On Unifying Relational and Object–Oriented Database Systems

Won Kim

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

Abstract. The past several years have been the gestation period for a new generation of database technology. There has been a flurry of activities to develop and experiment with database systems that support an object–oriented data model or that extend the relational data model with some object–oriented facilities. These activities have been fueled by the emergence of a broad spectrum of database applications which relational database systems cannot support and the increasing need to achieve another productivity leap in application development. As a result of these efforts, there is now a sufficient body of knowledge for the development of a commercially viable next–generation database system. Such a system should support a unified relational and object–oriented data model; that is, a full object–oriented data model in a way that is completely compatible with the relational model. This article examines motivations for unifying the relational and object–oriented data models in a single database system, and outlines design and implementation issues that must be addressed in building such a system.

1 Introduction

During the 1980s, relational database systems [11, 12, 13, 33] dominated the database market for business data processing applications. The relational database language SQL [3] became an industry standard. The theory, implementation, and use of database systems became a major discipline of computer science. The simplicity of the relational model of data and the dynamic management of a database have been accepted as vehicles for significant productivity enhancement in application development.

However, even as the acceptance of relational database systems spread, their limitations were exposed by the emergence of various classes of new applications. These applications include multimedia systems, statistical and scientific modeling and analysis systems, geographic information systems, engineering and design systems, knowledge–based systems, and so on. The limitations of relational database systems that these applications exposed fall into two categories: data model and computational model. The data–modeling facilities that relational systems lack include those for specifying, querying and updating

complex nested entities (such as design and engineering objects, and compound documents); arbitrary user–defined data types; frequently useful relationships, such as generalization and aggregation relationships; temporal evolution of data (i.e., temporal dimension of data, and versioning of data); and so on. The computation–modeling facilities that relational systems lack include the management of memory–resident objects for extensive pointer–chasing applications (e.g., simulation of a computer–aided design); long–duration, cooperative transactions; and so on.

The need to reduce the cost of developing and operating these applications pointed to the need for a fundamental advancement in database technology, that is, a paradigm shift, rather than incremental extensions of the capabilities of existing data management systems. The basis of this fundamental advancement in database technology is the object–oriented paradigm developed in object–oriented programming languages. There are two major reasons the object–oriented paradigm is a sound basis for a new generation of database technology.

One is that the object–oriented paradigm [16, 29, 32] can be the basis for a data model which subsumes the relational (and pre–relational) data models. Solutions to most of the data–modeling–related difficulties of relational database systems are inherent in an object–oriented data model. Relational systems are designed to manage only limited types of data, such as integer, floating–point number, string, Boolean, date, time, and money. In other words, they are not designed to manage arbitrary user–defined data types. On the other hand, a central tenet of an object–oriented data model is the uniform treatment of arbitrary data types and the facility to add new data types. Further, an object–oriented data model allows the representation of not only data, and relationships and constraints on the data, as the relational data model does; but also the encapsulation of data and programs that operate on the data, and provides a uniform framework for the treatment of arbitrary user–defined data types.

Another reason is that the object–oriented paradigm, through the notions of encapsulation and inheritance (reuse), is fundamentally designed to reduce the difficulty of developing and evolving complex software systems or designs. 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 object–oriented data model thus inherently satisfies the objective of reducing the difficulty of design and evolving very large and complex databases. The notions of encapsulation and inheritance are a key to the search for further productivity enhancement in database application development.

The promises of a database technology based on an object–oriented data model are clear. These promises fueled, during the past several years, a rush to develop a first wave of object–oriented database systems [30, 2, 8, 15, 14, 20]. About a half dozen systems are currently commercially available; some are persistent storage systems for C++ applications, some provide low–level support for engineering and design applications; and some offer a relatively richer set of database features [25].

If the object–oriented technology can truly deliver the fundamental advance in database technology to the overall database market, that is, transition the database technology past the

current relational technology, it needs to be unified with the relational technology. The relational paradigm should be extended with a set of fundamental object–oriented concepts found in most object–oriented programming languages and application systems. The database language that embodies the united modeling paradigm should be a minimal extension to the SQL relational database language. The database language should then be embedded in a wide variety of host programming lanugages to bring object–oriented modeling facilities to programmers of these languages.

SQL–compatibility is a key to the adoption of new database systems that incorporate object–oriented modeling facilities. It will minimize the time necessary for the current relational users to learn the new technology, and facilitate the migration of relational applications to the new systems. However, SQL–compatibility will not solve the mismatch that has existed until now between a database language and the host programming language in which the database language is embedded. Ideally, database application programmers should use a single database programming language for both general–purpose programming and database management. Once object–oriented programming languages become solidly established as database application programming languages, database systems that "seamlessly" support the languages will certainly be useful.

Some vendors currently offer unified database programming languages by augmenting object–oriented programming languages, notably, C++ and Smalltalk, with persistent storage, transaction management, and limited query facilities. However, important challenges remain. Persistent storage and database management facilities (especially the query facilities) must be provided without introducing obtrusive syntax or semantics that is different from that of the programming language being augmented. Database management facilities and persistent storage should be provided to objects of all data types allowed in the programming language. Further, database management facilities should be provided to not only persistent objects, but also nonpersistent objects. Finally, the extensions to one programming language should be applicable with minimal changes to a variety of other programming languages. It is important to note that, regardless of whether the database language is an extension of SQL or some object–oriented programming language, most of the considerations outlined in this paper remain valid. Ultimately, a database system must address the impact of object–oriented modeling concepts on the database architecture.

