storage manager level). When a transaction finishes, UniSQL/X automatically sends all objects created or updated by the transaction to the database to make them persistent. UniSQL/X application programs do not need to do anything to propagate the changes to the database.

I note that, unlike most OODBs that also provide workspace management facilities, UniSQL/X supports full query facilities and full dynamic schema evolution. Since at any point in time, an object may exist both in the database and in the workspace, and the "copy" in the workspace may have been updated, a query must be evaluated against the "copies" in the workspace for those objects that have been loaded into the workspace, and against the database objects for those objects that have not been loaded into the workspace. Further, if the user makes a schema change (e.g., drop an attribute of a class, or add an attribute to a class), the "copies" of objects in the workspace become invalid. UniSQL/X takes full account of these considerations in its support of automatic query processing and dynamic schema evolution.

Further, workspace management facilities are essential for making objects persistent and for supporting the performance requirements in object navigation for application programs written in OOPLs. Although UniSQL/X is not wedded to any particular OOPL, the sophisticated workspace management facilities provided in UniSQL/X mean that a rather simple translation layer may be implemented on top of UniSQL/X to support any particular OOPL (e.g., C++ or Smalltalk).

5. Interoperating with RDBs

The gateway approach that I discussed as an (unsatisfactory) alternative for unifying an OODB with RDBs serves one useful purpose. It allows an OODB and RDBs to coexist, and can potentially make it possible for one application program to work with data retrieved from both an OODB and one or more RDBs. As I remarked already, however, the current OODB-RDB gateways typically pass requests to only one RDB (e.g., to Sybase or to ORACLE), and do not treat the separate requests to an OODB and to RDB as a single transaction (i.e., collection of requests that is treated as a single unit).

A multidatabase system (MDBS) is logically a full generalization of a gateway. An MDBS is actually a database system that controls multiple gateways. It does not have its own database; it merely manages remote databases through the gateways, one gateway for each remote database. An MDBS presents the multiple remote databases as a single "virtual" database to its users. Since an MDBS does not have

its own "real" database, certain database facilities, such as those for managing access methods (creating and dropping B+-tree index, extendible hash table, etc.) and parameterized performance tuning, become meaningless.

However, an MDBS is a nearly full-fledged database system. An MDBS must provide data definition facilities so that the virtual database may be defined on the basis of the remote databases. The data definition facilities need to include means to harmonize (homogenize) the different representations of the semantically equivalent data in different remote databases. An MDBS user may query the definition of the virtual database, query and update the virtual database (requiring query optimization and query processing mechanisms). Multiple MDBS users may simultaneously query, update, and even populate the "virtual" database (requiring concurrency control mechanisms); the users may submit a collection of queries and updates as a single transaction against the virtual database (requiring transaction management mechanisms); the users would grant and revoke authorizations on parts of the database to other users (requiring authorization mechanisms).

To translate MDBS queries and updates to equivalent queries and updates that can be processed by remote database systems, an MDBS requires gateways for remote database systems. The gateways in an MDBS are often called "drivers" and remote database systems are called "local" database systems, and the single virtual database that an MDBS presents to its users is called a "global" database. Further, an MDBS is said to "integrate" multiple local databases into a single global database.

UniSQL/M is a multidatabase system from UniSQL, Inc. that integrates multiple UniSQL/X databases and multiple relational databases. UniSQL/M is UniSQL/X augmented to access external relational databases and UniSQL/X databases; as such, it is a full-fledged database system and UniSQL/M users can query and update the global database in the SQL/X database language. UniSQL/M maintains the global database as a collection of views defined over relations in local RDBs and classes in local UniSQL/X databases. UniSQL/M also maintains a directory of the local database relations and classes, their attributes and data types, and methods, that have been integrated into the global database. Using the information in the directory, UniSQL/M translates the queries and updates to equivalent queries and updates for processing by local database systems that manage the data that the queries and updates need to access. The local database drivers pass the translated queries and updates to local database systems, and pass the results to UniSQL/M for format translation, merging, and any necessary postprocessing (e.g., sorting, grouping, and joining). Further, UniSQL/M supports "distributed transaction management" over local databases, which means that all updates issued within one UniSQL/M transaction, even when they result in updates to multiple local databases, are simultaneously committed or aborted.

RDB vendors today offer gateways of different levels of sophistication. Some gateways allows SQL queries to be passed to a hierarchical database system (namely, IMS) or file systems such as DEC's RMS. Some gateway is currently being upgraded to accept both queries and updates, and even support distributed transaction management over local databases. However, none of these gateways are designed to pass SQL queries to OODBs; there has been little need to develop such gateways.

UniSQL/M differs from the gateways currently offered by RDB vendors and OODB vendors in three major ways.

- UniSQL/M is a full-fledged database system, rather than a mere gateway, supporting queries, updates, authorization, and transaction management over the global database (the specifications of views defined over local database tables and classes, and directory of information about local database tables and classes). Most current gateways do not accept updates.

- UniSQL/M connects to and coordinates queries and updates to multiple local databases for a single UniSQL/M transaction; in particular, it supports distributed transaction management over local databases. Most current gateways pass requests to only one local database, or do not allow simultaneous updates to multiple local databases within a single transaction, when they do support multiple local databases.

There is one more powerful advantage that UniSQL/M offers over any of the current gateways. UniSQL/M extends, although not fully (due to theoretical limitations), local RDBs to UniSQL/X; that is, UniSQL/M converts the tuples retrieved from relational local databases into objects by augmenting them with object identifiers and allowing the users to attach methods to them. In this way, UniSQL/M makes key object-oriented facilities provided in UniSQL/X indirectly available to local RDBs; in particular, SQL/X path queries, methods, and workspace management for objects in UniSQL/M memory.

UniSQL/M may be used in at least three different contexts. First, it may be used to allow co-existence of UniSQL/X with RDBs. Second, it may be used to turn a collection of RDBs (or a collection of UniSQL/X's) into a distributed database system. Third, when interfaced to a single RDB, it acts as the object management layer for the RDB engine, turning the RDB into UniSQL/X.