A formidable array of technical challenges must be overcome before a next–generation database system can be built that incorporates object–oriented modeling concepts into relational database systems. The purpose of this paper is to outline these challenges and how they may be met. First, the relational and object–oriented data models must be combined into a single coherent unified model. Second, a database language must be designed to allow the specification of data and relationships among them, and the querying and updating of the data. Third, the database system must be built to fully support all the facilities the database language allows. Fourth, the system must deliver high performance; in particular, it must deliver performance comparable to relational database systems for all operations that can be performed using relational database systems. The strength of an object–oriented data model is also its weakness; the richness of an object–oriented data model that makes it possible to

model complex objects and their relationships in nonbusiness data–processing applications necessarily introduces added complexities with which the users must cope. The richness of an object–oriented data model also implies added difficulties in implementing an object–oriented database system. The designers of the unified database language and database system must carefully revisit the philosophy and design rationale behind relational database systems, and address a host of technical problems in database architecture.

2 Design Issues

A database system is in essence a software that implements all the functions supported in a database language. A database language in turn is an embodiment of a data model. For example, the relational database language SQL stipulates how the users of a database system should create tables; specify data types and integrity constraints on the tables and columns within tables; populate the tables with rows; query, update, and delete the contents of the database; and so forth. A data model is thus the foundation of any database system.

Further, a database language consists of three sublanguages: data–definition, query and data manipulation, and data control language. A data–definition language allows the definition of a database. A query and data manipulation language allows the database to be populated, and the database contents to be queried, updated, and deleted. A data control language allows the specification of database sharing and administration.

In this section, I will show first that a unified relational and object–oriented data model is in essence an object–oriented data model obtained by extending the relational model with key concepts found in object–oriented programming. Next, I will outline in terms of the three database languages architectural issues in building a unified database system.

2.1 Data Model

The primary advantage of the relational model of data is its simplicity; however, simplicity is also a major disadvantage of the relational model. A relational database consists of a set of relations (tables), and a relation in turn consists of rows and columns. An entry in a table 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).

This simple data model is really a subset of an object–oriented data model; that is, the relational model can be generalized to arrive at an object–oriented data model. First, let us regard a table as a data type, and allow each entry of a table to be a single value belonging to any arbitrary user–defined table, rather than just the system–defined data types. The first CREATE TABLE statement in Figure 1 shows the specification of an Employee table under the relational model. The values of the Hobby and WorksFor columns are restricted to character strings. The second CREATE TABLE in Figure 1 reflects data–type generalization for the columns of a table. The value for the Hobby column no longer needs to be restricted to a character string; it may now be a row of a user–defined table Activity.

Allowing a column of a table to hold a row of another table (i.e., data of arbitrary type) directly leads to nested tables; that is, the value of an entry of a table can now be a row of another table, and the value of an entry of that table can in turn be a row of another table, and so forth, recursively. This 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). This is the basis for bridging the large gulf in data types supported in today's programming languages and database systems.

Next, let us allow the entry of a table to have any number of values, rather than just a single value; and further allow the set of values to be of more than one data type. The restriction in the relational model that the entry of a table may hold only a single value forces

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

2. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT,
     Hobby ACTIVITY, WorksFor COMPANY);

        CREATE TABLE Company
            (Name CHAR(20), Location STATECITY, Budget FLOAT);

        CREATE TABLE StateCity
            (State CHAR(20), City CHAR(20));

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

3. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT,
     Hobby set–of ACTIVITY, WorksFor COMPANY);

4. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT,
      Hobby set–of ACTIVITY, WorksFor COMPANY)
     PROCEDURE RetirementBenefits FLOAT ;

5. CREATE TABLE Employee
     (Name CHAR(20), Job CHAR(20), Salary FLOAT,
      Hobby set–of ACTIVITY, WorksFor COMPANY)
     PROCEDURE RetirementBenefits FLOAT
      AS CHILD OF Person ;

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

Figure 1. Successive Extensions to the Relational Model

the creation of an extra table if the column of a table should hold more than one value. For example, suppose that an employee may have more than one hobby. It is not possible to define the Hobby column in the Employee table to store employee hobbies, and thus an extra table, say EmployeeHobby, needs to be created. The Employee and EmployeeHobby tables must be joined to retrieve information about employee's hobbies. The third CREATE TABLE statement in Figure 1 shows that the data type of the Hobby column is a *set–of* Activity, that is, the value of the Hobby column may be a set of rows of a single user–defined table Activity.

Next, let us allow a table to have procedures that operate on the column values in each row [31]. The fourth CREATE TABLE statement in Figure 1 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. A column may be regarded as a restricted procedure.

A table now encapsulates a state and a behavior of its rows; 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 table to operate on the values of any row or rows of the table. There is virtually unlimited application of procedures. It is often convenient to think of columns as special procedures; the system provides "procedures" that read or update the values of columns.

Next, let us organize all tables in the database into a hierarchy, such that between a pair of tables 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, let us allow C to have more than one parent table from which it may take columns and procedures (this is called *multiple inheritance*). The hierarchy of tables is a directed acylic graph, with a single system–defined root. The child table is said to inherit columns and procedures from the parent table. Further, an IS–A (generalization and specialization) relationship holds between a child table and its parent table [28]. In the fifth CREATE TABLE in Figure 1, the Employee table is defined as a CHILD OF another user–defined table Person. The Employee table automatically inherits the three columns of the Person table; that is, the Employee table will now have the SSN, Age, and Address columns.

The hierarchical organization of tables also presents two types of *name conflict* problem [29, 6]. First, the names of the columns and procedures defined for a child table may be the same as those defined in a parent table. In this case, the names in the child table will override those of the parent table, that is, the child table does not take columns and procedures of the parent table by the same names. Second, if a child table is to have more than one parent table, more than one parent table may have columns and procedures with the same names. In this case, there needs to be a rule by which the child table may take appropriate columns and procedures.

This table hierarchy, with the semantics defined above, offers two advantages over the conventional relational model of a simple collection of largely independent (unrelated) tables. First, it makes it possible for a user to create a new table as a child table of an existing table; the new table inherits (reuses) all columns and procedures specified in the existing table

and its ancestor tables. Further, the IS–A relationship between a pair of tables is specified to the system, making it possible for the system to enforce and manage that relationship. In relational systems, this relationship must be managed (programmed) in individual applications by the application programmers.

Now, let us change the relational terms as follows. Change "table" to "class", "row of a table" to "instance of a class", "column" to "attribute", "procedure" to "method", "table hierarchy" to "class hierarchy", "child table" to "subclass", and "parent class" to "superclass". It should be clear that the data model described above is an object–oriented data model [5, 22]. An object–oriented data model can thus be viewed as an extended 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 includes all five types of extensions to the relational model described above.

It is important to note that the unified model described here by no means includes all useful data modeling concepts, for example, ordering in a set, bags, inverted functions, etc. Rather, the model includes a set of modeling concepts that are generally regarded as fundamental and most widely useful, and can of course be tuned with additional, relatively minor, modeling concepts. Further, it should be noted that a database system based on such a model, because of its relational foundation, may be built by adopting all the theoretical underpinnings of the relational database technology that have been developed during the past two decades.

Let us see the power of the unified data model in modeling a complex object, for example a memo shown in Figure 2 (adapted from [34]). The memo consists of a header and a body (ignore the trailer for now). The header in turn consists of a logo, TO–line, FROM–line, Date–line, and Subject–line. The body consists of a paragraph, a drawing, another paragraph, and an image. The drawing in turn consists of three subdrawings, each consisting of a box and a text. Figure 3 models the hierarchical composition of an abstract data type Memo. Memo consists of lower level abstract data types Header and Body, each of which in turn consists of even lower level abstract data types.

The compositional structure of Memo may be specified in the SQL/X data–definition language as shown in Figure 4. The class Memo is defined with two attributes, Header and Body; the data type of the Header attribute is the MemoHeader class, while the data type of the Body attribute is a heterogeneous set of Para, Drawing, and Image. The remainder of the classes are similarly specified.

Each of the nodes on the aggregation hierarchy represents a type (class). Further, new types may be added to the aggregation hierarchy. For example, the structure of Memo may be extended later to include a new component, namely the Trailer, which in turn may consist of an Attach–List and a CC–List, as shown in Figure 3. Further, Memo, being a class, can also be specialized into subclasses. Figure 5 shows a class hierarchy specialized from Memo.

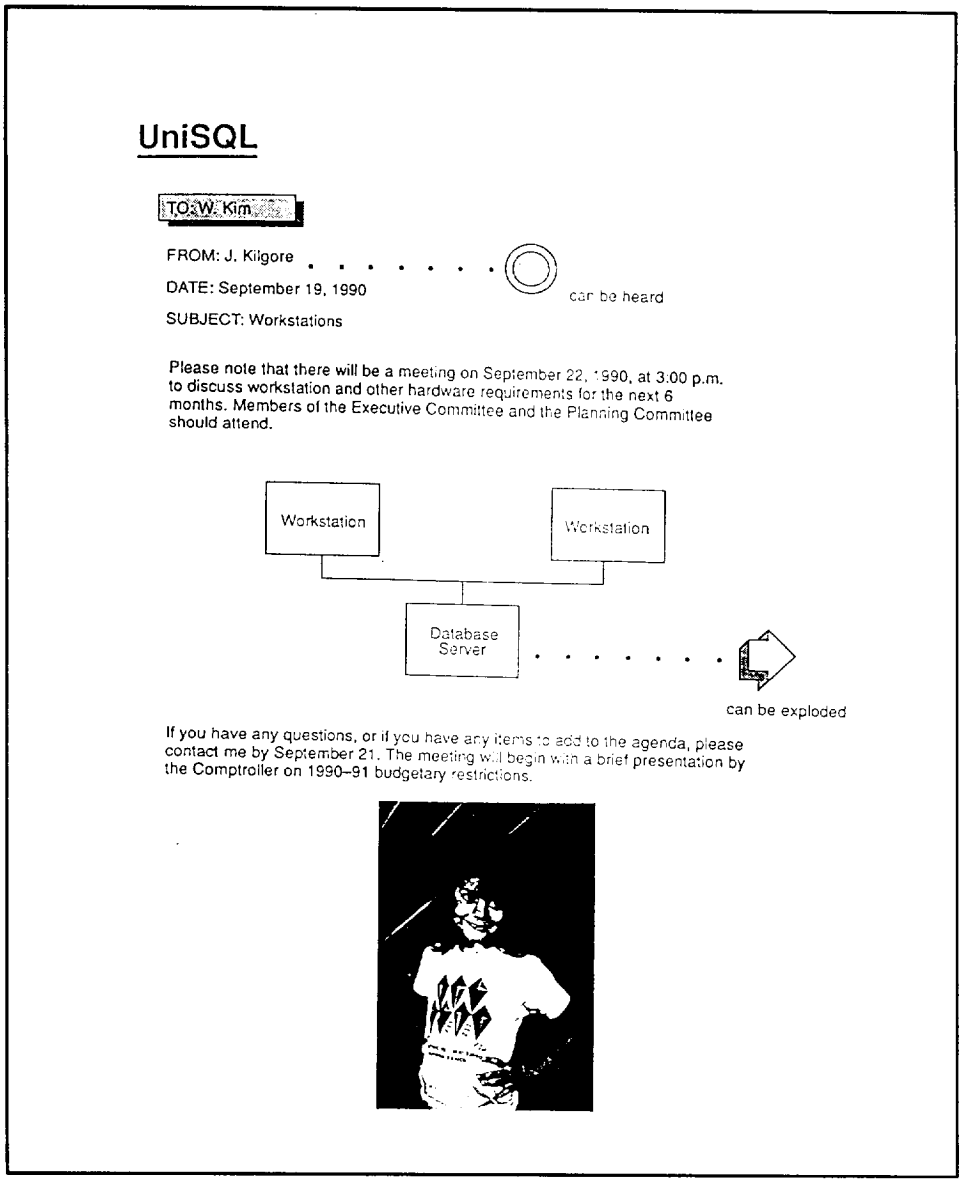

Figure 2. An Example Multimedia Document

2.2 Database Language and Its Implementation

The inclusion of object–oriented modeling concepts has significant impact on the architecture of a database system. In particular, it requires extensions to the conventional semantics of database schema definition, schema changes, queries, authorization, indexing, clustering, and even concurrency control. In this section, I will brief outline the issues in extending relational database systems with support for object–oriented concepts. The

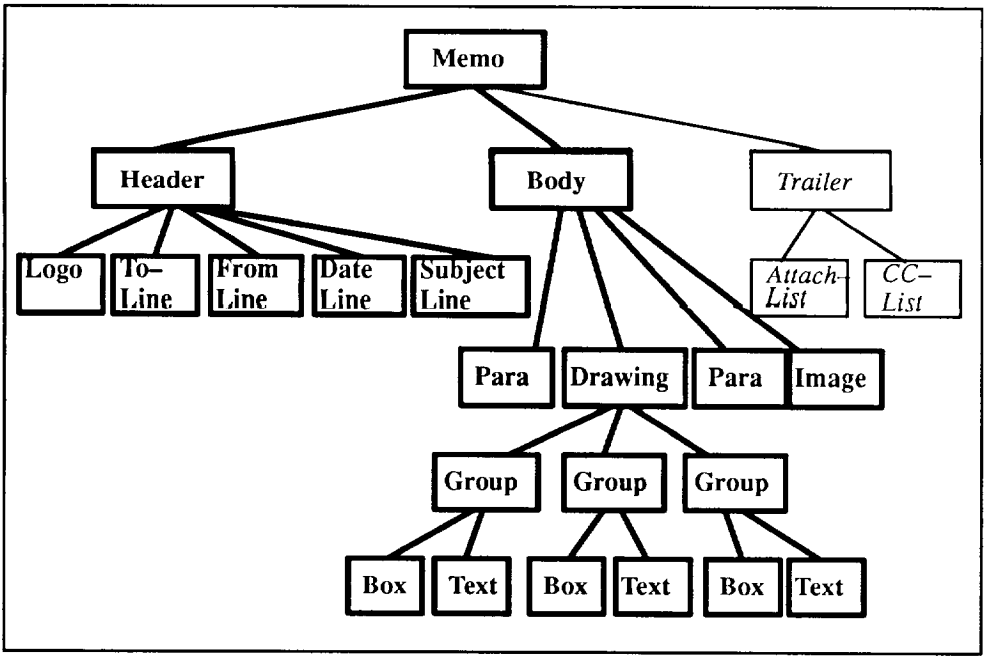

Figure 3. An Aggregation Hierarchy for a Compound Document

```
CREATE CLASS Memo
   (Header MEMOHEADER, Body set–of (PARA, DRAWING, IMAGE))

CREATE CLASS MemoHeader
    (Logo Image, To–Line CHAR(20), From–Line CHAR(20),
     Date–Line Date, Subject–Line CHAR(20)

CREATE CLASS Drawing
     (Grouping set–of GROUP)

CREATE CLASS Group
     (BoxText set–of (BOX, TEXT))
```

Figure 4. Schema Definition in SQL/X for the Database of Figure 3

interested reader should see [21] (or [23]) for a fuller discussion of the impact of object–oriented modeling concepts on the architecture of a database system; and pointers in the literature to descriptions of the solutions to many of the issues.

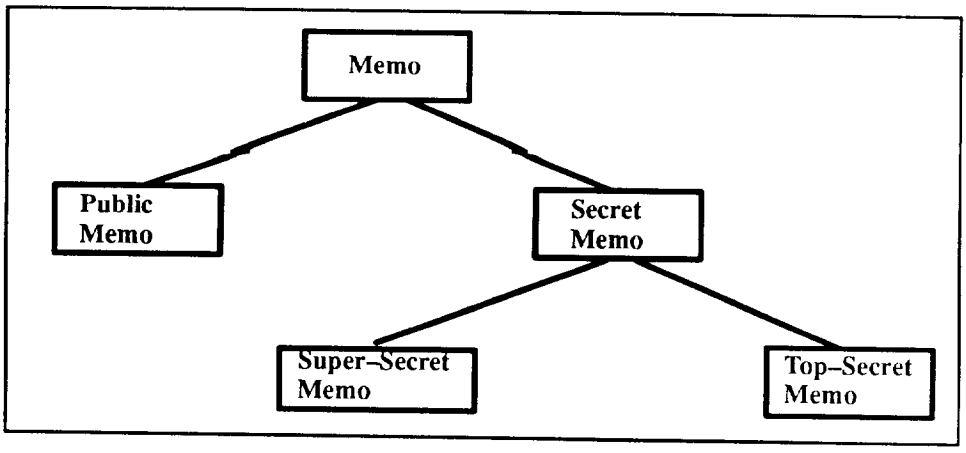

Figure 5. A Class Hierarchy for a Compound Document

Database Definition

The simplicity of the relational model means first of all a simple definition of the database. The user creates tables by specifying the table name, names of the columns of the table, data types for the columns, and integrity constraints on the table and/or the columns.

The inclusion of the object–oriented extensions in the unified data model leads to a more complex specification of the database. The user creates a class by specifying, besides those specified for a "comparable" table, methods, names of the superclasses from which the class is to inherit attributes and methods, and, in case of name conflict in inheritance, specification of conflict resolution.

Just as relational database systems allow the database definition to be changed dynamically, that is, without requiring system shutdown, so must a unified database system. However, the added complexity in database definition also means complications in dynamic changes to the database definition [6]. In a relational database, a limited set of dynamic changes to the database definition is meaningful. A new table may be created, a table may be dropped, a new column may be added to an existing table, and, possibly a column of a table may be dropped. A unified database has additional elements in its definition, and thus there are many more types of dynamic changes that need to be supported. These include adding and dropping methods, adding and dropping a superclass to a class, and even changing the data type of an attribute.

The semantics of some of the changes to the database definition are rather complex, including dropping of a class, and adding and dropping of a superclass. In a relational database, it is simple to drop a table; assuming that no foreign key or referential integrity is being enforced, a table is dropped, and all rows that belong to it are automatically dropped. In a unified database, a class and all its instances are similarly dropped; however, all attributes and methods inherited by subclasses also need to be dropped from the subclasses recursively, and, if the class was the data type of any attribute of any other class, the attribute now will

lose its data type. The adding and dropping of a superclass to an existing class is of course not applicable to relational systems. The name conflicts that may arise when a new superclass is added to a class or an existing superclass is dropped from a class must be resolved in a manner consistent with the semantics of multiple inheritance and name conflict resolution defined for the unified data model.

The semantics of changes to the database definition must also be compatible with those of comparable changes in relational database systems. An example is the dropping of a class. Some systems do not allow a class to be dropped if there are instances that belong to the class. This is incompatible with the approach taken in relational database systems. Further, the adding and dropping of a column in a table does not force immediate database reorganization in relational systems. The unified database system should adopt the same philosophy and avoid immediate database reorganization to the extent possible. In particular, such changes as adding or dropping an attribute or a method to a class, and adding and dropping a superclass to a class should be implemented by making appropriate changes to the data dictionary (system catalogs), and the database should be cleaned up at some later time in a background mode.

Further, relational systems present the data dictionary as tables to the users, allowing the users to query the data dictionary using the normal query language. The unified database system should present the data dictionary as classes to the users and allow the users to query it using the query language.

Query and Data Manipulation

Nonprocedural query languages and automatic optimization of queries are the essence of the current database technology. Insofar as an object–oriented data model subsumes the relational model, there is no reason to abandon these cornerstones of the relational database technology. Therefore, these are major requirements that must be met by a unified database system.

The query language of a unified database system must satisfy the following four difficult criteria if the system is to be commercially and technically viable. First of all, it must subsume the relational query language. Against a set of classes that are defined just as tables in conventional relational databases, it must be possible to issue the same set of queries that may be issued using the relational query language, including joins and subqueries, and grouping and ordering of the query result. Further, the unified query language must include facilities to define views and issue queries against views.

Second, it must allow the formulation of a set of additional types of queries 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 relational database systems [9, 10, 27, 19, 18]. In particular, it must allow *path queries* [35, 18], 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

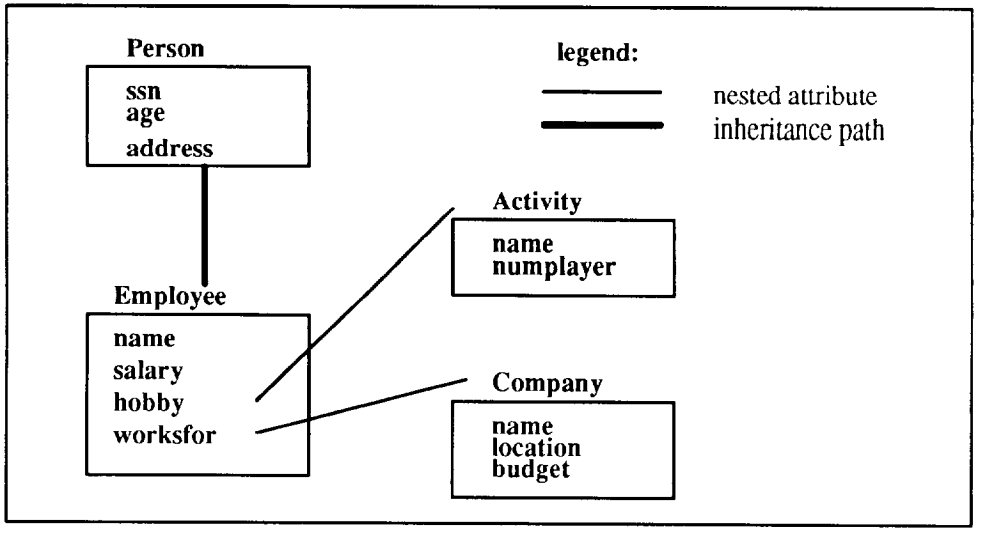

Figure 6. An Example Database

all its subclasses; for example, Person and Employee in the example database definition shown in Figure 6. An example of a path query that retrieves nested objects, using Figure 6, is "Find the names of all employees and their employers for whose employees who earn more than $50,000 and who work for companies that are located in Austin". This query is evaluated against the nested objects defined by the classes Employee and Company; the class Company is the data type of the WorksFor attribute of the class Employee. The query is formulated by associating the predicate 'Location = Austin' with the class Company, and the predicate 'Salary > 50000' with the class Employee. The query returns the Name of Employee and Name of Company from the nested Employee objects that satisfy the query conditions.

Third, the syntax of the unified query language must be upward–compatible with SQL [7]. This is simply for commercial viability; by casting the language in the familiar SQL–compatible syntax, it makes it easier for the relational users to adopt the object–oriented technology embedded in the unified database system. Of course, mere "SQL–like" syntax is not sufficient; the language must be "SQL–compatible" by adhering to the design rationale behind SQL. For example, the path query discussed above may be expressed in the SQL/X query language as follows.

```
SELECT  *
FROM    Employee
WHERE   Salary > 50000  AND
        WorksFor.Location = "Austin" ;
```

The dot notation in the predicate (WorksFor.Location = "Austin") extends the standard predicate expression to account for the nesting of attributes through the use of arbitrary data types. It should be easy to see that a method defined for a class may be included in a predicate as a part of the query conditions.

Fourth, just as relational database systems provide automatic optimization of queries [26], so must a unified database system. Simply put, the unified database system must not require any hints from the user to determine the most efficient way to evaluate any given query, and it must be able to automatically determine the best way to process any given query utilizing all access methods supported in the system. One of the more serious problems in some of the current object–oriented database systems is the lack of automatic query optimization of queries.

There are a few important open areas of research in query support in a unified database system (or object–oriented database systems). One is views and another is set–valued attributes. Just as views are a useful mechanism in relational database systems, they certainly are an important feature in a unified database system. Although research has begun to define the semantics of views in an object–oriented database [1, 18], serious further research is necessary. It is relatively straightforward to define a view using the syntax of a query language. However, various fundamental issues have not been addressed; for example, whether views may belong to a class hierarchy, and views may be used as domains of attributes. Further, support for set–valued attributes needs additional research. It is certainly not enough to allow the specification of set–valued attributes in the database schema. It is important to support both homogeneous and heterogeneous set–valued attributes. Further, query–based updates to set–valued attributes must be supported.

Data Control

The data control language consists of three sets of commands dealing with access method management, authorization management, and transaction management. Access methods are data structures that the user creates and the system maintains to expedite the processing of queries. Relational database systems support B–tree or hash–based indexes on user–specified columns of a table. The authorization mechanism makes it possible for the creator of a database to dynamically grant and revoke access privileges to other users. The transaction mechanism enforces database integrity for a sequence of database reads and writes despite system failures and even simultaneous access to the same database by more than one user.

A database system is much more than just a persistent storage manager that supports retrieval of objects one at a time. A unified database system must satisfy two requirements related to data control. First, it must support all major database control features supported in relational systems. Second, it must augment the data control facilities such that they are consistent with the semantic extensions necessitated by the data model. Beyond these, it should include additional features that have long impeded usefulness of the current database systems, including memory–resident data management, version management, and long–duration transaction management. Let us look at the second requirement more closely.

authorization management

Just as relational database systems, a unified database system must provide facilities for the creator of a database to dynamically grant and revoke access privileges on his database to other users. The authorization mechanism in relational database systems define three

aspects of an authorization: authorization subject, authorization type, and authorization object. The authorization subject is the user; typically, it may be a single user or all users (PUBLIC). The authorization type is the type of privilege, including read, update, alter table definition, and create index. The authorization object is either a table, a column of a table, or a view. For example, a user, Smith, may grant to another user, Brown, authorizations to read and update all but the Salary Column of his (Smith's) Employee table.

The unified data model requires extensions to the relational authorization mechanism. First, the authorization type needs to be expanded to include authorization to execute methods. Second, the authorization object needs to include a hierarchy of classes, rather than only one class.

Third, perhaps most importantly, there is now a legitimate need to expand the authorization object to include a single instance of a class [23]. This is because an object–oriented data model accords an individual instance the status of access unit. In relational databases, the access unit is a table; that is, the only way for an ordinary user to access a relational database is to issue a query, and a query is always targeted to one or more table, but never to a single row of a table identified by its "system–wide unique row identifier". In a unified database, an instance is assigned a system–wide unique identifier, making an individual instance the unit of database access. Further, in applications that manage a relatively small number of large objects (e.g. book chapters, microprocessor chip designs, building designs), it makes sense to allow the creator of database objects to grant authorizations on individual objects. Unless a great care is taken in their implementation, object–level authorizations can cause a significant performance overhead to the normal operation of the system.

transaction management

At the minimum, the same mechanisms supported in relational (and pre–relational) database systems for concurrency control and recovery can be used in a unified database system.

The page–level locking technique that is used to implement concurrency control in current database systems, however, is inadequate. For precisely the same reasons that an individual instance of a class should be made an authorization object in a unified database system, it should also be the locking unit. In other words, a unified database system should really support object–level locking, comparable to record–level locking in some relational systems.

There is a major difficulty with the transaction mechanism in current database systems. The transaction mechanism currently supported is appropriate primarily for short–duration canned transactions, and is seriously inadequate for long–duration, cooperative transactions occurring in interactive design and engineering environments. This is a serious problem that must be addressed in future database systems, regardless of the data model they may adopt. The solution is expected to consist of a number of separate mechanisms, such as versioning, checkout/checkin, public/private databases, and triggering [23]. There is no database system at this time that offers a satisfactory general mechanism for long–duration transactions.

access method management

Relational systems resort to two basic techniques to enhance performance: access methods and clustering. They allow the users to create B–tree or hashed indexes on an attribute (or a combination of attributes) of a table, and use them in processing queries. Further, they store rows of the same table physically close together on secondary storage to speed up queries that require a large subset of the rows to be searched; or store rows of more than one table close together to speed up queries that involve joins of the tables.

A unified database system must adopt all these facilities. Further, the richness of the database definition calls for additional facilities. First of all, it must include a dynamic loader and linker of methods, beyond just the facilities to manage stored data. Second, it needs a solution to the problem of quickly retrieving any object given its object identifier. If a logical identifier is assigned to an object, at the minimum a hash–based index for mapping the identifiers to the physical addresses of the objects must be maintained. Third, the system may need new types of index to process path queries and class–hierarchy queries more efficiently than with the conventional index [23].

2.3 Computation

One class of applications, for example, computer–aided design simulators and design layout routers, needs to do extensive pointer chasing, that is, traverse a large number of objects, recursively, from one object to related objects. To support this type of applications, the database system must fetch objects into memory, transform the logical link between one object to another into a memory pointer, and make the memory pointer available to the application. Relational database systems certainly are not designed to support this type of applications; to fetch a row of a table T logically linked to a row of another table S requires the processing of a query on table T with the key found in the row of table S. In fact, pre–relational systems are better able to support this type of applications.

Implementors of some of the first wave of object–oriented database systems have taken pains to optimize the performance of this aspect of their systems, namely, management of memory–resident objects. These systems are designed to store object links as object identifiers, and convert them to memory pointers when the objects are loaded. The techniques used to manage and map objects between the memory and the database are similar to those used to implement persistent storage system for programming languages [17, 4].

This facility to support pointer chasing through a large number of memory–resident objects is a key computational requirement in an important class of database applications [24]. If a unified database system is to support this class of applications, it must include this facility as well as the cursor–based fetch of a set of query result objects which is supported in relational systems.

3 Concluding Remarks

Incorporating object–oriented technology in a database system holds the potential for the next productivity leap in application development. However, there is a formidable array of technical challenges to implement a unified relational and object–oriented database

system. These challenges include the design of a database language which is upwardly compatible with the relational language, definition of the semantics of database definition and changes, implementation of authorization and concurrency control to individual object level, and so on. Further, a unified database system needs to go beyond merely supporting the object–oriented paradigm. It needs to remove some of the major deficiencies in data modeling and computation modeling in current relational database systems. These deficiencies include versioning, long–duration transaction management, support for applications that require extensive pointer–chasing through a large number of objects, and so on.

UniSQL/X is a commercial database system that faithfully unifies the relational and object–oriented data models. The design of the system addresses most of the issues that need to be addressed in implementing a unified relational and object–oriented data model. UniSQL/X implements SQL/X [18], an upwardly compatible extension to SQL. SQL/X, just as SQL, is designed to be a full–blown database language consisting of a data–definition language, query and data–manipulation language, and data–control language. UniSQL/X automatically determines an optimal way to evaluate any given query and data–manipulation statement; and provides automatic concurrency control and recovery facilities for multiuser support and failure tolerance. A major feature of UniSQL/X is the extended framework for the management of multimedia data and multimedia devices (input, output, and storage); it goes far beyond the object–oriented framework inherent in any database system that supports an object–oriented data model.

The current release of UniSQL/X will be ported to a range of UNIX platforms, and selected non–UNIX environments. Subsequent releases of UniSQL/X will include support for long–duration transaction management, enhancements to the already sophisticated query facilities, and additional features for higher performance and additional support tools for application development.

References

1. Abiteboul, S., and A. Bonner. "Objects and Views," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Denver, Colorado, May 1991, pp. 238–247.

2. Andrews, T., and C. Harris. "Combining Language and Database Advances in an Object–Oriented Development Environment," in *Proc. Intl. Conf. on Object–Oriented Programming Languages, Systems, and Applications*, Orlando, FL., October 1987, pp. 430–440.

3. American National Standards for Information Systems. *Database Language SQL.* ANSI X3.135–1986, October 1986.

4. Atkinson, M., et al. "An Approach to Persistent Programming," *Computer Journal*, vol. 26, no. 4, November 1983, pp. 360–365.

5. Atkinson, M., et al. "Object–Oriented Database System Manifesto," in *Proc. 1st Intl. Conf. on Deductive and Object–Oriented Databases*, Kyoto, Japan, Dec. 1989.

6. Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object–Oriented Databases," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, Calif., May 1987.

7. Beech, D. "A Foundation for Evolution from Relational to Object Databases," in *Proc. Intl. Conf. on Extending Data Base Technology*, Venice, Italy, March 1988.

8. Bretl, R., et. al. "The GemStone Data Management System," *Object–Oriented Concepts, Applications, and Databases*, (ed. W. Kim, and F. Lochovsky), Addison–Wesley, 1989.

9. Carey, M., D. DeWitt, and S. Vandenberg. "A Data Model and Query Language for EXODUS," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Chicago, Ill., June 1988, pp. 413–423.

10. Cluet, S., C. Delobel, C. Lecluse, and P. Richard. "Reloop: An Algebra–Based Query Language for an Object–Oriented Database System," in *Proc. 1st Intl. Conf. on Deductive and Object–Oriented Databases*, Kyoto, Japan, Dec. 1989.

11. Codd, E.F. "A Relational Model for Large Shared Data Banks," *Comm. ACM*, vol. 13, no. 6, October. 1970, pp. 377–387.

12. Codd, E.F. "Extending the Relational Model to Capture More Meaning," *ACM Trans. on Database Systems*, vol. 4, no. 4, Dec. 1979.

13. Date, C. *An Introduction to Database Systems* (4th edition), Addison–Wesley, Reading, Mass., 1986.

14. Deux, O., et al. "The Story of O2," *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

15. Fishman, D., et al. "Overview of the IRIS DBMS," *Object–Oriented Concepts, Applications, and Databases*, (ed. W. Kim, and F. Lochovsky), Addison–Wesley, 1989.

16. Goldberg, A. and D. Robson. *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, Reading, MA 1983.

17. Kaehler, T. "Virtual Memory for an Object–Oriented Language," *BYTE*, pp. 378–387, August 1981.

18. Kifer, M., W. Kim, and Y. Sagiv. "Querying Object–Oriented Databases," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, Calif., June 1992.

19. Kim, W. "A Model of Queries for Object–Oriented Databases," in *Proc. Intl. Conf. on Very Large Data Bases*, August 1989, Amsterdam, Netherlands.

20. Kim, W. et al. "Architecture of the ORION Next–Generation Database System," *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

21. Kim, W. "Architectural Issues in Object–Oriented Databases," *Journal of Object–Oriented Programming*, March/April 1990.

22. Kim, W. "Object–Oriented Databases: Definition and Research Directions," *IEEE Trans. on Knowledge and Data Engineering*, Sept. 1990.

23. Kim, W. *Introduction to Object–Oriented Databases*, MIT Press, November 1990.

24. Maier, D. "Making Database Systems Fast Enough for CAD Applications," *Object–Oriented Concepts, Applications, and Databases*, (ed. W. Kim, and F. Lochovsky), Addison–Wesley, 1989.

25. *Release 1.0*, Sept. 1989, EDventure Holdings, Inc., 375 Park Ave., New York, NY.

26. Selinger, P., et. al. "Access Path Selection in a Relational Database Management System," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Mass., May 1979, pp. 23–34.

27. Shaw, G., and S. Zdonik. "Object–Oriented Queries: Equivalence and Optimization," in *Proc. 1st Intl. Conf. on Deductive and Object–Oriented Databases*, Kyoto, Japan, Dec. 1989.

28. Smith, J., and D. Smith. "Database Abstraction: Aggregation and Generalization," *ACM Trans. on Database Systems*, vol. 2, no. 2, June 1977, pp. 105–133.

29. Stefik, M. and D.G. Bobrow. "Object–Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40–62.

30. Stonebraker, M., and L. Rowe. "The Design of POSTGRES," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Washington, D.C., May 1986.

31. Stonebraker, M., J. Anton, and E. Hanson. "Extending a Database System with Procedures," *ACM Trans. on Database Systems*, vol. 12, no. 3, Sept. 1987.

32. Stroustrup, B. *The C++ Programming Language*, Addison–Wesley, Reading, Mass. 1986.

33. Ullman, J. *Principles of Database Systems* (2nd edition), Computer Science Press, Rockvile, MD, 1982.

34. Woelk, D., and W. Kim. "Multimedia Information Management in an Object–Oriented Database System," in *Proc. Intl. Conf. on Very Large Data Bases*, Brighton, England, Sept. 1987, pp. 319–329.

35. Zaniolo, C. "The Database Language GEM," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, Calif., May 1983, pp. 207–218.